Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

Artificial
Intelligence
Laboratories

SEKI-REPORT

Expert-System Shells: Very-High-Level Languages for Artificial Intelligence

Harold Boley

SEKI Report SR-88-22

EXPERT-SYSTEM SHELLS: VERY-HIGH-LEVEL LANGUAGES FOR ARTIFICIAL INTELLIGENCE

Harold Boley, FB Informatik, Univ. 675 Kaiserslautern, Box 3049, W. Germany
boley@uklirb.uucp [1]

Abstract:

Expert-system shells are discussed as very-high-level programming languages for knowledge engineering. Based on a category/domain distinction for expert systems the concept of expert-system shells is explained using seven classifications. A proposal for a shell-development policy is sketched. The conclusions express concern about overemphasis on shell surfaces.

1. Introduction

Artificial intelligence has often inspired the development of new tools in computer science, both hardware and software. In particular, a large number of programming languages (processing kernels and interaction interfaces) have grown out of, or in connection with, AI projects.

The earlier development of functional, logical, and object-oriented programming languages supporting all subfields of AI (surveyed, e.g., in [Boley1983a]), is now accompanied by an explosive development of languages more specifically supporting the large AI subfield of knowledge engineering or expert systems (XPSs). Reaching for a language level that permits (non-programmer) experts of application domains to build simple XPSs, they are prototypes of what [Leavenworth1974a] anticipated as "non-procedural" or "very-high-level" languages.

Several terms are employed for these very-high-level expert-system languages, nine of which are represented by the following extended regular expression:

{knowledge engineering,expert system [building]}{tools,environments,shells}

Note that the term 'language' is not normally used, perhaps because these interface-intensive very-high-level languages are thought to be more than languages (whereas we adopt the definition "language = kernel + interface") or because there is some unfamiliarity with the tradition of very-high-level and AI languages.

The term "expert system shells" -- induced from "emptied" expert systems with interchangeable knowledge bases -- tends to be most often employed as the principal one, and we will follow this use here. A disadvantage of this convention is the term's too narrow connotation with "knowledge-emptied expert systems"; an advantage is that it can be shortened to

[1] This research was done on a stay as a visiting scholar at Siemens AG

"shells" without clashing against many other computer-science terms (except the clash with UNIX [2] shells).

A shell consists of three principal parts, the first two constituting the shell's kernel, inherited from OPS/PLANNER/PROLOG-like (forward/backward rules) and FRL/KRL/KL-ONE-like (frames/nets) AI languages, the third constituting the shell's interface, developed much further than in these AI languages:

(1) Knowledge base, containing both data-like and method-like knowledge representations.

(2) Inference engine, mechanically applying the methods to infer new knowledge from given knowledge.

(3) User interface, with knowledge browsing/acquisition and inference tracing/explanation features utilizing modern graphics technology.

Of course, this usual break-down of shells into three components oversimplifies reality. In particular, the "(augmented) truth-maintenance system" of many shells cannot be easily subsumed under either the knowledge base or the inference engine alone but rather belongs to the entirety of what we call the shell kernel. Also, besides the user interface most widely-used shells now have what may be called a "conventional data-processing interface", subdivided into interfaces to the operating system, to a bulk database, to a management-information system and/or the like.

Several surveys of shells have appeared in the literature, normally with a detailed treatment of the user interface ([Richer1986a], [Gevarter1987a], [Citrenbaum1987a], [Igney1987a], [Karras1987a], [Harmon1988a]).

The present paper directs most attention to the shell kernel. It attempts to characterize the shell concept by introducing a system of classifications such as special-purpose vs. general-purpose shells. Because of the kinship between XPS shells and AI classics like LISP, PROLOG, and SMALLTALK we suggest a policy based on a more coordinated and integrated development of both varieties of languages.

2. The category and domain of an XPS

Before turning to XPS shells we should clarify some aspects of the XPS concept itself. Several distinctions are used for describing XPSs in the literature, but a clear differentiation of what we feel are two basic dimensions is not made very often.

One important dimension is the category of an XPS [Waterman1986a], called "function capabilities" in [Gevarter1987a], specifying whether the XPS is applied for diagnosis, configuration, planning, etc.

---

[2] UNIX is a registered trade-mark of Bell Laboratories

The other important dimension is the domain of an XPS, specifying whether the XPS is applied in medicine, engineering, administration, etc.

Frequently these dimensions are put under the same vague heading of XPS "application", ignoring what we emphasize as the difference between "application for" and "application in", or, more technically, as category and domain, respectively.

Indeed, we consider these dimensions as almost-orthogonal, with all combinations being possible in principle, as in the following example (the second combination may be problematic):

Application	for diagnosis	in medicine.
Application	for configuration	in medicine.
Application	for planning	in medicine.
Application	for diagnosis	in engineering.
Application	for configuration	in engineering.
Application	for planning	in engineering.
Application	for diagnosis	in administration.
Application	for configuration	in administration.
Application	for planning	in administration.
	etc.	etc.

If this "Application for category in domain" schema is employed systematically, unusual -- perhaps still unexplored -- applications may arise. However, we describe the dimensions as "almost-orthogonal" only, because (a) categories like 'configuration' and 'planning' can also be regarded as one larger group that may be circumscribed as 'design' (while configuration stresses "space designs", planning stresses "time designs"), and (b) independent variation of category and domain occurs rarely in practice of human expertise (an architect, say, is not only an expert "for design in housing" but can also be employed -- with graceful performance degradation -- both "for diagnosis in housing" and "for design in shipping").

Regarding XPS architecture, the 'category' dimension correlates with the inference engine, while the 'domain' dimension correlates with the knowledge base. The distinction is interesting not only because of systematics but also because it draws attention to two complementary ways of proceeding from individual XPSs to "metasystems":

(1) If the inference engine is kept fixed and the knowledge base is varied a metasystem of XPSs for the same category in different domains can be constructed.

(2) If the knowledge base is kept fixed and the inference engine is varied a metasystem of XPSs in the same domain for different categories can be constructed.

The former possibility leads to those well-known metasystems called "shells" (to be explored in the following sections), the latter to new metasystems that might be called "pools" (only discussed briefly below).

A pool is a knowledge base in a particular XPS domain, say UNIX peripherals (printers, terminals, drives), that can support several XPS categories (say diagnosis and repair, configuration and planning) implemented by several inference engines (say forward reasoning, backward reasoning). The advantage of sharing the knowledge base for several XPS categories (inference engines) may be at least as big as that of sharing the inference engine for several XPS domains (knowledge bases); this may become increasingly obvious as the domain representations of XPSs evolve from shallow rules to deep models (cf. section 3.7). For example, we could augment the knowledge base of the printer diagnosis/repair XPS micro-UNIXPERT.M [Lessel1988a] for printer configuration -- reusing much of the knowledge acquired in micro-UNIXPERT.M -- and add inference mechanisms for configuration to obtain a printer configuration XPS. An exciting possibility arising with shared knowledge bases is XPSs cooperating in the same domain: in our example, after each printer reconfiguration the diagnosis XPS could access the updated knowledge generated by the configuration XPS.

3. A classification of shells

In the following subsections we give seven classifications of XPS shells, some related to criteria for AI programming-language design.

3.1. Academic and commercial shells

Shell development began in academia, as an offspring of the XPS MYCIN at Stanford University (EMYCIN) and an engineered version of the cognitive model of production systems at Carnegie-Mellon University (OPS5).

The development quickly continued commercially, joining with the work on graphic man-machine interfaces at XEROX PARC (LOOPS); today commercial shell development has an explosive character, fueled by the workstation and PC markets.

Some academically initiated shells were later extended and marketed commercially by AI startup companies, often recruiting people from the original shell-developer team. The academic/commercial distinction is not clear-cut also for another reason: Some universities are distributing their shells at prices ($100 to $500) that are usual for commercial PC shells.

3.2. Mainframe, workstation, and PC shells

While the first shells were run on mainframes (which today have a 'comeback' due to the "conventional data processing (DP) integration" trend, sketched in section 4.2), the rapid rise of the memory and speed characteristics of PCs -- up to those based on MOTOROLA 68020 and INTEL 80386 processor chips -- has permitted to run shells on PCs for smaller XPS applications. Often PCs are used for special-purpose shells (cf. section 3.5) and for delivery versions. The boundary between PCs and workstations is becoming increasingly hard to discern, and may vanish with systems based on processors like MOTOROLA 68030.

Today, however, the major hardware basis for shells is the intermediate computer class of workstations such as SUN, APOLLO, IBM-RT, and

HP-9000/350, some based on RISC technology. They are now expanding both 'up' (towards mainframes) and 'down' (towards PCs).

Competitors of workstations in AI are both LISP machines and PROLOG machines, which provide shells based on LISP and PROLOG with high computing power. Those using a special chip such as the TI Explorer II with its LISP chip are particularly suited for large XPSs. The price/performance ratio is improved by combined systems, like the microExplorer or MacIvory, using a LISP chip, like TI's or SYMBOLICS', as a coprocessor for a high-end PC, like Apple's Mac II.

LISP machines allied with high-end PCs are thus trying to counter the penetration of workstations into the LISP and shell domains. Since workstations share the universality of mainframes, they are favored by the "conventional DP integration" trend and by the ensuing tendency to replace LISP by C as the shell base language (cf. next subsection).

3.3. PROLOG-, LISP-, and C-based shells

While most shells are still written in LISP (some in PROLOG), there is presently a strong tendency towards C as the implementation language for XPS shells. C is favored by the de facto operation-system standard UNIX, by the "conventional DP integration" trend, and by the shell marketing strategy of squeezing out the last bit of efficiency from computers.

LISP has a good chance to stand this battle if the de facto standard COMMON LISP can quickly be developed into an ISO standard, and if excellent interfaces between the LISP standard and conventional DP standards (including C) are created. It would be ironic if now that there exist really fast LISP implementations (on chip and otherwise), AI's language of choice would be replaced by the (powerful but low-level) systems programming language C in commercial AI. The constant factor of speed improvement should become smaller as ever more optimizing LISP compilers are developed, making the much higher development and maintenance cost of C-based shells less and less acceptable (even inside the C community there is a recent trend towards higher-level variations such as C++). The time until LISP will completely dominate COBOL or FORTRAN in AI is called the "LISP Gap" in [Harmon1988a]. However, it is very hard to estimate the real amount of time up to a possible break-even point such as the one between LISP and C. If break-even would occur within the next few years it would be clearly unwise to 're-re-code' a big shell from LISP to C to LISP. Perhaps LISP-to-C translators like those offered by C-Lambda could further help avoid the need for people again having to write (all of) big XPS shells in C manually.

The situation regarding PROLOG -- not as a shell component, but as the shell-implementation language -- is similar, but perhaps more serious because of PROLOG's higher level (greater distance to C), which is at even greater variance to current shell implementation-language trends.

3.4. End-user, knowledge-engineer, and AI-programmer shells

Users of an XPS shell can have entirely different amounts of training in AI. A shell can be targeted to the large market of untrained, inexperienced end users (people working in any XPS domain, usually not computer science) or to the (still) small market of highly-trained, experienced AI programmers.

End-user shells employ their base language (which can be an AI language like LISP, PROLOG, or SMALLTALK, or a conventional language like FORTRAN, PASCAL or C) merely as an implementation language for a fixed set of components. The shell is restricted to accept domain knowledge, not AI programs, from the user.

AI-programmer shells employ their base language (which must be an AI language) as a programming platform with an open set of preprogrammed shell components. These are accessible as separate modules, like individual tools of a tool box (program library), and can also be extended by programming new tools for oneself, which are then accessed in the same manner. Subsets of these tools can be clustered as demanded by the XPS being built.

In between the end user and the AI programmer there is often a knowledge engineer trained to acquire largely unstructured domain knowledge from a human expert and to formalize it for input into a shell's knowledge representation(s).

Many shells have interfaces for all three kinds of users. However, commercial pressure seems to direct evolution towards colorful, menu-driven, 'bells-and-whistles' end-user shells, often not modifiable on an AI programming level. The recent JOSHUA language (described in [Rowley1987a] and [Schuelting1988a]) again seems to proceed in the opposite direction of powerful, LISP-integrated, 'wheels-and-engine' AI-programmer shells.

3.5. Special-purpose and general-purpose shells

Relating to the distinction of end-user and AI-programmer shells in the previous subsection, there are two principal ways of constructing an XPS shell.

First, it is possible to take an XPS directly implemented in an AI base language like LISP and empty it of (some or) all the knowledge stored in the knowledge base, thus leaving (the reusable knowledge and) the representation formats, the inference engine, and the user interface. This (more or less) empty "shell" -- in the original sense of the word -- can be refilled with knowledge describing another (subdomain or) domain to obtain another XPS for the same category. The classic example of this procedure is the LISP-developed MYCIN XPS and its derivatives, and the shell EMYCIN [Melle1984a], later extended commercially, as for example in S.1/M.1.

Second, it is possible to use an AI base language to construct various components for knowledge representation, inferencing (forward, backward, inheritance), and user interface, as needed for rapid XPS building. This user "shell" -- in a new sense of the word -- can be employed to build an

XPS for an arbitrary category in an arbitrary domain. Presently the most well-known commercial examples, developed with LISP as the base language, are ART, KEE, and KnowledgeCraft, used to build several XPSs.

While the first way leads to special-purpose shells (to build XPSs for the same category in different domains), the second way leads to general-purpose shells (to build XPSs for different categories in different domains). Of course, there really is a whole scale of more or less general shells: after conception, special-purpose shells can be generalized, and general-purpose shells can be specialized. About in the middle of this scale are shells that can be used to build XPSs for an entire group of categories such as configuration and planning.

3.6. Uniform and hybrid shells

The knowledge representation -- and consequently the inference engine -- of a shell can be uniform or hybrid; in both cases the shell may still be special-purpose or general-purpose in the sense of the previous subsection.

A uniform representation employs one canonical formalism throughout, say Horn clauses only (PROLOG) or forward rules only (OPS5). (Although PROLOG and OPS5 are classic AI languages, they can be used like uniform shells.)

A hybrid representation employs two or more formalisms in various parts of the knowledge base, say both forward rules and frames, or all of Horn clauses, forward rules, and frames.

The obvious advantages of uniformity are:

(1) less learning overhead if the user wants to learn the entire shell,

(2) no need for inter-representation translation (either human or machine),

(3) avoiding the evolution of redundant, overlapping representations,

(4) no need for a selection methodology for the individual representations.

The complementary advantages of hybridness are:

(1) no learning overhead if the user can find a familiar representation among those offered,

(2) knowledge bases coded in one of the component representations can be easily imported from other shells,

(3) additional insights gained by multiple representational perspectives,

(4) the most natural and efficient representation can be chosen pragmatically for each problem.

There is an ongoing debate about the merits of both classes of shells. A recent tendency in academic shell-like reasoning systems such as KRYPTON is to restrict hybrid systems to exactly two representations, thought to be be fundamentally different (and, we might add, to subsume all other representations under either of the two). This minimum or "essence" of hybrid representations [Brachman1985a] consists of the so-called ABox (for Assertional, or logic, representation) and TBox (for Terminological, or frame, representation). Since "ABox+TBox" hybrids are related to "logic+sorts" combinations -- which can be reduced to uniform predicate logic by reducing sorts to unary predicates -- an "essential hybrid" representation is not far removed from a uniform representation. No well-known commercial shell seems yet to be based on the "essential hybrid" paradigm.

3.7. Shallow and deep shells

The inference engine of a shell can support shallow or deep inferences (reasoning) over a knowledge base of rules or models, respectively.

A shallow inference consists of the chaining of rules that associates premises (e.g. observed symptoms) with a conclusion (e.g. a hypothetic diagnosis).

A deep inference consists of the probing of a model that collects -- often qualitative -- properties (e.g. causal, temporal, and/or spatial ones) of the domain.

Older, established 'rule-based' XPSs and shells support shallow inferences. Newer, exploratory 'model-based' XPSs and shells support deep inferences.

In present AI, ("common-sense") theories of model-based reasoning in domains like mechanical and electrical engineering as well as qualitative representation of time and space belong to the most active research areas. The combination of shallow and deep reasoning is also being attempted [Gladel-Speicher1988a].

While shallow reasoning can be realized in a natural fashion using forward rules (of production systems like OPS5) or backward rules (of Horn-clause languages like PROLOG), deep reasoning can be realized easily using frames (as in FRL) or objects (as in SMALLTALK or CLOS). The combination of shallow and deep reasoning would then call for a combination of forward/backward rules with frames/objects (as in many present-day shells).

The trend from shallow inferences to deep inferences may be an important reason for a present tendency from PROLOG back to LISP and SMALLTALK: the direct support of (backward) rules in PROLOG appears no longer as important, while the direct support of frame-like property lists and object-oriented extensions in LISP and the objects in SMALLTALK are becoming more important. However, it should be noted that it is quite easy to add frame-like structures and object-oriented message passing to PROLOG (see [Lee1986a] for one example). The present SMALLTALK-80 may turn out as an interim solution since a powerful standard for object-oriented programming extending COMMON LISP has finally been achieved with CLOS [Bobrow1988a].

4. A policy of shells

In the subsections below we make some proposals for a policy of shell development.

4.1. In-house development

There is a number of reasons why an organization -- if it has programmers who are capable of programming in languages up to the level of LISP or even PROLOG -- should not blindly rely on any shell product, and afford a (perhaps small) in-house AI-language or shell-development group (even after deciding to purchase a large shell):

(1) It is not clear from the outset whether a desired XPS is best programmed in raw LISP (PROLOG, SMALLTALK, ...), in an extension of such a language by a few shell components, or in a full-fledged shell; the in-house shell group should be consulted for each XPS.

(2) The selection of the right shell (not only suited for the first XPS) can be done much more securely if the purchasers have their own technical shell know-how, which can only be acquired through practical developments.

(3) A 'mini-version' of a shell -- showing many of the kernel concepts such as forward/backward rules and frames -- can be realized and studied in LISP or PROLOG with surprisingly little effort (cf. [Boley1987a]).

(4) Better, more efficient use can be made of a big commercial shell if experience in shell implementation concepts has been gathered, perhaps initially with a small model shell.

(5) If the knowledge bases of the big commercial shell and the small in-house shell are compatible or translation tools between them can be realized, less copies of the big shell need be paid because the unlimited number of copies of the small shell will be sufficient for a lot of (clerical) work during XPS development -- perhaps also for the XPS delivery version.

(6) The past has shown that even a large -- seemingly prospering -- AI firm can get into problems and stop the support of its products, in which case at least minimum maintenance should be transferrable to the in-house group.

If we accept the opinion of [Citrenbaum1987a], that no (and we might add, currently existing) shell is likely to have "all the attributes needed for all purposes", we may well come to the conclusion that every XPS-developing organization must have these crucial abilities: to evaluate shell tenders down to the implementation level, to adapt shells acquired with a source-code license for its own purposes, as well as to develop shell components, configure a shell from given components, and if necessary even program a new shell on its own.

4.2. Conventional DP integration

After the "first wave" of enthusiasm about AI products has gone, these days it seems very difficult to market "stand-alone AI systems" rather than the more and more important "embedded AI systems". In particular, XPS shells must now be integrated with conventional data-processing systems in order to successfully compete with "trend-setter shells" that already did this move into mainstream DP.

Purely academic AI researcher may worry about whether "the real AI" might get lost in the floods of this mainstream, or become transformed beyond recognition. In fact, a marketing recommendation like "hide the AI component of your system" may sound discouraging to AI students contemplating about their career, especially if they feel themselves "close to the solution of the central AI problem".

However, in our opinion efforts to improve the true intelligence of a system need not be corrupted by efforts to interface it with all sorts of unintelligent systems. An AI system, like a human, is free to make intelligent use of operating-system utilities, bulk databases, numerical simulation packages, and so on, building a pyramid of unconventional reasoning methods on a platform of conventional DP.

Perhaps the most important DP interface of XPS shells is the one to bulk databases because knowledge bases in main memory are often too small for realistic XPSs. Translation procedures between XPS knowledge bases and conventional bulk databases should be employed to smoothly switch between (unformatted) knowledge and (formatted) data representations of the domain. In the case of PROLOG-like clausal knowledge and SQL-like relational data this is quite easy: several PROLOG vendors offer relational database interfaces.

4.3. Standardization

A suggestion to provide translation facilities to a standard shell knowledge base in order to gain portability is made in [Citrenbaum1987a]. The paper proposes that the standard would have different forms for rules, schemata, and so forth.

For uniform shells as discussed in section 3.6 such a standard would become a complete standard of the knowledge base. Presumably it will not be a trivial matter to agree on a uniform formalism for a knowledge-base standard because this depends on resolving old but still open AI issues such as the declarative-procedural debate (today reappearing as a logic-oop debate). Whatever a possible standard may look like, a knowledge base having a clear, simple, uniform ASCII representation will be more easily translated to the knowledge bases of other shells than those without such representations. (ASCII will also facilitate knowledge transmission via networks such as ETHERNET.)

Regarding the large number of shells coming on the market every year, each developer of a new shell should provide "portability links" as part of the documentation: they should show which (parts of) knowledge bases of other

well-known shells can be imported -- perhaps, with which tools -- and to which ones they can be exported. Perhaps an entire "portability web" could thus evolve, making users feel more comfortable when investing into a new shell.

Besides the central standardization issue for knowledge bases (cf. our 'pool' concept in section 2), there is a related issue for inference engines, and a separate standardization theme for the user interface.

In accordance with the "conventional DP integration" trend, a shell standard for the user interface should be compatible with general interface standards, e.g. the developing X-Windows standard.

Similarly, interfaces to other conventional DP subsystems such as databases and operating systems (say SQL and UNIX, respectively) should of course follow the standards established there.

Finally, for the shell implementation language a de facto standard such as COMMON LISP (incl. CLOS), CPROLOG, SMALLTALK-80, or C(++) should be chosen; hopefully, the forthcoming ISO standards for some of these languages will encourage choices of such a kind. A "conservative use" of a very large language such as COMMON LISP, restricting it to a non-controversial subset, may be a good method to ensure conformity with its eventual standardized version.

4.4. User organization

The explosive development of XPS shells confronts prospective users with a bewildering 'zoo' of originals and variants, choices and pressures, trend and hype.

One useful perspective on shells is that of 'commercialized programming languages':

(1) Since the main developmental effort was invested by companies -- based on earlier R & D done by universities -- shell concepts are more often presented in advertising-marketing contexts than in critical-scientific contexts.

(2) Shell-implementation techniques are not often treated in detail in publications, since competition encourages confidential and proprietary procedures.

(3) In their desire to be the best of all, compatibility with other shell vendors has not (yet) been a widely recognized design goal of companies.

Although there is a number of journals and newsletters watching the shell market, these should be complemented by a more interactive way of information exchange. Perhaps a moderated UUCP newsgroup on XPS shells, [Bachmann1988a], will be a good medium. (Occasionally there are shell contributions in comp.ai.digest, but this group already has a lot of traffic.)

Similarly, no vendor-independent, international "user organization for XPS shells" seems to exist presently; perhaps it could be formed in connection with the shells newsgroup. Among other things, it could distribute public-domain shells (cf. NASA's COSMIC at the University of Georgia), develop shell benchmarks (perhaps somewhat more complex than NASA's Monkey & Bananas benchmark), organize shell contests (unpublished draft), compile shell bibliographies (see appendix), maintain price/performance comparisons, and work towards a shell standard (see previous subsection).

The field of XPS shells seems to be ripe for some kind of coordination along these lines. XPSs and shells have become so important in commercial AI that academic AI must be concerned about their further development, which may well be critical to the future of AI as a whole.

5. Conclusions

The hype surrounding XPSs (including shells) has often been criticized inside and outside of AI. Indeed, some shell names, advertisements, and presentations can make laymen think of a glittering 'magicians box': Windows are created 'out of thin air' in the midst of the screen with a mouse click (e.g., to browse into some structure) or even surface automatically (e.g., to signal an error), animated color graphics move and blink and transmute much like in a video game, and sometimes all the bells ring and all the whistles blow. At the peak of the XPS hype it would not have been a great surprise if a new company would have advertised a forthcoming product called 'The Magic Shell' in which the rectangular shape (the remaining symbol of pure rationality) of windows was replaced by asterisk-shaped 'star windows', the mouse was replaced by a magic-wand-like 'laser pen', the flat screen was replaced by a crystal-ball-like 'holographic screen', and an important inference was announced (and possible errors were concealed) not only by verbose voice output but also by emitting a little cloud of blue smoke.

Non-computer scientists watching a shell presentation may well feel like an audience that is puzzled by the tricks of a stage magician [3] because it cannot tell the 'kernel' events below the thick layers of the 'surface' show. The animated objects on the screen give only a superficial picture of the operational semantics of the real computation going on inside the shell; the mystified layman observing a shell screen is in a situation still worse than the observer in Plato's "cave" parable because the meaning of people's shadows is deciphered more easily than that of abstract object's screen images.

We do not want to question the value of graphic visualization and direct manipulation of abstract structures and processes if the users are trained appropriately (actually, we argue for iconic programming elsewhere). In particular, more consciousness should be developed about what is the

[3] After writing this section, a report from the Industrial Exhibition of AAAI-88 in St. Paul, Minn., showed how reality can surpass fiction: The main attraction was a magician employed by a company for advertising.

external knowledge presentation and what is the internal knowledge representation. Otherwise, it seems that overloading a shell interface with impressive but useless graphics can conceal the real -- perhaps too trivial? -- computation (in hybrid shells already the kernel does have a baroque architecture). Also -- and perhaps even worse -- the beauty of its surface can lead to a superficial use of a shell: playing with all these things can prevent the 'hypnotized' users from having their work done on the expensive equipment. Thus, clear logical-linguistic thinking may be slowly replaced by cloudy illogical-visual imagining. Early signs of this can be discerned for instance in universities when watching students 'doing graphics' on PCs or workstations.

Even though modern computer support of imagination is important in the creative, 'brainstorming' phases of a project, classical computer support of thinking is still important in its analytical, criticizing phases. Otherwise, excessive computer use may have a 'brainwashing' effect similar to the one we must now witness in the abuse of TV and video (games).

Unfortunately, when presenting a shell to a large audience, people watching the screen from a greater distance can observe almost only the 'graphics events' (unless a beamer is employed). This is one more reason seducing vendors to give 'shallow presentations'. Things can become embarrassing, however, both for some stage magicians and shell presenters. Towards the end of a graphics-oriented shell presentation there once came the simple suggestion "Would you please show us how to define factorial in your formalism" from the audience; the trivial recursive function could not be written down by the shell company's representatives.

Thus, while some years ago graphic surfaces were only an addendum to AI systems, these days the relation could well become inverted: At least the large number of programmers new in very-high-level languages or AI (XPSs) having to decide on an XPS shell -- with support from their more playful managers -- may be dazzled so much by a shell surface that they purchase almost any shell kernel along with a pretty surface.

Therefore, this paper has tried to show that there is quite a lot of space for variation in the knowledge base and inference engine. Hopefully, open kernel issues such as "essential hybrid" systems or suitable combinations of backward-chaining and forward-chaining strategies will play an increasing role in future shell-purchase decision making.

Like other successful fields that originated in connection with AI, modern workstation graphics has already become a mainstream computer-science area on its own, and even if graphics is a necessary sales argument for XPS shells, to let it look more and more like a sufficient argument means to push one's shell outside the market of artificial-intelligence products. Blurring the difference between graphics tools and AI systems may result in short-term AI success through graphics success, but the AI field may have to pay the long-term price of loosing its identity.

Acknowledgements

I want to thank Siemens AG, ZTI (now: ZFE F 2), Munich, for the inspiring working environment. In particular, SYS 5 and SOF 1 are thanked for their hospitality. This paper was written while I was hosted by the AIL-Clearingstelle, profiting from discussions with Harald Lausecker, Claus Jaekel, and Wolfgang Weber. Claus and Wolfgang also suggested improvements to an earlier draft. Moreover, I want to thank David M. W. Powers, Macquarie University, for helpful comments on the final draft. Of course, I have the sole responsibility for any remaining omissions and errors, and I apologize for them. All opinions expressed in this paper are my own.

References

Bachmann1988a.
Bernd Bachmann, Harold Boley, Norbert Kratz, Robert Rehbold, Michael M. Richter, Peter Spieker, and Thomas Wetter, "An Informal Proposal for a Newsgroup on Expert System Shells," Submitted to mod-ki and comp.ai.digest, Univ. Kaiserslautern, FB Informatik (Dec 1988).

Bobrow1988a.
Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon, "Common Lisp Object System Specification," Document 88-002R, X3J13 (June 1988).

Boley1983a.
Harold Boley, "Artificial Intelligence Languages and Machines," Technology and Science of Informatics Vol. 2(3) pp. 137-158 (May-June 1983).

Boley1987a.
Harold Boley, "Frame and Heir: Clausal Frames and Multiple Inheritance in LISPLOG," SEKI Working Paper SWP-87-09, Univ. Kaiserslautern, FB Informatik (Nov 1987).

Brachman1985a.
Ronald J. Brachman, Victoria Pigman Gilbert, and Hector J. Levesque, "An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of KRYPTON," IJCAI-85, pp. 532-539 (Aug 1985).

Citrenbaum1987a.
R. Citrenbaum, J. Geissman, and R. Schultz, "Selecting a Shell," AI Expert Vol. 2(9) pp. 30-39 (Sep 1987).

Gevarter1987a.
William B. Gevarter, "The Nature and Evaluation of Commercial Expert System Building Tools," IEEE Computer, pp. 24-41 (May 1987).

Gladel-Speicher1988a.
Simone Gladel-Speicher, "Vergleichende Untersuchung der Integrationsproblematik von Diagnoseexpertensystemen und Tiefenmodellierungssprachen am Beispiel verschiedener, bereits existierender Systeme," SEKI Working Paper SWP-88-04, Univ. Kaiserslautern, FB Inform. (Mar 1988).

Harmon1988a.
Paul Harmon, Rex Maus, and William Morrissey, Expert Systems Tools and Applications, John Wiley & Sons, New York (1988).

Igney1987a.
Karin Igney, "Hybride Expertensystem-Shells: Konzepte und anwendungsbezogene Evaluationskriterien," Diplomarbeit, TU Muenchen, Institut fuer Informatik (Nov 1987).

Karras1987a.
Detlef Karras, Lutz Kredel, and Uwe Pape, Entwicklungsumgebungen fuer Expertensysteme. Vergleichende Darstellung ausgewaehlter Systeme, Walter de Gruyter, Berlin, New York (1987).

Leavenworth1974a.
B. Leavenworth and J. Sammet, "An Overview of Nonprocedural Languages," SIGPLAN Notices Vol. 9(4)B. Leavenworth (Ed.): ACM SIGPLAN Symposium on Very High Level Languages, (Mar 1974).

Lee1986a.
N. S. Lee, "Programming with P-Shell," IEEE Expert Vol. 1(2)(Summer 1986).

Lessel1988a.
Michael Lessel, "Modellbasierte Diagnose von Benutzer- und Systemsoftware-Fehlern, dargestellt am Beispiel des UNIX-Spoolingsystems," Diplomarbeit, Univ. Kaiserslautern, FB Informatik (Oct 1988).

Melle1984a.
W. van Melle, E. Shortliffe, and G. Buchanan, "EMYCIN: A Knowledge Engineer's Tool for Constructing Rule-Based Systems," G. Buchanan and E. Shortliffe (Eds.): Rule-Based Expert Systems, pp. 302-313 Addison-Wesley, (1984).

Richer1986a.
Mark H. Richer, "An Evaluation of Expert System Development Tools," Expert Systems Vol. 3(3) pp. 166-182 (Jul 1986).

Rowley1987a.
Steve Rowley, Howard Shrobe, Robert Cassels, and Walter Hamscher, "Joshua: Uniform Access to Heterogeneous Knowledge Structures or Why Joshing is Better than Conniving or Planning," AAAI-87, pp. 48-52 (Aug 1987).

Schuelting1988a.
Heinz-Werner Schuelting, "JOSHUA, eine flexible KI-Sprache mit einheitlicher Oberflaeche," Preprint, Symbolics (1988).

Waterman1986a.
Donald A. Waterman, A Guide to Expert Systems, Addison-Wesley, Reading, MA (1986).

A P P E N D I X: THE IRUBESS PROPOSAL

Incremental REFER-UUCP Bibliography on Expert System Shells (IRUBESS-0)

Pmail: H. Boley, FB Inform., Univ. 675 Kaiserslautern, Box 3049, W. Germany
Email: lisplog@uklirb.uucp

General-purpose tools made for knowledge representation & processing in expert systems (rather than special-purpose tools derived from expert systems for diagnosis, configuration etc.) are the 'shells' to be emphasized here.

Such expert system shells can be regarded as -- often hybrid -- AI languages with good interactive environments because the label 'expert systems' is popularly used for a large subset of AI applications.

While we are far away from 'an exact science of shells' (with subfields like 'shell semantics', 'shell correctness', and 'shell complexity'), the compilation of a bibliography on serious shell papers could help users, developers, and analysts of these -- often commercial -- tools. Because the IRUBESS data are provided in machine-readable form in a non-profit fashion, they may contribute to saving duplication of effort, which should also be attempted in other fields inside or outside AI.

Since REFER is now a de facto standard for notating bibliographies in ASCII (with conversion routines to other, possibly non-ASCII, notations being simple), it was selected for the shell references in IRUBESS.

Because of REFER's UUCP-wide use IRUBESS can also be made available and further expanded electronically via UUCP (email and newsgroups). We can profit from the REFER convention to sort a bibliography on demand only, just appending new entries at the end of the ASCII file that constitutes the bibliography (addbib irubess): IRUBESS can be UUCPed incrementally in well-defined parts from update to update, and recipients can simply copy these together and then have the new IRUBESS file sorted before pretty printing (sortbib irubess | roffbib | more).

If you haven't used the family of REFER programs before, e.g. for collecting the sub-bibliography of references quoted in an nroff/troff/xroff text, please consult the manual pages (man refer). You may try REFER with the entries at the end of this text. Should IRUBESS grow very large, indexing with INDXBIB may become necessary, but fortunately this would still support incremental growth. Is there a REFER better than our May 1986 version?

Follow these rules when emailing references to the above electronic address:

* Specify "Subject: IRUBESS update" for normal updates and -- in separate email -- "Subject: IRUBESS correction" for corrections of existing entries (if such should still become necessary), and give both your email address (return paths are often nasty and sometimes unusable) and your

"shortest unique physical/postal/paper address" (as the last resort) in the body of your email, followed by the correct references in REFER syntax.

* Send references about shell overviews, critiques, and user experiences, as well as references technically describing individual shells, etc., but do not send pseudo-references of commercial shell descriptions (no advertisement).

* Feel free to email references of those of your own papers that fit well into the category of shells compiled in IRUBESS, even if your shell is experimental only (has no external users); you should pmail hardcopies of any such emailed references to the above paper address.

* Since we will watch for quality we cannot be obliged to distribute (all) references emailed to us, or to leave references distributed in an update also in later versions of the full version of IRUBESS (such a "bibliography non-monotonicity" may be unavoidable in exploding fields, where it can be hard to distinguish short-term relevance from long-term relevance); one small aspect of quality, supported by IRUBESS, is awareness of related work.

* Make sure that you don't send references already in IRUBESS, if uncertain, by requesting the most recent update(s) or the full version via email (since parallel submissions cannot be avoided the final job of duplicate elimination must be done centrally in Kaiserslautern).

* Use exactly the capitalization and abbreviation conventions exemplified in the sample entries below (for authors don't use initials if you know their full first names).

* Supply complete references only, e.g. both volume and issue numbers of journal articles (the normal ADDBIB doesn't prompt for the issue number, but you can end the volume line with "\" and then type the prompt "%N" yourself).

* You may send references with optional fields like keywords and abstract, but these may be eliminated from the distributed version because of the problems of overall consistency and manageable size.

Other important rules of the game are the following:

* Even though every sender is urged to cross-check every single reference submitted to IRUBESS, we cannot accept any liability for errors in the IRUBESS data (in this entire text "IRUBESS data" includes all updates).

* It is forbidden to exploit the IRUBESS data commercially, as by selling them as a stand-alone bibliography to third parties; it is permissible to use a REFER-generated sub-bibliography of IRUBESS even within a technical paper that is distributed commercially.

* Only if there is no other way to obtain the IRUBESS data (e.g. from the friend who told you about them) will they be sent directly to a recipient via email from Kaiserslautern, the normal way being its incremental

submission to a newsgroup. (Perhaps an ftp option or an automatic mail replier for "Subject: IRUBESS request" may be installed later.)

* The IRUBESS management in Kaiserslautern may be ended at any time, perhaps after having found another site to continue the managing job.

The following REFER input example is a prefix of the file used for the references of this paper, the 'version zero' of IRUBESS:

```
%A William Mettrey
%T An Assessment of Tools for Building Large Knowledge-Based Systems
%J AI Magazine
%V 8
%N 4
%P 81-89
%I AAAI
%C Menlo Park
%D Winter 1987

%A Detlef Karras
%A Lutz Kredel
%A Uwe Pape
%T Entwicklungsumgebungen fuer Expertensysteme.
Vergleichende Darstellung ausgewaehlter Systeme
%I Walter de Gruyter
%C Berlin, New York
%D 1987

%A William B. Gevarter
%T The Nature and Evaluation of Commercial Expert System Building Tools
%J IEEE Computer
%P 24-41
%D May 1987

%A Mark H. Richer
%T An Evaluation of Expert System Development Tools
%J Expert Systems
%V 3
%N 3
%P 166-182
%D Jul 1986

%A R. Citrenbaum
%A J. Geissman
%A R. Schultz
%T Selecting a Shell
%J AI Expert
%V 2
%N 9
%P 30-39
%D Sep 1987
```