Beyond Exploratory Programming: A Methodology and Environment for Conceptual Natural Language Processing

Philip Johnson
Wendy Lehnert
Department of Computer and Information Science
University of Massachusetts
Amherst, Mass. 01003

Abstract

This paper presents an attempt to synthesize a methodology and environment which has features both of traditional software development methodologies and exploratory programming environments. The environment aids the development of conceptual natural language analyzers, a problem area where neither of these approaches alone adequately supports the construction of modifiable and maintainable systems. The paper describes problems with traditional approaches, the new "parallel" development methodology, and its supporting environment, called the PLUMber's Apprentice.

Introduction

AI software systems are rarely developed "by the book." Despite the plethora of design methodologies available today, AI programmers often perceive their problems as unamenable to solution through "structured" analysis and design techniques. Instead, we have come to rely on a set of unabashedly unstructured programming techniques or "design heuristics", collectively referred to as "exploratory programming." While these techniques may be sufficient for developing systems to test research hypotheses, they do not provide much aid in attaining the goals of production quality systems: reliability, extensibility, and maintainability.

This paper describes a software development methodology and environment for conceptual natural language processing, a domain in which the desire for production quality systems far exceeds their availability. The methodology attempts to find a middle ground between the inflexibility of structured design methodologies and the looseness of exploratory programming. This is accomplished by extending the exploratory programming paradigm to include the "structured growth" of requirements and specifications as well as implementation level descriptions of the system. From another perspective, it could be viewed as removing the strict ordering of requirements - specifications - design - implementation from traditional design methodologies.

To motivate this approach, we first examine the strengths and weaknesses of current design methodologies, and discuss why these approaches cannot be successfully applied to conceptual natural language processing. Next, the methodology and its supporting environment, called the "PLUMber's Apprentice", is introduced and examples of its use are described. Finally, several approaches for the automation of the design process within this paradigm are discussed.

Current Programming Methodologies

The majority of software development methodologies are based upon the traditional "software lifecycle," which divides the development process into the following stages:

- *Requirements:* The needs of the user community are assessed and described, usually informally.
- *Specifications:* Requirements obtained are used to produce a formal and complete description of the behavior of the final system.
- *Design:* High-level algorithms and data structures are constructed which together implement the functionality of the specifications. Modules and the interfaces between them are specified.
- *Implementation:* The design is translated into the source language.
- *Testing:* The system is checked to ensure it runs correctly and implements all specifications.
- *Maintenance:* The system is modified to support changes in functionality desired by the user community.

Many languages and tools have been developed for these phases, some examples of which are described in [1,15,20,25]. In addition, [3] gives an overview of several major design methodologies. An assumption underlying this work is "linearity" in the software lifecycle– requirements can be specified and fixed before specifications, specifications before design, and so on. Thus tools and techniques for design can rely on complete and fixed specifications, for example. While this assumption is perfectly valid in many problem areas, it is often violated in AI applications.

Sheil [23] describes some of the problems in using these approaches in problem domains where specifications cannot be completely specified and frozen in advance. Since specifications are used to generate the module structure and interfaces between them, the system structure reflects its initial functionality. Changes in the specifications which cut across module boundaries will be difficult to implement, due to both the inherent complexity of such a process, and because features of structured languages (like strong typing) tend to complicate those types of changes.

Automating the implementation process (i.e., making specifications "executable") is one answer to the problem of frequently changing requirements[9,11,28]. If the system is responsible for generating the implementation, then system development cen-

ters around the maintenance of specifications. While human intervention is usually required to make the system-generated implementation efficient, this can often be delayed until the requirements for the system have stabilized. However, all of these approaches require complete specification for the behavior of the system. Unfortunately, this type of specification language remains beyond the state of the art for problem areas such as vision, robotics, or natural language understanding, so the application of this approach in these areas does not appear to be imminent.

The exploratory programming approach begins by dismissing this view of the software lifecycle altogether. Sandewall [21] terms the development process "structured growth":

> *An initial program with a pure and simple structure is written, tested, and then allowed to grow by increasing the ambition of its modules... The growth can occur both "horizontally", through the addition of more facilities, and "vertically", through a deepening of existing facilities and making them more powerful in some sense.*

As Sheil (op. cit.) explains, this approach necessitates a great deal of automated support, including sophisticated environments for entering, inspecting, debugging, and modifying code. In addition, he claims it requires a language with late binding and weak type checking, in order to minimize the amount of language-level design "rigidity". Unfortunately, the lack of higher-level descriptions for requirements and specifications has a cost. One benefit of these descriptions is that they can aid the developer in implementing a system which naturally reflects the structure of the problem domain. An exploratory programmer has no such guidelines, and thus system structures can become highly idiosyncratic, dependent primarily upon the *order* with which the programmer decided to "increase the ambition of his modules." Requirements and specifications also have an important role as documentation, which can also be lost in exploratory environments.

The PLUMber's Apprentice

Conceptual Natural Language Processing.

Conceptual natural language processing (CNLP) is not well suited to either traditional methodologies or exploratory paradigms. It suffers from a variety of developmental problems, including:

- CNLP development generally proceeds by first developing a system to handle a small number of sentences, eventually extending this system to cover the full domain. Frequently this process resembles the building of a "house of cards", where minor changes can cause the whole structure to come crashing down, requiring extensive redesign.
- Maintenance is a terrible problem. In addition to the threat of a "fallen house of cards", CNLP designs tend to be highly idiosyncratic, based upon which particular subset of the domain was handled first, and how the developer decided to extend it outward from there. It is often difficult or impossible to understand how a particular modification alters the global behavior without re-running the system on a large set of sentences.
- Due to the immaturity of the field, there are a variety of techniques for CNLP: novices learn primarily by experimentation and first-hand experience.
- Compounding the above problems is the fact that natural language interfaces often need to evolve quickly and continuously– both in the front end (the types of sentences to be handled) and the back end (the set of commands or representations output by the interface).

These problems make it difficult to apply existing methodologies successfully. The change in requirements and specifications as the front and back ends evolve makes the traditional life cycle model unsuitable. The limited availability of generic CNLP strategies precludes automated approaches. What appears to be necessary is a combination of traditional and exploratory approaches. If the developer could create and manipulate higher level descriptions of the system, its conceptual structure and its limitations would become more apparent earlier in the development process. However, the developer must still be able to experiment with different implementations, and thus the high-level descriptions must be able to evolve easily and continuously with the implementation. The PLUMber's Apprentice is an attempt to achieve this goal of fluid yet explicit structure.

The "Consistency Criterion."

The PLUMber's Apprentice includes a set of languages which describe the developing system with differing levels and types of abstraction. These languages are analogous to those developed for formalizing the initial phases of the traditional lifecycle model– requirements, specifications, design, and implementation. The PLUMber's Apprentice also includes facilities similar to the "power" tools of exploratory environments– including a graphical interface providing several views of the evolution of memory structures during system execution, and editors for creating and modifying each description of the system[10].

Unlike either traditional or exploratory facilities, however, the PLUMber's Apprentice also includes tools for assessing and maintaining the *consistency* of each system description with each of the other descriptions. This feature has a number of implications:

- Unlike the traditional lifecycle model, where each stage must be completed before moving on to the next, the PLUMber's Apprentice allows the descriptions to be developed in any order. Furthermore, the system allows these descriptions to co-exist in a state of partial completion, as long as what has been specified is not contradictory.
- Consistency checking allows development to occur under a variety of paradigms. The traditional top-down approach, proceeding from abstract to concrete descriptions is, of course, supported. An "exploratory" paradigm, using simply the implementation language and the power tools is also possible. But a more powerful "parallel" development process is also possible. Since partial descriptions of the system at any level of abstraction are supported, the developer is free to "explore" the system structure on all of these levels simultaneously. This corresponds more closely to the way in which programmers naturally develop software: for example, implementation issues often arise during requirements analysis, or during implementation the

developer may become aware of additional implicit specifications. This information is usually lost – if not to the original developers, then certainly to their successors. The PLUMber's Apprentice allows these decisions to be captured at whatever point in the development process they are discovered.

- Finally, these tools allow the languages to be used in surprising ways. When the implementation is found to be in conflict with the higher level descriptions, the developer might modify the implementation to bring it into line with the "conceptual model". On the other hand, the developer might just as easily change the specification to more accurately reflect the new behavior. The mere existence of accurate high level descriptions serves as invaluable documentation aid. In addition, novices attempting to understand the system might test their understanding by adding new high level descriptions and seeing if they are consistent with the actual system. When extensions to the system are made, the Apprentice can ensure that they are faithful to the "spirit" of the system as expressed by the higher-level descriptions.

PLUM: The Implementation Level Language.

PLUM (The Predictive Language Understanding Mechanism) incorporates principles of conceptual sentence analysis developed by a number of researchers in the field of natural language processing [5,6,7,8,12,16,18,19,24,26,27]. Unlike syntactic sentence analyzers that strive to produce a syntactic parse tree in response to a sentence, a conceptual sentence analyzer attempts to produce a conceptual meaning representation that captures the semantic content of a sentence.

Particular constituents within a sentence (typically verbs) are associated with predictive concept frames that allow an understander to identify meaningful relationships among other parts of the sentence. For example, the active form of the verb "to give" will predict an actor or agent, an object, and a recipient. Syntactically, the actor is expected to correspond to the subject of the sentence, the object corresponds to the direct object, and the recipient corresponds to an indirect object or object of a prepositional phrase. The goal of a conceptual analyzer is to fill each slot in its top-level concept frame with appropriate slot fillers found throughout the sentence. Slots inside frames can be filled with memory tokens (normally associated with noun phrases) or other concept frames. For example, the top level concept frame for "John told Mary that he was going home" contains slots for an information source (John), an information recipient (Mary), and the information being transfered (the fact that John was going home). In this case, another concept frame representing John going home is needed to fill the object slot of the top level concept frame. An excellent introduction to conceptual sentence analysis can be found in [22]. Documentation specific to PLUM is available in [14] and [13].

In PLUM, a declarative structure is associated with each concept frame. This structure is called a `prediction prototype`. Prediction prototypes describe not only the concept frame to be instantiated (added to memory) during sentence analysis, but everything PLUM requires in order to achieve this instantiation as well. Four key components give PLUM all it needs to know:

- The `Concept Frame` describes the frame structure and specifies any slot constraints that narrow the set of possible slot fillers appropriate for a frame instantiation.

- The `Control Structure` specifies useful search routines for each slot in the Concept Frame. Some searches must look backward for information already present in the sentence while others must search forward, effectively waiting to see what the rest of the sentence holds in store. In either case, it may be necessary to halt a search at clause boundaries or other sentence boundaries. The Control Structure describes such search parameters by means of a simple grammar called an Expect Clause. When PLUM reads a Prediction Prototype, it interprets all the Expect Clauses within the Control Structure, producing executable search routines for possible run-time execution during sentence analysis.

- The `Predicted Prototypes` are a list of other Prediction Prototypes that will be triggered if the current Prediction Prototype is successfully instantiated. Prediction Prototypes can be triggered during sentence analysis two ways: (1) a word encountered in the sentence can trigger a Prediction Prototype, or (2) an instantiated Prediction Prototype can trigger new Prediction Prototypes. The first method corresponds to "bottom-up" sentence processing, while the second enables "top-down" sentence analysis. In conceptual sentence analysis, the general idea is to go bottom-up until you know enough to go top-down.

- The `Required Slots` list all slots in the Concept Frame that must be filled before the frame can be instantiated. This is an optional component within the Prediction Prototype, but a default requirement is assumed in the event that no Required Slots are specified. If this component is omitted, PLUM considers the corresponding Concept Frame to be instantiated only if one of its slots if successfully filled. Any slot will suffice, but at least one must be filled.

The idea of a Prediction Prototype is easiest to understand with a concrete example in hand. The following (slightly simplified) Prediction Prototype defines a Conceptual Dependency case frame for ATRANS, an abstract transfer of possession. This prototype is triggered by an active ATRANS verb (e.g. "to give").

```
(create-pred
  type (ATRANS)
  comment (triggered by 'give' & other active ATRANS
           verbs)
  concept-frame (actor = (animate)
                 act = ATRANS
                 object = (physobj)
                 source = (same-as actor)
                 recipient = (animate))
  control-structure
   ((expect actor in past referent)
    (expect object in future referent)
    (expect recipient in future referent)
    (expect recipient in future direction-to value))
  predicts (cd)
  required-slots (actor object recipient))
```

In this prototype, all slots are filled by referent frames (corresponding to memory tokens created by noun phrases) that satisfy the slot constraints specified by the Concept Frame. The object being transfered is expected to be a physical object while the recipient of the transfer is presumed to be animate. These slot constraints are needed to sort out direct objects and indirect objects, since the searches specified by the expect clauses only know to search for referents appearing after the verb. Notice that two expect clauses are defined for the recipient slot. One is designed to pick up recipients that appear as indirect objects (John gave Mary the book) and the other is designed to cover cases where the recipient appears in a prepositional phrase (John gave the book to Mary).

When a system designer builds a natural language interface with PLUM, he or she designs Prediction Prototypes as if programming in a highly declarative programming language. Since all slot constraints are readily found in the definition for the Concept Frame, and the searches used to fill these slots are described declaratively in the Control Structure, the role of any prototype in the processing of an individual sentence is relatively easy to understand. The difficulties arise when a large number of prototypes are designed to interact with one another in their slot-filling activities. It is much harder to anticipate all the possible interactions between prototypes that can arise in various sentences.

The traditional manner of controlling interactions is by periodically checking the system out on a large testbed of sentences. Whenever a new Prediction Prototype is added to a system that previously checked out, there is the possibility that some side effect from the new prototype will cause new interference effects within the previously consistent system of Prediction Prototypes. Unfortunately, this process only uncovers system inconsistencies, it does not provide any direction either in pinpointing the source of the inconsistency within the prototype definitions, or in determining the necessary modifications to the system. Often the modifications made to remove one inconsistency simply produce another.

To combat this problem, the PLUMber's Apprentice provides two languages: `abstract prototypes` for describing the system structure, and `abstract instantiation sequences` for describing processing strategies. Use of these languages aids the developer in understanding not only how to extend the system successfully, but also when the current system design is not adequate for the desired modifications.

Abstract Prototypes.

To help the developer manage the complexity of dozens or hundreds of prototypes in a system, the PLUMber's Apprentice provides `abstract prototypes`. Abstract prototypes are similar to Smalltalk Classes and Zetalisp Flavors, enabling the developer to specify inheritance hierarchies of the properties of prototypes. While Flavors and Classes impose "top-down" development (requiring the abstract description before an instance of it can be made), abstract prototypes can be developed "bottom-up" or "middle-out", in addition to top-down. For example, the developer might first develop a set of prototypes, then develop the abstract prototypes to modularize the system and to capture the common properties between them.

Figure 1 contains one useful hierarchy for VMSPLUM[13], a prototype natural language help facility for the VAX/VMS command language. In this case, the leaf nodes are PLUM prototypes, the others being abstractions. Sometimes the abstraction can specify properties that must hold true of its instances' structures (such as the requirement that `<command-frame>`s search for command qualifiers.) They might also group prototypes by related behavior, rather than structure (for example, `<values>`.) In the former case, the Apprentice can check to ensure that an addition to that class has the required properties. In the latter case, new additions to the class can be checked for the prerequisite behavior. More than one inheritance hierarchy could be mapped onto the same set of prototypes, resulting in multiple "perspectives" or viewpoints on the system.

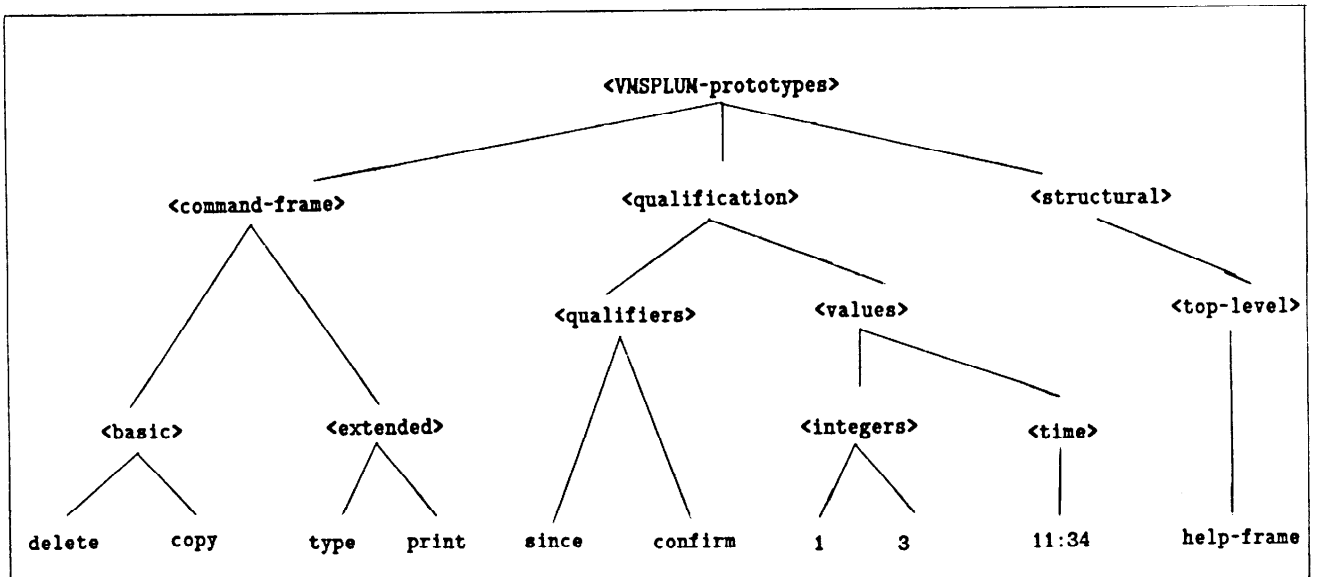

Figure 1: A Partial Prototype Abstraction Hierarchy for VMSPLUM.

The developer could have generated the abstract prototype hierarchy in Figure 1 in several ways. Perhaps a small interface handling only <basic> commands without qualifiers was implemented initially, and the structure grew "upwards" and "outwards" from there. This corresponds to the "exploratory" approach. Or, the developer might have begun by considering all the cases that could occur, and built "downwards" from there, corresponding to the "top-down" approach. The most efficient path is probably somewhere in between; perhaps first developing a "rapid prototype" implementation, then creating an abstract structural model, and then using that model to systematically extend the interface. The hierarchy aids greatly in this extension by providing a kind of "semantic type checking" on the prototypes. Finally, the hierarchy provides aid to the novice in understanding the similarities between and requirements of the dozens or hundreds of prototypes in a PLUM interface.

Abstract Instantiation Sequences.

The control structure of the PLUM implementation language is decentralized, being specified within each of the prototypes. While this has the advantage of automatically defining a "local context" for the search of memory, and a natural notation for description of an individual prototype's control structure, it has the disadvantage of leaving implicit the global behavior of the system, and thus obscuring global processing strategies. Abstract Instantiation Sequences are designed to allow the developer to make explicit the sequence of PLUM events necessary and sufficient for prototype instantiation. By making this sequence explicit, the rationale for the structure of the expect clauses becomes much more obvious. The sequences also serve an important debugging function, by providing a method for catching unanticipated interactions among prototypes which might result in their addition to memory in a context unforeseen by the developer.

The abstract instantiation language is related in spirit to Constrained Expressions [2,4]. The developer gives an algebraic expression, whose terms are "atomic PLUM events" and whose operators express sequencing and causality information. Atomic PLUM events are predicates whose arguments are simply prototypes, either abstract or implementation-level, and which denote the fundamental state changes of the system. For example, a few of the atomic PLUM events in VMSPLUM are:

- `(predict <command-frame>)`
  (Prediction of a member of the `<command-frame>` class of prototypes.)
- `(instantiate help-frame)`
  (Instantiation of the `help-frame` prediction prototype.)
- `(slot-fill <top-level> <command-frame>)`
  (A member of the `<top-level>` class of prototypes has a slot filled by a member of the `<command-frame>` class of prototypes.)

The operators determine the legal orderings of these events during system execution, and their relationship to each other. The current operators are:

- `||` *(Strict Concatenation)* System events must be directly contiguous in time.
- `...` *(Loose Concatenation)* One event follows another with any arbitrary intervening events.
- `OR` *(Alternation)* At least one of the events must occur.
- `->` *(Causality)* The first event "causes" the following event. Causality is context-sensitive; verifying that `Instantiate(X) -> Predict(Y)` requires different analyses than verifying that `Slot-fill(X,Y) -> Instantiate(X)`.
- `#` *(Non-ordered Occurrence)* The specified set of events must occur, but in any order.
- `NOT` *(Negation)* The specified event or expression must not occur.

For example, one global processing strategy used in VMS-PLUM is the following:

> The necessary and sufficient set of events for the instantiation of `top-level` frames are as follows: First, there must occur a prediction of an instance of a `<top-level>` prototype. After some (possibly zero) intervening events, an instance of a `<command-frame>` prototype is instantiated. This event causes the slot-filling of the `<top-level>` prototype, which causes its instantiation. Following this, no prototype of the class `<command-frame>` may be instantiated.

This strategy can be expressed by the following abstract instantiation sequence:

```
(def-instantiation-sequence (<top-level>)
    (predict <top-level>) ...
    (instantiate <command-frame>) ->
    (slot-fill <top-level> <command-class>) ->
    (instantiate <top-level>) ...
    (not (instantiate <command-frame>)))
```

A few of the errors which this abstraction can identify are:

- Ambiguities in the sentence which cause the instantiation of two `<command-frame>` prototypes.
- Modification to `help-frame` such that slot filling by a `<command-frame>` prototype no longer results in instantiation of `help-frame`.
- Instantiation of `help-frame` without slot filling by a `<command-frame>` prototype.

Future Directions.

The PLUMber's Apprentice is currently under construction at the University of Massachusetts. While its facilities have only been applied to small systems, such as the VMSPLUM interface, initial results have been encouraging. A number of directions for future extension of the system are currently under study, including:

- *A language for "assertions".* Occasionally the developer may want to constrain the behavior of the system in a fashion not directly related either to the structure of the prototypes, or to the sequence of events necessary for their instantiation. Frequently this takes the form of restric-

tions on the input (form of the query), or output (form of the final memory structure), or some intermediate stage of processing. To accomodate this, the Apprentice provides a "hook" for developers. At the present time, this simply takes the form of lisp function defined by the developer, which is called at a point in time during execution specified by the developer, and which may access (but not modify) the values of internal memory structures. Greater experience with these "assertions" about system behavior may allow the development of a language specifically for their expression.

- *User testing of the methodology and environment.* Much more experience with the environment needs to be obtained before definitive conclusions about its worth can be made. It would be interesting to determine what developmental "path" is taken by users in this environment-what are the methodological differences between novice and expert interface developers? Can abstract processing structures and strategies be identified and ported to new domains?

- *Automated system extension by analogy.* The abstraction languages provide a great deal of information about the structure and function of the implementation. Enough high-level information can be specified about the VMS-PLUM interface that the system can automatically add certain kinds of new VMS commands to the interface by analogy, though this requires making structural changes to certain prototypes, and adding new instances of abstract classes. It may be possible to provide facilities for extension by analogy at the system level, in addition to the prototype level.

- *Automated prototype hierarchy generation.* Given a set of prototypes, there exists an unbounded number of possible abstraction hierarchies. However, only a few of them reflect the conceptual structure of the system. It seems likely that by analyzing the use and occurrence of prototypes in successfully processed queries, automated aid in construction of the hierarchy could be provided.

Conclusion.

Programming methodologies in artificial intelligence serve a dual purpose that is not present in other areas of computer science. On the one hand, those of us involved in AI applications are trying to build useful and reliable systems for large user populations. On the other hand, those of us who engage in basic research often write programs for the sole purpose of testing out an idea or investigating a new problem area. It is not reasonable to assume that a single programming methodology will be equally effective in the service of both goals [17]. But what should we do when technologies discovered by basic research entail a level of conceptual complexity that makes technology transfer into application systems problematic? We can either reject the technology as being unmanageable and ill-conceived (as Dijkstra might recommend), or we can design tools to help us manage these more demanding levels of program complexity.

Techniques in natural language processing provide especially compelling arguments for more powerful software development tools, since the information processing requirements of natural language are both highly demanding and highly idiosyncratic. While traditional programming methods encourage us to identify and exploit linguistic regularities, the heart of the natural language problem is more accurately characterized by inevitable exceptions to almost any rule, non-generalizable irregularities, assumptions that might be wrong, and adequate (as opposed to correct) interpretations.

This paper argues for a new set of languages to aid the design of natural language systems. More specifically, we are implementing a set of specification languages for a conceptually-oriented language analyzer, PLUM. These languages describe declarative and procedural information about a developing language interface at varying levels of abstraction. Unlike most languages for specification or design, the PLUM abstraction languages do not impose a developmental sequence: the designer may freely intermix design, specification, and implementation. The specification-level languages can be used for debugging, program understanding, and prototype synthesis, as well as for specification. This freedom appears to allow a more "ergonomic" development process – one that is better suited to the way people think about systems during their development.

Acknowledgements.

this work supported in part by NSF Presidential Young Investigator Award NSFIST-8351863 and in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under contract no. N00014-85-K-0017.

The authors gratefully acknowledge comments and criticisms of the ideas in this paper by John Brolio and Jack Wileden.

REFERENCES

[1] M. Alford, "SREM at the Age of Eight: The Distributed Computing Design System", *Computer*, April 1985.

[2] G. Avrunin, L. Dillon, J. Wileden, W. Riddle, "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986.

[3] G.D. Bergland, "A Guided Tour of Program Development Methodologies", *IEEE Computer*, October 1981.

[4] R. Campbell, A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science,* vol. 16, Springer-Verlag, Heidelberg, 1974, 89-102.

[5] E. Charniak, "A Parser with Something for Everyone", *Technical Report No. CS-70*, Department of Computer Science, Brown University, Providence, RI. 1981.

[6] G. DeJong, "Prediction and Substantiation: A New Approach to Natural Language Processing", *Cognitive Science*, 3, 1979.

[7] M. Dyer, *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension*, MIT Press, Cambridge, MA. 1983.

[8] A.V. Gershman, "Knowledge-Based Parsing" (Ph.D. thesis) *Research Report 156*, Department of Computer Science, Yale University, New Haven, CT. 1979.

[9] C. Green, D. Luckham, R. Balzer, T. Cheatham, C. Rich, "Report on a Knowledge-Based Software Assistant", *Tech. Report KES.U.83.2*, Kestrel Institute, Palo Alto, Ca. July, 1983.

[10] P. Johnson, "Requirements Definition for a Plumber's Apprentice", in *Proceedings of the Second Annual Workshop on Theoretical Issues in Conceptual Information Processing*, May 1985.

[11] P. Kruchten, E. Schonberg, J. Schwartz, "Software Prototyping Using the SETL Programming Language", *IEEE Software*, October, 1984

[12] M. Lebowitz, "Memory-Based Parsing", *Artificial Intelligence*, 21, 4. 1983.

[13] W. Lehnert, K. Narasimhan, B. Draper, B. Stucky, M. Sullivan, "Experiments with PLUM", *Counselor Project Technical Memo No. 2*, May, 1985.

[14] W. Lehnert and S. Rosenberg, "The PLUM Users Manual," *Counselor Project Technical Memo No. 1*, May 1985.

[15] B. Liskov, S. Zilles, "Specification Techniques for Data Abstractions", *IEEE Transactions of Software Engineering*, Vol. SE-1, No. 1, March 1975

[16] S. Lytinen, "The Organization of Knowledge in a Multilingual, Integrated Parser", (Ph.D. thesis) *Research Report 340*, Department of Computer Science, Yale University, New Haven, CT. 1984.

[17] D. Partridge, Y. Wilks, "Does AI Have A Methodology Different From Software Engineering?", Unpublished Manuscript, New Mexico State University, 1986.

[18] C.K. Riesbeck and C.E. Martin. "Direct Memory Access Parsing", *Research Report 354*, Department of Computer Science, Yale University, New Haven, CT. 1985.

[19] C. Riesbeck and R. Schank, "Expectation-based Analysis of Sentences in Context", *Research Report 78*, Department of Computer Science, Yale University, New Haven, CT. 1976.

[20] D. Ross, K. Schoman, Jr., "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering* Vol. SE-3, No. 1, Jan. 1977.

[21] E. Sandewall, "Programming in an Interactive Environment: The Lisp Experience", *Computing Surveys* 10(1), 1978.

[22] R.C. Schank and C.K. Riesbeck, *Inside Computer Understanding*, Hillsdale, NJ, Lawrence Erlbaum Associates. 1981.

[23] B. Sheil, "Power Tools for Programmers," *Datamation*, February, 1983.

[24] S. Small, "Word Expert Parsing: A Theory of Distributed Word-based Natural Language Understanding", (Ph.D. thesis) TR-954, Department of Computer Science, University of Maryland. 1980.

[25] D. Teichroew, E. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, Jan. 1977

[26] D.L. Waltz and J.B. Pollack, "Phenomenologically Plausible Parsing" in *Proceedings of the 1984 American Association for Artificial Intelligence Conference*, 1984.

[27] R. Wilensky, "A Knowledge-based Approach to Language Processing" in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1981.

[28] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 5, May 1982.