1991-06

The Role of Prototyping Languages in CASE

Luqi

World Scientific Publishing

Luqi, "The Role of Prototyping Languages in CASE", International Journal of Software Engineering and Knowledge Engineering, June 1991, Vol. 1, No. 2, pp.131-149.
https://hdl.handle.net/10945/65707

International Journal of Software Engineering and Knowledge Engineering
Vol. 1, No. 2 (1991) 131–149

THE ROLE OF PROTOTYPING LANGUAGES IN CASE*

LUQI
Computer Science Department, Naval Postgraduate School, Monterey, CA 93943, USA

Received 3 February 1991
Revised 26 February 1991
Accepted 26 February 1991

Prototyping languages form a new category in the computer language family. They are different from the commonly familiar computer languages because they are used to support a higher level of automation at early phases of software development as well as throughout the entire process. They are used to create mechanically processable and executable descriptions or models of proposed software systems. Prototyping languages are also used to firm up requirements via frequent modifications and demonstrations of the models in an iterative process of prototype evolution. The benefits of a prototyping language are fully realized when it is used with its computer-aided prototyping system (CAPS). In this paper, we describe the background, requirements, characteristics, computational features, and general principles for the design of prototyping languages. An example of a prototyping language design is used to illustrate these concepts.

1. Introduction

Computer Aided Prototyping Systems(CAPS) should be used to prototype large, parallel, distributed, real-time, and knowledge-based systems because the requirements for design of such software systems are difficult to assess, leading to demand for prototyping support in these areas [15, 24]. The formal prototyping languages needed to build CAPS form a special category in the computer language family.

Rapid prototyping languages are used to create software prototypes, which are mechanically processable and executable descriptions or simplified models of proposed software systems. They are also used to modify the models frequently in an iterative prototype evolution process for the purpose of firming up the requirements. Figure 1 illustrates the prototyping process, which consists of two stages: prototype construction and code generation. The prototyping stage firms up software requirements through iterative negotiations between customers and designers via examination of executable prototypes. The designer adjusts the requirements and modifies the prototype accordingly based on feed-back from the customer until the customer agrees on the requirements. The code generation stage

* This research was supported in part by the National Science Foundation under grant number CCR-9058453.

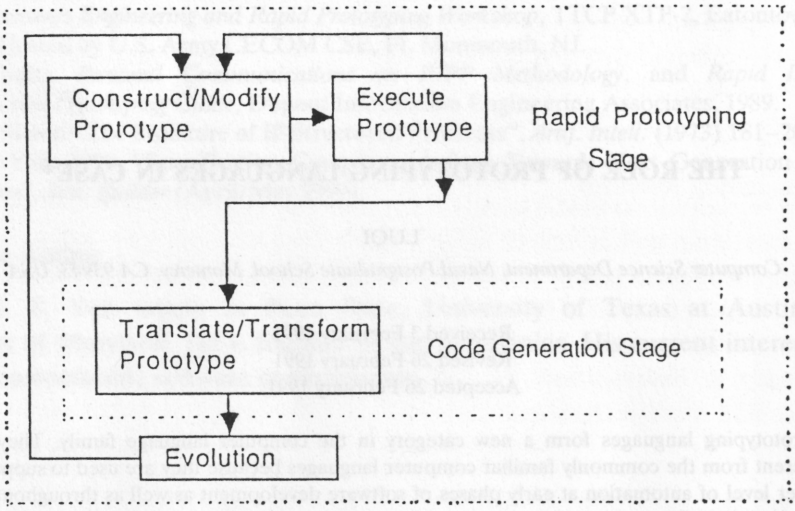


Fig. 1. Rapid prototyping process

focuses on augmenting the prototype to generate the production code. Prototypes are built to gain information to guide analysis and design, and support automatic generation of the production code.

A rapid prototyping language intended to apply to a variety of software systems is an ambitious goal that raises many interesting research problems [7]. Goals for such a language have to include the modeling of those systems, providing a fundamental mechanism for the integration of the tools in its CAPS, and computer-aided transformations of prototypes into implementations of the production version of the software. Solutions to these problems are essential for achieving significant improvements in the quality and productivity of the software development process [26].

It is useful to briefly examine the history of computer languages. The terminology for describing languages has been changing dramatically with implementation technology. Originally any compiled programming language was a very high level language. As systems became more complex, the meaning of the term shifted towards design languages which can describe system structure without introducing low level implementation details and can define generalized components that can be adapted to many different situations. Technologies improved to the point where programming languages could support abstraction and generalization, e.g. Ada and Smalltalk. Systems became even larger, and the meaning of the term shifted again, towards languages describing what a system is supposed to do, without specifying how the system is to accomplish its goals. As technology advances some of these specification languages are becoming executable. The term "very high level language" is not precisely defined, since the concept of a very high level language is a moving target that depends on the current state of compiler technology and the

speed, memory capacity, and cost of available hardware. Currently it refers to languages with abstractions and powerful mathematical constructs used in the early stages of the traditional software life cycle, such as domain modeling languages, specification languages, design languages, and prototyping languages.

As compiler and hardware technology improves, the gaps among these high level computer language categories are getting smaller and may eventually disappear. Programming languages are getting more expressive and more flexible, and are supporting more abstract descriptions of the processes to be carried out, while specification and design languages are incorporating larger executable subsets [4, 27]. In the near future these types of languages will remain distinct to more effectively support different classes of tools. Programming languages will support optimizing compilers whose main objective is to produce efficient implementations. Specification and design languages will support tools for requirements analysis and proving the correctness of designs and implementations. Prototyping languages will support tools for prototype construction, demonstration and production code generation.

The theory and technology of computer languages has become sufficient to support exploratory development of a general purpose rapid prototyping language. In 1985, a special purpose rapid prototyping language PSDL (Prototype System Description Language) and its CAPS were published in [30] for rapid prototyping in the development of large and real-time systems [15]. In the past years, the feasibility study for the tools in the CAPS has shown great promise [1]. The design and implementation of a rapid prototyping language can benefit from past work on specification, design, and programming languages. Many such languages previously designed for programming, design or specification were used or intended to be used for prototyping purposes in different degrees. In the 1970's, the SETL programming language was designed based on set theory and already had the primitive concepts for integrating high level mathematics with an execution capability [23]. Other languages that have been used for prototyping at the programming level include APL, SNOBOL, LISP, and PROLOG. The Gypsy language from UT Austin provides a simple basis for representing, executing, and proving correctness of communicating processes [8]. Research on real-time modeling and scheduling provides fundamental support for the design of hard real-time systems [17]. Work on attribute grammars paved the road to automated approaches for prototyping special purpose languages [9, 22]. The GIST system at ISI has explored computer-aided requirements modeling with the aid of symbolic evaluation and English paraphrasing [6, 25]. The OBSERV system has used abstract ports coupled with graphical interconnection facilities to support rapid prototyping and exploratory design [29]. The Argus project at MIT has explored implementation of atomic transactions in distributed systems [13]. The wide spectrum language Refine from Reasoning systems [21], the DRACO system from UC Irvine [19], and the CIP project from Munich [2] have explored the feasibility of using transformations to realize specifications.

The strength of a specification language is its simplicity, abstraction, clarity of expression, and means for rigorous logical reasoning [3]. The strengths of a design language are its expressiveness and support for recording goals and justifications. A common weakness of specification and design languages is lack of efficient facilities for execution or lack of any effective means for execution. The strength of most programming languages is supporting efficient execution, while common weaknesses are the need for specifying many details and lack of facilities for recording goals and justifications in a formal way. A prototyping language should integrate the strengths and functions of specification and design languages with the capability for execution. Because of the wide range of goals for prototyping languages, design decisions for those languages are difficult to make. It is helpful to examine the functions of a prototyping language in the rapid prototyping process before further discussion of language principles.

2. CAPS and Prototyping Languages

In the rapid prototyping process, software prototypes must be constructed quickly and at low cost to be practical. The prototyping language and its CAPS are used for

- Rapidly building and modifying prototypes,
- Simplifying the design of complex systems,
- Demonstrating prototype systems to users,
- Evaluating and checking proposed design, and
- Efficiently and reliably transforming prototypes into production code.

A prototyping language has no obligation to give detailed algorithms for all components of the system as long as it is descriptive and executable. Its CAPS should automatically supply algorithmic details needed for execution, help the designer manipulate, analyze, demonstrate, and explain the prototype, and help the development team transform the prototype into a production version of the system. The design of the prototyping language is subject to the need of supporting the software tools in a CAPS system.

2.1. *Tools in a computer aided prototyping system*

Prototyping with computer-aided tools makes a rapid process possible. A CAPS designed for supporting a special prototyping language should make it easy to specify, construct, demonstrate, understand, explain, and modify a software prototype. The language provides the basic communication and representation medium for all the tools in a CAPS system. In the CAPS proposed in [15], guidelines for decomposing software modules, reusable components, a prototyping interface, a design database, and a software base management system with inference capabilities make the process possible.

The tool set should provide facilities for analyzing the consistency of a prototype design. Some of the properties that should be checked include:

- Type consistency,
- Feasibility of timing constraints,
- Consistency between the levels of a hierarchical description,
- Preconditions on input parameters and generic parameters,
- Constraints on relative rates of producer and consumer processes,
- Absence of deadlocks in distributed and parallel systems,
- Absence of unhandled exceptions.

In addition to providing facilities for constructing and checking the internal consistency of a prototype, the tool set should provide facilities for generating input data, debugging, displaying output, and evaluating the results of prototype execution in terms of the same semantic model used for the design of the prototype.

To support user validation and system evolution, a prototyping language should interface to a facility for maintaining the correspondence between requirements and design decisions. Tools are needed for determining which parts of a prototype are affected by a requirements change, and which requirements are affected by a proposed change to the behavior of a prototype.

The tool set should also provide a design database for maintaining the design history, to capture dependencies between different versions of the system and to record alternative designs that were considered [5, 10, 16, 20]. This database should be capable of recording and maintaining constraints between the components of a prototype. An issue in the design of such a database is the relation between the prototyping language used for representing the design objects in the database, and representations for the attributes, relationships, and constraints used by the CAPS tools. The information in the designer's view of the language is a subset of the information in the tool views, since tools often add additional attributes to the entities defined by the language for recording the results of analysis and synthesis procedures. Since the tool set is likely to grow, the representation of design objects should be extendible without affecting the interfaces to existing tools.

2.2. *The purpose of a prototyping language*

A prototyping language is used by both people and software tools. To support the human users, a prototyping language should be easy to write, understand, and modify. To support the tools, the language should be easy to analyze and transform mechanically. The requirements for a prototyping language follow.

A prototyping language should be clear and simple. 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 a simple abstract syntax and a precisely defined meaning. The underlying model should have a mathematical basis to support execution, rigorous reasoning, and transformations. The model should also support tools for communication with untrained people. Such tools should provide graphical summary views, English paraphrasing, and explanation facilities.

A prototyping language should be powerful and concise. It should enable brief prototype definitions for a wide variety of software systems, including the tools in the CAPS. The language and the tools should support abstractions, incomplete descriptions, and automated design completion. The language should support abstractions for concurrency, synchronization, and timing constraints in addition to traditional functional, data, and control abstractions. The constructs of the language should match decisions made by the designer more closely than the operations performed by the processor, to make prototype descriptions easy to understand and change. The designer should be able to specify only the essential attributes of a proposed system, and the tools should supply default values for all attributes needed for execution of a software prototype. To avoid complicating the language, high level abstractions should be expressed as standard pre-defined components in a knowledge base whenever it is possible to do so without extending the language.

A prototyping language should localize design decisions and interactions between system components or pieces of knowledge in the knowledge base. These features allow independently designed subsystems of complex systems to cooperate without unexpected interference, simplify concurrency issues, and aid in scheduling.

A prototyping language should be able to represent black-box specifications. Specifications are needed for documenting prototypes, retrieving reusable software components, and verifying implementations via automated testing and proofs. Such descriptions also form the basis for automated synthesis capabilities, inheritance of common properties and constraints, and consistency checking. This part of the language may contain non-computable constructs such as unbounded logical quantifiers to gain expressive power.

The language should be able to represent clear-box design information. This information includes interconnections of subcomponents, dependencies between components, design goals such as invariant constraints, and criteria for choosing between alternative designs. The language should have facilities for adapting components to new uses and making small perturbations on their behavior without examining the details of the internal implementation of the components, to make it easier to reuse components.

The language should have an expressive executable subset. This subset should be easily recognizable both by people and automated tools. It should be possible to transform or augment expressions of the language outside the executable subset to make them recognizably executable.

The language should support the construction of efficient implementations by augmenting the prototype description with annotations. These annotations should describe additional constraints or lower level design decisions. This enables developers to treat optimization as a refinement step where additional information is added to the original descriptions, rather than a complete redefinition of the system. This avoids repeated description of the same information in different ways, and reduces opportunities for making errors.

It must be possible to run test cases and gather data in a practical amount of time. Efficiency does not have the highest priority in a prototyping language, but it cannot be ignored completely. Thus execution mechanisms based on exhaustive enumeration are insufficient for a prototyping language, although they may be supplied as a default to enable execution of small test cases even in the absence of information about more efficient execution strategies. The language should therefore support relatively efficient execution mechanisms, tools for locating performance bottlenecks, and incremental optimization transformations to improve prototypes that are impractically slow.

Real-time constraints require execution times to be predictable, although not necessarily very fast. Prototypes of real-time systems may operate in simulated time or linearly scaled real time, but the actual execution times for the production version must be predictable within accurate bounds.

3. Example of a Prototyping Language

The rapid prototyping language PSDL [14, 15, 30] and its CAPS were designed to support prototyping of large, parallel, and real-time systems. The CAPS provides a designer interface for constructing, analyzing, and modifying a prototype, along with execution facilities which realize timing constraints with respect to either actual or linearly scaled real-time.

PSDL encourages localized descriptions and software structures, to aid the designer in constructing and modifying understandable models of complex systems. The language has a simple and expressive computational model based on modified dataflow augmented with non-procedural constraints. The elements of the model are operators and data streams. Every operator is a state machine and some operators are functions, i.e. machines with an empty set of state variables and a single internal state. Every data stream carries values of an abstract type, and some streams carry exception values.

The computational model was developed based on two main concerns: ease of use by human designers, and ease of processing by the tools in the CAPS. Data flow diagrams have been widely used in requirements analysis, partially because they are easy to understand both by developers and by customers. This motivated us to adopt a graphical notation in PSDL, based on operators and data flows. However, the data flow diagrams commonly used in informal, manual approaches to requirements analysis are not sufficiently precise to support execution and analysis of real-time behavior of a proposed system. The behavior of a classical data flow diagram is not completely determined because the diagram shows only the data that can flow through the system, and does not specify the cause and effect relationships and the timing properties that determine the details of system behavior. We added annotations to the graphics to express this information, and found that this was sufficient to enable automatic execution and real-time scheduling by the CAPS tools.

A PSDL prototype consists of a hierarchically structured set of definitions for operators and types. Each definition has a *specification part* for black-box descriptions and an *implementation part* for clear-box descriptions. The specifications are used both for documentation of the prototype and for retrieval of reusable software components. A specification is executable without further information from the designer if the CAPS can automatically retrieve, adapt, and combine the reusable software components in its software base to match the specification. In cases where this is not possible, the designer must develop an implementation part decomposing the specified system into more primitive subsystems and provide black-box specifications for each of the subsystems. The decomposition is done in terms of the PSDL computational model, using augmented data flow diagrams.

An augmented data flow diagram is a directed graph, annotated with control constraints and timing constraints. The nodes in an augmented data flow diagram represent operators, the edges represent data streams, and the numbers associated with the nodes represent maximum execution times. The diagram has a graphical representation and the constraints have a text representation, as illustrated in Fig. 2. The example shows a simple control system illustrating some typical features of embedded software. The filter performs a smoothing operation to reduce the noise in the sensor data, and the controller uses the filtered sensor data to determine how to respond to commands from a human operator, which are transmitted to the embedded system via a switch on the operator's control panel. A simple smoothing filter might take the form

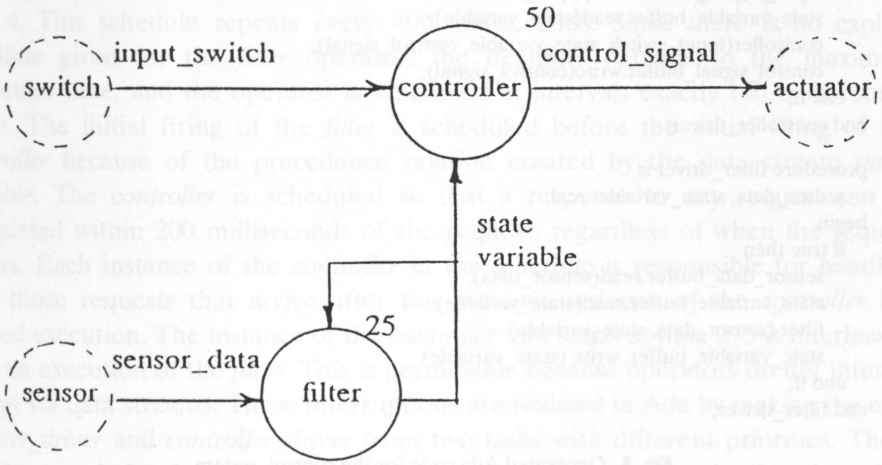
$$\text{new state\_variable} = w * \text{old state\_variable} + (1.0 - w) * \text{sensor\_data}$$

where the weight w is a real number between 0 and 1 that must be chosen to provide the best trade-off between response time and sensitivity to noise. The example has a minimal specification part with an informal description. The implementation part contains a graph showing the decomposition of the system into two subsystems, and the control constraints give control and timing information. The control constraints and the timing constraints determine both the conditions under which the operators are triggered and the buffering disciplines for the data streams. In general, control constraints can express conditional execution or output, and can control exceptions and timers. PSDL timers are used to control durations of states, and can be thought of as software stopwatches.

Timing constraints can be added to operators to define hard real-time deadlines for time-critical operators. Timing constraints can express sporadic data-driven execution as well as periodic execution. In Fig. 2 both operators are time-critical. The *filter* operator must be fired periodically, every 100 milliseconds. The *controller* operator is fired sporadically, whenever a new value for the *input_switch* arrives, and must complete execution within 200 milliseconds of the arrival of the new value.

```
OPERATOR control_system
SPECIFICATION
 INPUT operator_switch: boolean, sensor_data: real
 OUTPUT control_signal: real
 STATES state_variable: real INITIALLY 0.0
 DESCRIPTION { top level of a simple embedded system }
END

IMPLEMENTATION
 GRAPH
```




```
 CONTROL CONSTRAINTS
  OPERATOR filter PERIOD 100 ms
  OPERATOR controller TRIGGERED BY ALL input_switch
   MAXIMUM RESPONSE TIME 200 ms
   MINIMUM CALLING PERIOD 200 ms
END
```

Fig. 2. Example of an augmented data flow diagram in PSDL

There are two possible buffering disciplines for a data stream: *dataflow* and *sampled*. Dataflow streams act as first-in-first-out buffers, and are used for synchronizing data-driven computations. Sampled streams act as continuously available sources of data which can be read or updated on demand, and are used for connecting unsynchronized operators which can fire at different or unpredictable rates. Data streams have dataflow buffers if and only if they appear in a TRIGGERED BY ALL control constraint. In Fig. 2, the streams *input_switch* and *control_signal* are dataflow streams, while *sensor_data* and *state_variable* are sampled streams. The triggering conditions express requirements for the *controller* and the *actuator* to respond exactly once to every new value in the streams *input_switch* and

(a) generated buffer declarations and initialization

```
input_switch_buffer is new fifo_buffer(boolean);
sensor_data_buffer is new sampled_buffer(real);
state_variable_buffer is new state_variable(real, 0.0);
```

(b) generated driver code

```
procedure controller_driver is
  input_switch: boolean;
  state_variable, control_signal: real;
begin
  if input_switch_buffer.new_data then
    input_switch_buffer.read(input_switch);
    state_variable_buffer.read(state_variable);
    controller(input_switch, state_variable, control_signal);
    control_signal_buffer.write(control_signal);
  end if;
end controller_driver;

procedure filter_driver is
  sensor_data, state_variable: real;
begin
  if true then
    sensor_data_buffer.read(sensor_data);
    state_variable_buffer.read(state_variable);
    filter (sensor_data, state_variable)
    state_variable_buffer, write (state_variable)
  end if;
end filter_driver;
```

Fig. 3. Generated Ada code for the control_system

control_signal. The other streams must be sampled because the *filter* operator must operate at fixed times and values may be written into the *sensor_data* stream or read from the *state_variable* stream at unpredictable times.

The PSDL translator [1] generates Ada code for interconnecting operators appearing in an augmented dataflow diagram. Figure 3 shows a simplified version of the Ada code that would be generated for the example in Fig. 2 if the operators *filter* and *controller* are realized by reusable components from the software base. In the example we assume these components are Ada procedures with the same names as the corresponding PSDL operators. There are two different kinds of code generated: buffer declarations and driver code. Buffers are instances of pre-defined generic packages corresponding to the two buffering disciplines for PSDL data streams, with variants for declaring the initial values of streams that represent state variables for state machines. The driver procedures *controller_driver* and *filter_driver* realize the control constraints and perform the data stream operations. These procedures are called from the static schedule tasks at times determined by a pre-computed static schedule.

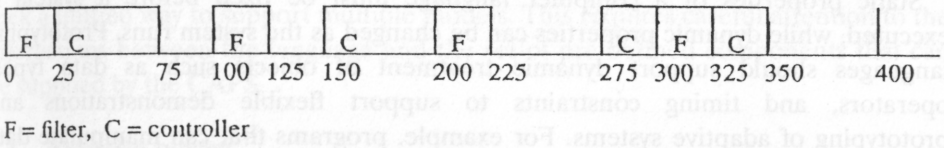


Fig. 4. Static schedule for the control_system

The timing constraints associated with time-critical operators are realized by the static scheduler in CAPS. A static schedule for the operators in Fig. 2 is shown in Fig. 4. This schedule repeats every 400 milliseconds. Since there is no explicit deadline given for the *filter* operator, the deadline defaults to the maximum execution time, and the operator is scheduled at intervals exactly 100 milliseconds apart. The initial firing of the *filter* is scheduled before the initial firing of the *controller* because of the precedence relation created by the data stream *state_variable*. The *controller* is scheduled so that a response to any request can be completed within 200 milliseconds of the request, regardless of when the request occurs. Each instance of the *controller* in the schedule is responsible for handling only those requests that arrive after the previous instance of the *controller* has started execution. The instance of the *controller* that starts at time 275 is interleaved with an execution of the *filter*. This is permissible because operators do not interact except via data streams. These interruptions are realized in Ada by making the calls to *filter_driver* and *controller_driver* from two tasks with different priorities. There are short periods of time at the beginning and at the end of each execution of an operator when such interruptions cannot be scheduled, because of atomic read and write operations on the data streams.

PSDL was designed so that operators can interact only via data streams. This locality property simplifies the design and modification of prototypes by ensuring that operators can be executed in parallel without interference and can interact only via the documented interfaces, and also makes it easier to construct schedules. The locality property is realized by the absence of a mechanism for transmitting objects with internal states along data streams and scoping rules that do not allow direct non-local data references.

4. Designing a Prototyping Language

A prototyping language should simplify the designer's view of the system and support automated means for bridging the gap between this simplified view and the detailed algorithmic descriptions currently needed for efficient execution. The CAPS system should provide mechanisms for execution, static analysis, preparation of test cases, display and analysis of results, and debugging to allow the prototype designers to work entirely within the simplified view.

4.1. *Static and dynamic properties*

Static properties of a computer language must be fixed before a system is executed, while dynamic properties can be changed as the system runs. Prototyping languages should support dynamic treatment of objects such as data types, operators, and timing constraints to support flexible demonstrations and prototyping of adaptive systems. For example, programs that can manipulate data types, programs, and schedules at run-time can adapt to unanticipated circumstances more readily than those that cannot.

This goal is hard to meet, because static declarations allow tools to provide more information about a proposed system and enable more efficient execution techniques. A prototyping language with many dynamic features requires type checking, interpreter calls or compilation, loading, linking, and scheduling to be done as the system runs. These facilities are difficult to implement efficiently, and are not supported by the class of languages usually used for production versions. Uniform guarantees of type correctness, clean termination, or meeting hard real-time constraints may not be possible without static restrictions on these properties. Thus a prototyping language should be able to represent optional static restrictions, and CAPS should support transformations adding explicit static restrictions to improve efficiency or predictability.

4.2. *Computational model*

The models underlying a prototyping language provide the 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 developing a prototyping language is finding models that can coherently span the range of applications required.

There is no common model of expert systems available for rapid prototyping. First order logic is one of the most familiar models for reasoning, but it has been criticized for its weaknesses, such as difficulties in handling uncertain information, representing heuristic methods for speeding up conclusions, and performing non-monotonic reasoning. Many other kinds of logic have been proposed, but there has been no consensus on whether there is a single logic suitable for constructing all types of expert systems, or which variety of logic is the most promising. Some approaches to expert systems use models other than logic, such as semantic networks, Bayesian statistics, and production systems. It is not clear which approach will yield the best results in the long run.

There is also no commonly accepted model for representing real-time constraints. Some approaches that have been explored include temporal logic, state machines, mode charts, augmented data flow diagrams, Petri nets, and I/O automata. The model for a prototyping language should be chosen to enhance the application of recent results in logic, graph theory, and combinatorics to provide an effective execution mechanism. Other unexplored areas include effective models for real-

time databases and real-time communications networks. Since different models appear to be best for different purposes, practical prototyping languages should seek a unified way to support multiple models. This requires careful attention to the interactions between the language and the set of pre-defined components that can be supplied by the CAPS.

4.3. *Execution support*

A knowledge-based approach is needed to provide adequate execution support for a prototyping language without requiring excessive algorithmic detail. CAPS should provide knowledge base support for the following functions:

Design—The CAPS system should contain models of common design activities and common classes of design decisions, to allow prototypes to be expressed in the conceptual framework of the designer rather than that of the machine. If the system is aware of the choices faced by a designer at each point in the design, it can present compact representations of the choices using menus. Such alternatives should have corresponding representations in the prototyping language.

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. This allows the system to find algorithms and data structures without imposing all of the details of the designer.

High level debugging—Errors and failures during prototype execution should be mapped from the programming language level to the level of the prototyping language, to keep programming details from intruding when the designer tests and demonstrates the prototype.

Optimization—The transformations for optimizing a prototype version of a system to produce a production version should be performed with minimum interaction with the designer. CAPS should keep track of optimization decisions made for previous versions of the system, determine which of those decisions are valid for later versions, and automatically apply the ones that are still valid. While it is not currently feasible to produce highly optimized implementations without human help, it is possible to automate routine decisions and rely on the designer for only the difficult decisions.

Explanations—Justifications for decisions made by CAPS should be available to provide feedback to the designer in cases where automated design completion procedures fail. Such a facility is needed to support systematic computer-aided design in situations where complete automation is not possible. This requires an expert system with a substantial knowledge base.

It is natural to consider the execution aspect of a prototyping language in terms of compiler technology. Unfortunately, ordinary compiler technology is insufficient for execution of a rapid prototyping language. The reasons are:

Flexibility—The need for flexibility and run-time handling of newly created types and procedures to support expert systems provides challenges for efficient implementation techniques.

Real-time constraints—Conventional translation techniques must be coupled with facilities for scheduling to meet hard real-time constraints, transformations to allow the execution of incompletely specified processes, and access to an interpreter or an incremental compiler at run-time.

Missing details—An issue that must be faced by an execution support system for a prototyping language is providing missing details. One of the goals of a rapid prototyping system is to execute prototype descriptions that do not contain details of algorithms and data structures [28]. This has to be handled by combining program transformations and specialized schedulers with a knowledge base containing programming and problem domain knowledge.

Transformations are needed to execute incompletely specified components. Such transformations should supply reasonable default values for attributes necessary for execution if the designer does not explicitly specify them. Different choices for these attributes can be explicitly specified to produce a more accurate model of the system or to improve its performance. In particular, default algorithms for unspecified or partially specified components should be supplied. Key problems are finding systematic ways of developing such transformations and determining reasonable default values based on models of the application domain. It is essential to automatically generate stubs for components that are unavailable in the software base and have not yet been designed to allow testing and demonstrating partially completed systems. Such stubs can be created by simple or increasingly sophisticated techniques, such as asking the user to supply values, using random selections from a fixed set of responses, using logic programming to simulate black-box specifications, or using transformation techniques to generate efficient implementations from the black-box descriptions. Other examples include assignment of tasks to physical processors and choosing display formats for outputs and error messages.

5. Semantics of a Prototyping Language

The key to computer-aided prototyping is finding simple formal models that can express the range of expected applications and effectively support automated processing. The desire for extensive automated processing in a CAPS system argues for Spartan simplicity in the design of the prototyping language: new language features should be resisted whenever it is possible to cover all required functionality without introducing new primitives. This section examines some likely application areas for rapid prototyping and some of the implications for the design of a common prototyping language.

5.1. *Supporting real-time systems*

The language and the CAPS knowledge base should support representations for timing constraints and overload resolution policies. Scheduling is a difficult issue for real-time systems. High level representations of timing constraints and overload resolution policies are essential to allow the prototype to express the necessary constraints on the scheduling of different tasks at a level matching the problem rather than at the level of the underlying run-time support system. Timing constraints on communications primitives are needed to handle distributed real-time systems.

5.2. *Modeling parallel systems*

High-level mechanisms for coordinating independent activities [11] and primitives for defining independent activities that are guaranteed not to interfere with each other are needed to simplify and speed up the construction of parallel systems. Localized modules with limited data access are essential for this purpose. Message passing, dataflow, and object-oriented ideas have been proposed to address this problem. Another consideration is avoidance of deadlock. It is useful to have a syntactically recognizable subset of the language that is capable of describing concurrent computations and carries a uniform guarantee of freedom from deadlock. Such a guarantee is possible if a suitable computational model is chosen. This kind of restricted subset is sufficient for many applications, and it can be augmented with facilities for adding additional constraints on global orderings of events, which are known to be potentially unsafe and are designed together with tools for checking safety of particular designs. For example, atomic transactions can simplify the design of distributed systems, although they introduce the potential for deadlocks.

5.3. *Designing distributed systems*

A prototyping language should provide a high level means for describing

1. constraints on communication time,
2. the granularity of atomic transactions,
3. standard protocols for achieving reliability despite processing and communications failures, and
4. constraints on the assignment of software tasks to physical processors.

This information should be optional, and the default should be the safest option rather than the most efficient one.

5.4. *Prototyping knowledge-based systems*

CAPS can provide generic pre-defined software components to realize many of the common building blocks for knowledge-based systems. These include facts,

rules, patterns, frames, contexts, constraints, demons, instance generators, pattern matchers, unification mechanisms, constraint propagation mechanisms, and inference engines. Thus the prototyping language does not appear to require special features beyond the flexibility described in Sect. 4.1 to provide these facilities. However, standardization of these building blocks requires careful analysis 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. A solution to this general problem that has been applied successfully in other contexts such as parsing context free languages is meta-programming: a family of reusable components is defined implicitly via a special purpose problem definition language and a program generator that tailors a known general solution strategy to the particular problem represented by a statement in the problem definition language. The program generator creates instances of the component family based on statements in the problem definition language. The components created by the program generator can be represented in the prototyping language or in a conventional programming language. In the context of parsing, an example of a problem definition language is a grammar notation such as BNF, which defines the source language to be parsed, and an example of a corresponding program generator is an LALR parser generator such as the UNIX yacc tool, which creates a parsing program corresponding to the given grammar. Other examples include tools for generating user interface software [12, 18]. Thus in addition to a prototyping language, a CAPS system may need a meta-programming language for specifying computed families of reusable components in the software base. A meta-programming language and families of reusable software components could also provide a means for conveniently defining problem-specific external representations and input facilities for the knowledge in the knowledge base.

To support rapid construction of expert systems a prototyping language should provide:

1. unrestricted higher order objects such as types, functions, tasks, and generators, and
2. control mechanisms such as state-triggered demons, backtracking, run-time control over task priorities, and temporal events.

Several of these features are needed to support flexible prototypes for other kinds of systems as well.

The presence of real-time constraints severely restricts the kinds of computations a system 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 is a largely unexplored area, and significant research progress is needed in this area to fully realize the goals of a rapid prototyping language.

6. Conclusions

Prototyping languages are designed based on knowledge and experience from all levels of the computer language hierarchy since they address functions from all of the levels. Studying the relevant aspects of specification, design, and programming languages is helpful in the design of prototyping languages.

The purpose of a prototyping language is to define an executable model of a system. The language is used to create specifications, express designs, and execute prototypes. Prototyping languages are used in requirements analysis for the purpose of requirements validation via early demonstrations to the customer. They are also useful for evaluating competing design alternatives, validating system structures, and exploring feasibility. In contrast, specification languages are used for defining external interfaces in the functional specification stage and for defining internal interfaces during architectural design at the highest levels of abstraction. They are also used for verifying the correctness and completeness of a design or implementation. Design languages are used for recording conventions and interconnections during architectural design and module design.

The difference between specification and design languages is the difference between interface and mechanism: a specification says what is to be done, and a design says how to do it. The main evaluation criterion for both specification and design languages is the ability to express simple, concise, and humanly understandable descriptions of complex behavior. It is useful for specification and design languages to be executable, but simplicity and ease of expression takes precedence when the considerations conflict. Computer aid is desirable for determining the properties of a specification and certifying that a design realizes a specification. An execution capability can contribute to these goals, but other types of mechanical analysis may turn out to be more useful for this purpose.

The difference between a design and a program is the difference between a plan and a finished product: a design records the early decisions that determine an implementation strategy, while a program contains all the details necessary to get an efficiently executable system. The primary goal of a design is documentation rather than execution, while the primary goal of a program is usually efficient execution.

A prototyping language aims at validating specifications and designs mainly via an execution facility. Prototyping languages can benefit from mechanisms developed to increase the expressiveness of specification and design languages, but must accept some restrictions to support execution. The execution mechanisms of a prototyping language should draw on programming language technology, but must accept some inefficiencies to support flexibility and ease of expression. The constructs of a prototyping language should be chosen to allow generation of efficient implementations by smoothly adding additional information and constraints.

Since completely automatic and totally correct implementation of powerful specification languages is an algorithmically unsolvable problem, research on rapid prototyping should explore human interactions for effectively guiding computer-aided implementation tools. A promising approach is augmenting abstract

specifications with annotations or pragmas giving advice about implementation strategies. An important problem is finding concepts and notations that can naturally express such advice in an abstract and orthogonal way. It is desirable to keep the abstract specification separate or easily mechanically separable from the annotations to provide simplified views of large system models.

Providing execution capability for high level prototype descriptions requires a knowledge based approach. The required knowledge base grows with the problem domain the language addresses. A substantial part of the knowledge in the knowledge base consists of reusable software components augmented with descriptions of their properties. Other kinds of relevant knowledge include methods for adapting and combining the components in the software base, properties of application domains, and the CAPS tools. Progress on rapid prototyping languages depends on solutions to open research problems in areas including semantic modeling, real-time scheduling, program transformations, version control in prototype databases, and retrieval of reusable software components.

References

1. C. Altizer, "Implementation of a Language Translator for a Computer Aided Prototyping System", M.S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1988.
2. F. Bauer, B. Moller, H. Partsch and P. Pepper, "Formal Program Construction by Transformations—Computer-Aided, Intuition-Guided Programming", *IEEE Trans. Softw. Eng.* **15**, 2 (February 1989) 165–180.
3. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1991.
4. A. Berztiss, "The Specification and Prototyping Language SF", Report 78, Systems Development and Artificial Intelligence Laboratory, Department of Computer and Systems Sciences, Stockholm University, 1990.
5. E. Borison, "Program Changes and the Cost of Selective Recompilation", Technical Report CMU-CS-89-205, Computer Science Department, CMU, Pittsburgh, PA, July 1989.
6. D. Cohen, "Symbolic Execution of the Gist Specification Language", in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 1983, pp. 17–20.
7. R. Gabriel, ed., *Draft Report on Requirements for a Common Prototyping System*, DARPA Information Science and Technology Office, Arlington, VA, Nov. 1988.
8. D. Good, R. Cohen and J. Keeton-Williams, "Principles of Proving Concurrent Programs in Gypsy", in *Proceedings of the 6th Annual Symposium on Principles of Programming Languages*, ACM, 1979, pp. 42–52.
9. R. Herndon and V. Berzins, "The Realizable Benefits of a Language Prototyping Language", *IEEE Trans. Softw. Eng.* **SE–14**, 6 (June 1988) 803–809.
10. M. Ketabchi, V. Berzins and S. March, "ODM: An Object Oriented Data Model for Design Databases", in *Proc. ACM Computer Science Conference*, February 1986, pp. 261–269.
11.. B. Kraemer, "SEGRAS—a Formal Language Combining Petri Nets and Abstract Data Types for Specifying Distributed Systems", in *Proceedings of 9th International Conference on Software Engineering*, March 1987, pp. 116–125.
12. T. Lewis, F. Handlosser, S. Bose and S. Yang, "Prototypes from Standard User Interface Management Systems", *IEEE Computer* **22**, 5 (May 1989) 51–60.

13. B. Liskov, "Distributed Programming in Argus", *Communications of the ACM* **31**, 3 (March 1988) 300–312.
14. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. Softw. Eng.*, (October, 1988) 1409–1423.
15. Luqi and M. Ketabchi, "A Computer Aided Prototyping System", *IEEE Softw.* **5**, 2 (March 1988) 66–72.
16. Luqi, "A Graph Model for Software Evolution", *IEEE Trans. Softw. Eng.* **16**, 8 (August 1990) 917–927.
17. A. Mok, "A Systematic Approach to the Design of Hard Real-Time Systems", in *Proceedings of the ONR Workshop on the Foundations of Real-Time Computing*, Office of Naval Research, Falls Church, VA, Nov. 1988, pp. 47–51.
18. B. Myers, "User Interface Tools", in *Milestones in Software Evolution*, IEEE Computer Society, 1990, pp. 261–270.
19. J. Neighbors, *Draco: a Method for Engineering Reusable Software Systems*, ACM Press, 1989.
20. J. Nestor, "Toward a Persistent Object Base", in *Advanced Programming Environments*, Vol. 244, Springer-Verlag, 1986, pp. 372–394.
21. *Refine User's Guide*, Reasoning Systems Inc., Palo Alto, CA, 1988.
22. T. Reps and A. Demers, "Sublinear-Space Evaluation Algorithms for Attribute Grammars", *Trans. Prog. Lang and Syst.* **9**, 4 (July 1987), pp. 408–440.
23. E. Schonberg, J. Schwartz and M. Sharir, "An Automatic Technique for the Selection of Data Representations in SETL Programs", *Trans. Prog. Lang and Syst.* **3**, 2 (April 1981), pp. 126–143.
24. J. Schwartz, "Broad Agency Announcement for New Language in Rapid Construction of Software Prototypes", *Commerce Business Daily*, Arlington, VA, Feb. 1989.
25. W. Swartout, "GIST English Generator", in *Proceedings of the National Conference on Artificial Intelligence*, AAAI, 1982.
26. M. Tanik and R. Yeh, "The Role of Rapid Prototyping in Software Development", in *Proceedings of the 22nd Hawaii International Conference on System Science*, Kona, Hawaii, January 1989, pp. 337–338.
27. R. Terwilliger and R. Campbell, "PLEASE: Executable Specifications for Incremental Software Development", *J. Syst. Softw.* **10** (1989), 97–112.
28. J. Tsai, M. Aoyama and Y. Chang, "Rapid Prototyping Using FRORL Language", in *Proc. COMPSAC 88*, Oct. 1988, pp. 410–417.
29. S. Tyszberowicz and A. Yehudai, "OBSERV Object-oriented Specification, Execution and Rapid Verification System", in *3nd Israeli Conference on Computer Systems and Software Engineering*, Tel-Aviv, Israel, June 1988.
30. R. Yeh, N. Roussopoulos, Luqi and etc., "Research in Software Reusability", Final Report, Tech. Rep.-p106/83/0004-3, Ballistic Missile Defense Advanced Technology Center, Huntsville, Alabama, July 1985.