1989

Rapid Prototyping Languages and Expert Systems

Luqi

IEEE

Luqi. "Rapid Prototyping Languages and Expert Systems." Intelligent Systems, Summer 1989, pp. 2-5, vol. 4
https://hdl.handle.net/10945/65357

EXPERT Focus

Ed: Lance B. Eliot
Expert Systems Lab
University of Southern California

Rapid Prototyping Languages and Expert Systems

Luqi, The Naval Postgraduate School

DARPA/ISTO — the Defense Advanced Research Projects Agency, Information Science and Technology Office — is seeking to develop a new language for the rapid construction of software prototypes. The common prototyping language to be designed will be part of larger subsequent efforts to develop a comprehensive prototyping system providing additional tools that realize a high-productivity software design and prototyping environment.

While unrelated to the DARPA/ISTO project, Luqi addresses this topic below. Specifically, in this IEEE Expert *exclusive, she responds to four questions in the following order: (1) What are prototyping languages? (2) How do prototyping languages serve expert system applications and development? (3) How can expert system techniques influence prototyping languages? (4) How can a common prototyping language interface with Ada?*

Henry Ayling
Managing Editor

Work on rapid prototyping languages aims at reducing software development costs via prototyping. A software prototype is an executable initial version of a proposed system. Prototypes are built (among other reasons) to assess whether a proposed system will be acceptable to its users and whether a proposed design will provide adequate functionality and performance. A prototype is constructed prior to the system's production version to (1) gain information that guides analysis and design, and (2) support generation of the production version. To be useful, prototypes must be constructed quickly and economically. Therefore, a prototyping language must make prototypes easy to construct, modify, and monitor — possibly at the expense of efficiency, completeness, capacity, or robustness. Prototyping is especially useful for large systems or novel application areas. Expert systems often fall into the latter category.

Rapid prototyping languages should support a comprehensive set of tools for computer-aided software design and prototyping. The goals for such a language and the associated prototyping system are

(1) To rapidly construct and adapt software,

(2) To enable the development of more powerful systems,

(3) To validate that specified systems are acceptable to users,

(4) To check the internal consistency of proposed designs, and

(5) To ensure the correctness of transformations and verify that implementations fully conform to specifications.

We will discuss the principles of language support for rapid prototyping, based on our experience in designing a prototyping language and conducting feasibility studies for its implementation over the past five years. We will then examine some basic issues involved in the design of a rapid prototyping language, paying special attention to issues involving expert system prototyping.

Requirements

Developing a general-purpose prototyping language is an ambitious task, and requires solutions to some open research problems for complete fulfillment. Prototyping has potential benefits for large software systems, many of which are concurrent and distributed, exhibiting hard real-time constraints. Expert systems are being implemented with concurrent and distributed configurations, and a growing demand exists for expert systems able to meet such real-time constraints.

General properties. Let's examine the general properties of a comprehensive prototyping language. A prototyping language should have a clear and simple structure and semantics to make it easy to learn, understand, and process mechanically and rapidly. This implies uniform structure, a small number of orthogonal constructs, and general interpretations without special cases or restrictions. To support automated tools, the language should have an abstract syntax and an unambiguous and precisely defined meaning. The underlying model should have a mathematical basis to support execution, analysis, verification, and trusted transformations. In particular, the semantics of the language should support rigorous reasoning about the properties of prototypes described in the language and transformations of the language's expressions. The language should also support a user interface to communicate with untrained people, including graphical summary views, English paraphrasing, and explanation facilities.

A prototyping language should be expressive. The language should be easy to use when constructing concise and clear descriptions for many varied systems. This implies language support for abstractions, uniform communication, logical inference, incomplete descriptions, and automated design completion. In addition to providing traditional facilities for functional, data, and control abstraction, the language should also support abstractions for concurrency, synchronization, and timing constraints. The language should be at a specification and design level rather than at a programming level: language constructs should correspond directly to decisions made by designers, rather than to operations performed by processors. This will make prototype descriptions self documenting and easy to change. The language should allow designers to specify only selected attributes, which requires automatically supplying default values for all attributes needed for the execution of a software prototype. The language should be capable

0885-9000/89/0500-0002 $1.00 © 1989 IEEE

of constructing software tools in its own prototyping environment.

To support large-scale prototypes, system evolution, and parallel execution, a prototyping language should have mechanisms for (1) localizing design decisions in the description, and (2) localizing interactions between system components or pieces of knowledge in the knowledge base. These features allow independently designed subsystems of complex expert systems to cooperate without unexpected interference.

To support user validation and system evolution, a prototyping language should support a facility for maintaining correspondence between requirements and design decisions. Tools will be needed to locate the parts of a software prototype that are affected by a requirements change. The language should provide a harmonious interface for such tools.

Facilities in a prototyping language for recording black-box specifications can provide the benefits of a specification language. They support prototype component documentation, verification via proofs and automated testing, and queries for reusable component retrieval. They also form the basis for automated synthesis capabilities, inheritance of common properties and constraints, and consistency checking. For expressiveness, this part of the language may contain noncomputable constructs including quantifiers ranging over unbounded sets. The language should support facilities for describing clear-box characteristics of designs; for example, interconnections of available components, dependencies between components, design goals (including invariant constraints or bounding functions), and design justifications (including criteria for choosing between alternative designs).

The language should have a distinguished executable subset that is easily recognizable by human users and by automated tools. Every expression in this distinguished subset should be executable for all possible initial conditions, although some expressions may denote nonterminating computations. The distinguished subset need not contain all executable expressions, and expressions outside the distinguished subset may be partially executable in the sense that execution may fail under some conditions. It should be possible to either augment or transform expressions of the language outside the executable subset to make them executable.

Besides supporting queries for the retrieval of reusable software components, the language should have facilities for adapting components to new uses and making small perturbations on their behavior without examining details of the internal implementation of components.

To support high productivity, the language should support the construction of efficient implementations by augmenting the prototype description with annotations describing additional constraints or lower level design decisions. This enables designers to view optimization as a refinement step where additional information is added to original descriptions, rather than a complete reformulation of the system description. Such an approach saves designer time by avoiding repeated treatment of the same issues in different ways, and by reducing the opportunities for making transcription or translation errors.

While more of a concern for the system's production version than for the prototype, efficiency cannot be ignored because we must be able to run test cases and gather data in a reasonable amount of time. This implies that execution mechanisms based on exhaustive enumeration are insufficient to meet the requirements of a prototyping language, although they may be supplied as a default to allow running small test cases in the absence of information about more efficient execution strategies. Therefore, the language should provide a set of fairly efficient execution mechanisms, tools for locating performance bottlenecks in larger systems, and incremental optimization transformations to improve prototypes that are impractically slow.

Real-time constraints. Real-time constraints impose a slightly different set of subgoals: execution times must be predictable, although not necessarily fast. Prototypes of real-time systems may operate in simulated time or linearly scaled real time, but actual execution times for the production version must be predictable within accurate bounds. The presence of real-time constraints severely restricts the kinds of computation systems may perform and — in the case of expert systems — limits the amount of logical inference that can be performed. The design of expert systems that operate within real-time constraints has been largely unexplored; significant research progress is needed in this area to fully realize the goals of a comprehensive prototyping language.

The rapid construction of software prototypes depends on simplifying the system view through which specifiers and designers do their work, and providing automated means for bridging the gap between this simplified view and the detailed programming-level description currently needed to make a software system efficiently executable. This automated support should include mechanisms for execution, static analysis of proposed system properties, preparation of test cases, reporting and analyzing results, and diagnosing ill-formed descriptions and departures from desired behavior to enable specifiers and designers to work entirely within the simplified view — at least during construction of the initial prototype. While the construction of tools is not required until later phases of the project, supporting the construction of such a tool set is a major driving force for language design. Developing an integrated set of tools requires a consistent and simple semantic model rich enough to express and support all of these functions. Finding suitable models is the key to the project.

Modeling issues

Models underlying the language provide a common ground for the associated set of tools. The semantic model for the language provides the basis for automated analysis, while the computational model provides the basis for execution. One of the main challenges in this project is to find a model that can coherently span the range of applications required. This will require a significant advance in the state of the art.

There is no single common model of expert systems available for rapid prototyping. First-order logic is one of the most familiar models for reasoning, but has been criticized for weaknesses including the lack of facilities for handling uncertain information, representing heuristic methods for speeding up conclusions, and nonmonotonic reasoning. Many other kinds of logic have been proposed, but theories of these logical methods are still being explored and there has been no consensus on whether a single logic is suitable for constructing all types of expert systems, or which variety of logic is the most promising. Approaches to expert systems are based on models other than logic; for example, semantic networks, Bayesian statistics, and production systems. Since it is not clear which approach will yield the best results in the long run, a comprehensive prototyping language must find a unified way of treating most issues raised by this diverse set of models.

Moreover, no single commonly accepted model can represent real-time constraints. Some approaches that have been explored include temporal logic, state machines, mode charts, augmented dataflow diagrams, Petri nets, and I/O automata. The model for a comprehensive prototyping language should be chosen to enhance the application of recent results in logic, graph theory, and combinatorics to link the semantic model to an effective execution mechanism. Other unexplored areas include effective models for real-time databases and real-time communications networks. In both areas, the problems of providing service within guaranteed worst-case time bounds remain largely unexplored.

Expert system design

Several special-purpose systems for supporting expert system design have been developed, some of which are known as expert system shells. A comprehensive proto-

typing language should improve on available facilities if possible, and integrate them with facilities suitable for producing other kinds of software. Two approaches support the prototyping of expert systems: (1) adding special-purpose features to the prototyping language, and (2) adding predefined reusable software components that we can define within a general-purpose language; for example, specialized data types, state machines, and functions. The predefined-component approach is preferable to the addition of specialized language features because of the requirement for simplicity. However, such predefined components should have standardized interfaces to improve portability.

Standardization. Many standard building blocks for expert systems can be provided as generic predefined components. These include facts, rules, patterns, frames, contexts, constraints, demons, instance generators, pattern matchers, unification mechanisms, and forward and backward chaining inference engines. Standardization requires careful analysis of these components and specification of their required properties. An open issue is whether current mechanisms for defining generic components are flexible enough to adequately capture the range of behavior required for these kinds of components — and, if not, what extensions are required.

Expert system prototyping. Some requirements for a prototyping language are determined by the need for prototyping expert systems. Examples of such requirements are (1) a means for conveniently defining external representations and input facilities for the knowledge in the knowledge base, (2) support for the proper treatment of higher order objects (including types, functions, tasks, and generators), and (3) support for control mechanisms such as state-triggered demons, backtracking, runtime control over task priorities, and the scheduling of temporal events. It is important to meet these requirements in a prototyping language for expert systems.

Knowledge base issues

Knowledge base management is an important part of expert system design. Rapid prototyping of knowledge-based systems brings special requirements for the design of a prototyping language as well as its environment.

Expert system technology is useful in implementing parts of the supporting environment for a prototyping language. For example, such an environment needs knowledge base support for the following items:

(1) Managing reusable components: The environment should contain a large software base with reusable components. This software base should be coupled with a set of rules for tailoring and combining available components to fulfill queries that do not exactly match any of the components explicitly stored in the software base.

(2) High-level debugging: Errors and failures during prototype execution should be mapped from the programming-language level to the prototyping-language level, thereby enabling designers to work entirely in terms of the semantic model associated with the prototyping language.

(3) Optimization: The transformations for optimizing a system prototype to produce a production version should be performed with minimum designer interaction. This implies keeping track of decisions made by designers in optimizing previous versions, determining which of these decisions are still valid for later versions, and automatically applying those valid decisions.

(4) Explanations: Justifications for decisions made automatically should be available to provide feedback to designers when automated design completion procedures fail. This requires an expert system with a substantial knowledge base.

These needs indicate that the prototyping system associated with a comprehensive prototyping language will need expert system technology for realizing some of its major subsystems.

The Ada interface

DARPA/ISTO's proposed connection between the common prototyping language and Ada raises several issues that must be considered. Goals for the common prototyping language include computer-aided transformations of prototypes into Ada implementations of the software's production version, and eventually implementing the tools in the prototyping system in Ada to provide portability. This poses a problem because ordinary compiler technology is insufficient to execute the prototyping language. The need for flexibility and runtime handling of newly created types and procedures to support expert systems also provides challenges for efficient implementation techniques in terms of Ada. Conventional translation techniques must be joined with (1) facilities for scheduling to meet hard real-time constraints, (2) transformations that execute incompletely specified processes, and (3) access to an interpreter or incremental compiler at runtime.

Ada provides a completely static-type system, treats types and functions as second-class objects, and requires that task priorities be known at compilation time. Clearly, the flexibility required for supporting expert system development can be provided by adding a runtime interpreter on top of the Ada language. The difficulty will be to provide these features efficiently, and without introducing excessive runtime overhead for prototype portions that do not require flexibility beyond that provided directly by Ada.

Ada provides relatively weak guarantees about task scheduling, and limits programmer control over scheduling to statically specified priorities. Since this is somewhat removed from the support level needed for implementing hard real-time systems, the execution support system for the prototyping language will have to provide higher level facilities for scheduling real-time operations. Such facilities can be classified as on-line (done at runtime) and off-line (done prior to execution). There is no universally accepted approach to real-time scheduling.

Worth Knowing: Kudos and Recent IEEE Expert *appointments*

For the second consecutive year, *IEEE Expert* is a finalist for the Western Publications Association's Maggie Award as best computer science publication in its category. Also up for Maggies are *IEEE Computer Graphics and Applications* and *IEEE Software*.

Effective with this issue, Editor-in-Chief David Pessel has appointed Lance B. Eliot editor of our News and Focus Sections in recognition of his numerous contributions over the last three years. In addition, Dr. Eliot will continue as an *IEEE Expert* editorial board member.

Last February, EIC Pessel appointed Craig A. Anderson editor of *IEEE Expert*'s Products Section. Editor Anderson plans to increase product reviews and to implement product-related interviews with developers, vendors, and users. Anderson replaces A. Winsor Brown, who served as products editor for the magazine's first three years. An *IEEE Expert* Editorial Board member, Brown will continue with Products in an advisory capacity. Kittur S. (Doc) Shankar, also an Editorial Board member, continues as resources editor. For a complete Editorial Board listing, see page 7.

The *IEEE Expert* staff welcomes and applauds these recent appointments, plus the addition to our staff masthead (in February) of highly regarded commentators Ware Myers and Tom Schwartz as *IEEE Expert* contributing authors.

Optimal scheduling algorithms consume considerable time and generally cannot be carried out on-line, while off-line approaches are inflexible and do not handle overload situations well. Many different scheduling algorithms exist; choosing the best one for a given application is difficult.

Transformations are needed to execute incompletely specified components. Such transformations should supply reasonable default values for attributes necessary for execution if designers do not explicitly specify them. These attributes can be explicitly specified to produce a more accurate system model or to improve its performance. One example of such attributes is the assignment of tasks to physical processors. Sometimes the assignment of specific critical tasks to specific processors is necessary to meet tight timing constraints by avoiding the overhead of some interprocessor communication. However, designers usually don't care about the placement of all tasks, and would like systems to assign reasonable default locations to all tasks not having explicit processor assignments.

DARPA/ISTO has made an important decision to develop designs for a rapid prototyping language for software systems, which is intended to apply to various large software systems, including knowledge-based systems, parallel systems, distributed systems, and real-time systems. DARPA/ISTO's Common Prototyping Language Project has an ambitious set of goals raising interesting research problems, many of which involve expert systems. Solutions to these problems are essential for achieving significant improvements in the quality and productivity of the software development process.

Over the last few years, Luqi has worked on rapid software prototyping and has made considerable recent contributions to software engineering and AI literature in IEEE Expert *(Winter 1988, pp. 9-18),* Computer, IEEE Software, *and* IEEE Transactions on Software Engineering. *She received her BS in computational mathematics from Jilin University (PRC) and her MS and PhD in computer science from the University of Minnesota, where she taught and performed software R&D. Before joining the Naval Postgraduate School, she worked for International Software Systems, and did research for the Academy of Science in Beijing. An assistant professor of computer science since 1986, Luqi can be reached at the Computer Science Dept., Naval Postgraduate School, Monterey, CA 93943-5100.*