Inferring Software System Structure

Philip Johnson

April 1988
COINS Technical Report 88-46

Abstract

Current AI software tends to be difficult to understand, extend, and reuse in new contexts. This is in part due to peculiarities in the AI software lifecycle, which makes conventional software engineering techniques for the development of robust software difficult to apply. This paper introduces *software structure inference*, a technique for describing architectural properties of software automatically. Software structure inference recasts several software engineering techniques into a form compatible with the AI software lifecycle. We describe software structure inference, a prototype implementation and its use on an AI system, and discuss its potential for improving the robustness of AI software.

This work is supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract F30602-85-C-0008, and by the Office of Naval Research, under a University Research Initiative Grant, Contract #N00014-86-K-0764.

1 Introduction

The current state of artificial intelligence software development (AISD) has a paradoxical nature: while the languages and tools emphasize support for reuse and modification, the programs actually produced are often extremely brittle[1]. A major cause of this paradox is the peculiar nature of the AI software lifecycle. While more conventional problems allow a careful analysis of requirements and specifications before attempting the design and implementation of a software system, the user needs and/or functional capabilities of most AI software can only be established through experimentation with a succession of implementations. In the worst case, an implementation may change the problem to be solved altogether!

For these reasons among others, the conventional software lifecycle model fails to adequately describe AI program development [Par86,She84,SB82]. Viewed from that perspective, AI software simply spends its entire life in the implementation phase. This renders many software engineering techniques for robust software development ineffective, as they are designed for use during the requirements, specifications, or design phase of development. As a result, while the initial implementation of an AI program might exhibit a clear and even elegent structure, the changing needs for the system force fundamental restructurings of large volumes of code. Eventually, inconsistencies accumulate until understanding the now idiosyncratic software requires extensive knowledge of prior implementations and their motivating functionalities.

In our view, AISD can be greatly aided by recasting software engineering techniques for robust software development into a form suited to the AI software lifecycle[2] One major need is to better understand the architectural qualities of an AI system and how they change over time. To that end, this paper introduces a technique called *software structure inference.* Software structure inference combines knowledge derived from analysis of the static and dynamic properties of the software with knowledge about software engineering principles for robust software structure to produce descriptions of the structure of the system automatically. These descriptions aid in identifying the architectural outlines of the system, groups of related functions and objects, and the software context required for system components. This information helps in assessing architectural quality, incorporation of new functionality, and extraction of system components for reuse in new situations. In sum, they provide a step towards maintaining the robustness of AI software throughout its lifetime.

In the following section we describe some software engineering principles for robust software, and problems with their application within AISD. We follow this with a description of a prototype implementation of software structure inference, and conclude with an assessment of the technique and future research directions.

2 SE Principles for Software Structure

In conventional software development, after the requirements and specifications of the system have been determined, the algorithms to implement the specifications and a supporting system structure are defined. From the variety of techniques to accomplish this design phase[Ber81], four properties have emerged as generally indicative of robust software[Par72]:

[1] We use "brittle" in the software engineering sense to mean software that is difficult to change.

[2] Barstow [Bar87] surveys additional approaches to combining software engineering and AI.

- *Low Coupling:* Coupling refers to the measure of strength of interconnection between modules in a system. Minimization of the coupling between modules is important since their greater independence allows them to change without conflict with other modules and aids in their reuse in new contexts.
- *High Cohesion:* Cohesion refers to the measure of strength of connection among components of a module. Cohesion has been decomposed into seven types [SMC74]: functional, sequential, communicational, procedural, temporal, logical, and communicational. These types of cohesion measure relationships between components such as their dependence upon common data, their co-occurrence temporally and within the control flow of the program, and their structural, functional, and logical similarity. High levels of cohesion within a module are desirable because they facilitate understanding the capabilities of the module and its role within the architecture of the system.
- *Information Hiding:* Information hiding, following Parnas [Par79], involves the identification of *secrets*, or things that change, and then the hiding of these secrets within modules through the creation of an interface which is insensitive to changes in the secrets. Information hiding is important because it facilitates internal changes without impact outside of the module.
- *Hierarchical structuring:* A hierarchical structure of modules is important because it supports delegation of responsibilities for different modules to different groups, reduces the tendency to create "spaghetti code," and aids in maintaining the intellectual manageability of the system.

2.1 Problems with Application of SE Principles to AISD

2.1.1 The Need for Specification-level Knowledge

In conventional design methods, the specifications are used to drive the design of a system structure which embodies these properties as well as the desired functionality. Since no implementation yet exists, the design process is free to structure the software in the optimum manner for the required specifications.

Unfortunately, the difficulty of the problems attempted in AI (such as natural language understanding, vision, etc.) means that no formal specifications corresponding to the requirements can be created. Without a priori knowledge of the specifications, AI software developers resort to discovering them through experimentation with implementations whose initial functionality is a small subset of the ultimate capabilities desired [San78]. Without much predictability in the way the specifications might change, creation and maintenance of desirable structural properties becomes a difficult task.

An alternative is to embed the conventional lifecycle model within each implementation iteration. In this case, before each implementation, a new set of requirements and specifications are generated and used in the design of a system structure which satisfies both the specifications and the desired qualities of low coupling, high cohesion, information hiding, and hierarchical structuring. While this method would dramatically increase the robustness of the resulting systems, it has a practical barrier to its application. By "custom coding" each implementation to precisely fit the current specifications, much of the previous implementation becomes unavailable to its successor. For research-oriented software development

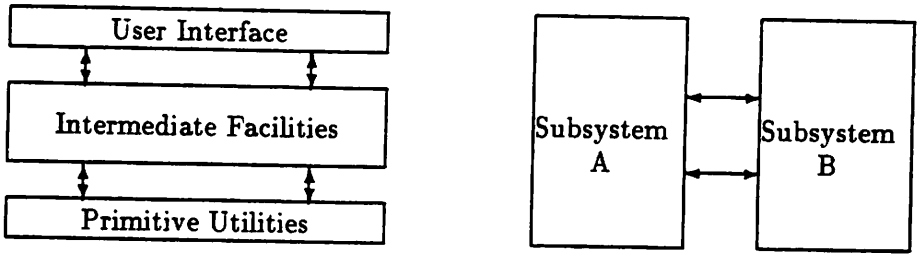

Figure 1: Two orthogonal views of system structure

in particular, this often entails unacceptably high overhead[3].

2.1.2 Lack of multiple viewpoints of the system architecture

Conventional design techniques result in a single hierarchical decomposition of the system structure, which well serves the managerial aspects of the upcoming implementation phase. However, in AISD, the primary role of structural descriptions is to aid in reuse and extension of an existing implementation. In this case, more information is provided by several "orthogonal" decompositions of the system structure. For example, a system may well be described by both *horizontal* decompositions, which emphasize the layers of abstraction in the code, and *vertical* decompositions, which emphasize the the supporting machinery required for the implementation of facilities. (See Figure 1.)

In general, horizontal viewpoints often provide perspectives useful for extension of system facilities, while vertical viewpoints aid in identifying how to extract system components for reuse. (Of course, a single system decomposition may contain a combination of vertical and horizontal decompositions.)

2.1.3 Insufficient support for polymorphism

Most AI systems display polymorphism. Two of the most common forms of polymorphism are the ability of functions to operate on arguments of different types, and the ability for the same data object to be viewed as an instance of different data types. Conventional software engineering modularity mechanisms like Ada packages [Ich80] or PIC [Wol85] do not allow the different *polymorphic variants* of a function or data object to be represented separately. This can obscure the actual dependencies among system components and make component reuse more difficult.

Consider the following illustrative example, and the two representations of it in Figure 2. Function **eat** accepts objects of type *banana* or *potato*. Depending upon the argument type, **eat** will call (among other modules) **peel** or **boil**. A very common scenario in AISD is the desire to extract a polymorphic function and its supporting facilities for use in a new context with a different combination of argument types. Conventional modularity mechanisms view **eat** in the manner of the left-hand diagram, which gives no aid in determining the dependencies between the type of argument and the required facilities to process it. A

[3]Transformational approaches [Bal85,SKW85] show promise in overcoming some of the overhead associated with this alternative, however.

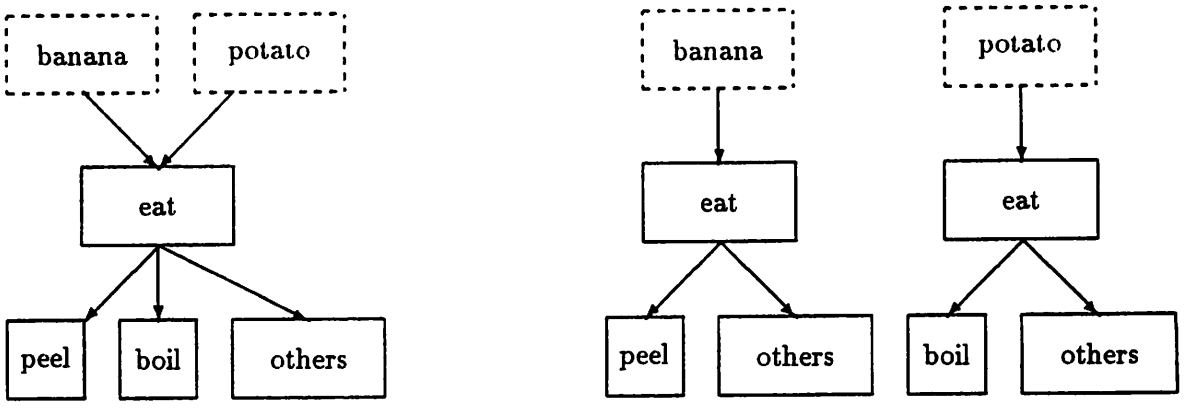

Figure 2: Two modularizations of polymorphism

better representation, such as the one represented on the right-hand side of Figure 2, makes these dependencies explicit.

2.1.4 Insufficient support for second order system structure

A final problem with conventional usages of SE structuring principles is in supporting *second order* system structure, or the functions and data types defined dynamically by the system itself. The use of second order system structure varies from small numbers of functions which return functions as their value, to application-generation systems which define an entire system dynamically. The Generic Blackboard System (GBB) [CGM86] is one example of the complexity that second order system structures may possess. GBB generates a highly optimized blackboard implementation based upon an application specification. The generated system not only displays a complex internal structure, but also complex relationships with GBB itself, which is part of the run-time environment. Compared to conventional program generation systems like compilers, blackboard technology is in its infancy, and the requirements for GBB are rapidly evolving. Understanding and restructuring not only GBB but the systems it generates poses a major software engineering problem for this project.

In summary, though AI software suffers from brittleness, the special nature and requirements of the AI software lifecycle and AISD prevent the straightforward application of conventional techniques for robust software development. Software structure inference addresses each of the problems with conventional techniques described above. Rather than using specification-level knowledge of the system, software structure inference relies on implementation-level knowledge. Different structural viewpoints can be generated by varying the way the knowledge about software structure is applied to the system's static and dynamic properties. Finally, the use of static and dynamic properties allow representation and enhanced understanding of polymorphism and second order system structure.

3 Midwife: A Prototype Implementation

MIDWIFE[4] is a prototype implementation of software structure inference. MIDWIFE is imple-

[4]Modularity Inference, Description, With Information For Evolution

Cluster Attributes	
Function Members	The functions which comprise this cluster.
Member Attributes	The static and dynamic information collected for each function member of this cluster.
Component Clusters	The clusters which comprise this cluster. (Initial clusters have no components.)
Interface	Includes members called by external clusters, external clusters and their members called by this cluster, data objects used and referenced, etc.
Cluster Ratings	Evaluations of this cluster by SE-Criteria. Provides a partial description of *why* the cluster was created from its components.

Table 1: Major attributes of cluster objects

mented in Common Lisp on Texas Instrument Explorers and produces structural descriptions of Common Lisp programs. The implementation of software structure inference by MIDWIFE can be separated into three phases. First, static and dynamic properties of the software are collected. Second, an initial set of *clusters*, the units of modularity in MIDWIFE, are created. Third, these initial clusters are incrementally organized into larger clusters representing the overall architecture of the system. This organization process is guided by heuristic approximations to the software engineering principles for robust structure. By varying the importance of the different principles in guiding the organization, MIDWIFE can generate different descriptions of the system structure.

3.1 Static and Dynamic Analysis

The static analysis component of MIDWIFE collects information about the functions of interest in the system, the calling hierarchy among them, and the data objects and global variables of interest and where they are referenced or modified within a function.

Dynamic analysis collects information about the behavior of the system during its execution. Dynamic analysis begins by encapsulating each function identified during static analysis with code to collect information about the objects passed to, manipulated within, and returned from these functions. Dynamic analysis also augments information about the static calling hierarchy with information about functions called dynamically (using `funcall` or `apply`), and objects referenced or modified dynamically (using `eval`).

3.2 Initial Cluster Formation

Once data on the static and dynamic behavior of the system has been collected, the initial "atomic" set of *clusters* are created. The most important attributes of cluster objects are shown in Table 1.

When the initial set of clusters are created, at least one cluster is created for each function. If a function exhibits polymorphism in its argument types, then separate clusters will be created for each of these variants. Following our previous whimsical example, two clusters

would be created for eat, one collecting dynamic information about it when called with *banana*, and one collecting the dynamic information about it when called with *potato*. Both of these clusters would access the same static information about eat. This separation of polymorphic variants into separate clusters overcomes the limitations of conventional modularity mechanisms in dealing with polymorphism as described in Section 2.1.3.

3.3 Creating Composite Clusters

Once the initial set of clusters is created, they are incrementally aggregated into new clusters until the entire system is modularized. The following pseudocode describes this process:

```
Set clusters-to-cluster to the initial clusters
While not finished clustering
        Rate all pairwise combinations of the elements of clusters-to-cluster
        Determine the clusters to comprise the new cluster
        Create new-cluster
        Add new-cluster to clusters-to-cluster
        Delete components of new-cluster from clusters-to-cluster
```

These steps can be elaborated as follows:

- **Rate all pairwise combinations of the elements of clusters-to-cluster.**
  This is accomplished in two steps:
  1. Each pair of clusters is evaluated against heuristic encodings of software engineering criteria for good system structure, called *SE-Criteria*. These encodings return an evaluation of the degree to which the composite cluster would satisfy its criteria for good system structure. The criteria themselves are described in Section 3.4.
  2. The combined rating for the composite cluster is computed by the weighted sum of the individual ratings given by the heuristics.
- **Determine the clusters to comprise the new cluster.**
  Determining the cluster to create is normally made by choosing the cluster with the highest aggregate rating from the SE-Criteria. However, this can lead to many superfluous subclusters within a cluster. To ameliorate this, if the highest set of cluster ratings consist of cluster pairs with similar ratings for the same criteria, and each function member in this highest set is paired with all of the others, then the entire set of clusters is aggregated at once.
- **Deciding when to stop clustering.**
  One technique is to stop only when the set of clusters to cluster has been reduced to a single cluster which has all the other clusters as members. Unfortunately, this will often result in superfluous clusters. To prevent this, clustering ends when a threshold aggregate value for clustering is not reached by the highest rated cluster pair.

3.4 SE-Criteria: Heuristics for Robust System Structure

MIDWIFE contains heuristics providing quantitative approximations of coupling, cohesion, information hiding, and hierarchical structuring. In general, each heuristic is passed two clusters, and evaluates the benefits to the system structure of creating a composite of the two clusters. The current set of heuristics in MIDWIFE are briefly described below:

1. **Functional-communicational-cohesion**
   This heuristic evaluates the structural similarity between two clusters based upon:
   (a) Similarities between the types of arguments passed, manipulated, and returned;
   (b) Similarities between the arguments/results relationships (for example, if the return value type is the type of a component of the argument).
   (c) Polymorphic similarity (the extent to which the members of the two clusters are polymorphic variants of each other.)
2. **Procedural cohesion**
   This heuristic evaluates the degree to which the members of the two clusters are called together within a common function in the system.
3. **Sequential cohesion**
   This heuristic evaluates the extent to which one cluster uses the results returned by another cluster.
4. **Functional information hiding**
   This heuristic returns a high value if the composite of the two passed clusters results in a function no longer being called outside the composite.
5. **Data information hiding**
   This heuristic returns a high value if the composite of the two passed clusters results in a data object no longer being referenced outside the composite.
6. **Layering**
   This heuristic measures whether the composite of the two passed clusters will aid in the creation of a horizontal layer in the system.
7. **Looping**
   This heuristic measures whether the composite of the passed clusters will eliminate loops in the call graph of the current level of modularization of the system.

3.5 Analyzing Unimem with Midwife

MIDWIFE has been used to generate structural descriptions of Unimem [Leb87], an AI system for incremental concept formation. Unimem takes a set of *instances* as input, and generates a hierarchical organization of *generalization-nodes*. Each instance consists of a set of *features* with associated *values*. The implementation of Unimem used is a small AI system (approximately 80 functions), but is sufficient for demonstration purposes. After analysis of the source code and the run-time characteristics, MIDWIFE can generate a variety of descriptions of Unimem, depending upon the weightings assigned to the heuristic SE criteria. Two descriptions are discussed below.

One description produced by MIDWIFE is roughly horizontal in nature. The first aggregate clusters created are those that implement the various primitive structures in Unimem, such as instances, features, and generalization-nodes. These clusters group together not only the accessor, creation, and other functions implicitly defined by `defstruct`, but also user-defined functions such as a less-than predicate on features, and a predicate to see if two features contradict. Other clusters group together the top-level functions which form the interface to Unimem, and the remaining functions "in between." This description is useful for identifying the overall architecture of the implementation.

Another description is less horizontal in nature. In this description, MIDWIFE concentrates on clusters which hide features of the implementation such as the data or functional objects employed. One resulting cluster "hides" the functions associated with the implementation of one aspect of the generalization hierarchy as an ordered list, maintained by a binary search function. This description is useful, of course, if reimplementation of that representation was desired.

3.6 Limitations of Midwife

As a prototype, MIDWIFE implements only a subset of the analysis facilities and software engineering knowledge possible within the software structure inference paradigm. The most important limitation of the current implementation is that Common Lisp structures are the only object type handled. Thus, systems that make extensive use of other objects (such as lists or flavors) as the major or sole data object representation cannot be well analyzed by the current implementation. In addition, the machinery for recognizing dynamic code generation is not yet implemented, which limits its ability to recognize and analyze the second order structure of systems.

A more fundamental problem with the current implementation of MIDWIFE is the somewhat ad hoc nature of the weightings and combining functions required to determine which clusters to aggregate at any particular point. Resolution of this difficulty will require more experimentation with MIDWIFE on a wide variety of systems, as well as enhancements to MIDWIFE described below.

4 Future Directions

While initial experiences with MIDWIFE are promising, more work on the prototype is needed, both to broaden the range of Common Lisp programs that can be analyzed, and to increase the sophistication of the software engineering knowledge used and the manner in which it is applied. The major tasks in this area are:

- The range of object types recognized will be extended. Both internal extensions to support object-oriented type systems and user-level object definition facilities will be supported. In addition, the machinery to recognize and support second-order system structure will be implemented.
- More heuristics for software structure assessment will be developed, and the sophistication of those currently available will be increased.
- The clusterization algorithm must become more sophisticated. A likely method will be to recast it within the blackboard paradigm, encoding each heuristic as a knowledge source, and providing control knowledge sources to determine how and when to apply the heuristics.

The motivation for MIDWIFE is not simply the analysis of software systems in isolation, but to provide aid in understanding how systems evolve over time. One application of MIDWIFE will be to track the evolution of an AI system, documenting the kinds of structural changes that occur to the system. From this data we hope to learn more about general kinds of changes to software in AISD and their effect. This kind of information could then be used

to create a language for expressing the kinds of changes *intended* for the system structure, with MIDWIFE providing an analysis of the effects of such a change.

Finally, the data from MIDWIFE can be used as input to other systems which monitor higher-level issues in the software development process. For example, the plan-based system of Huff and Lesser [HL87] could use the information about the developing system provided by MIDWIFE to infer more detailed knowledge about the development process and provide better assistance to developers.

References

[Bal85] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11), November 1985.

[Bar87] D. Barstow. Artificial intelligence and software engineering. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 200–211, March 1987.

[Ber81] G. Bergland. A guided tour of program design methodologies. *IEEE Computer*, 13–37, October 1981.

[CGM86] D. Corkill, K. Gallagher, and K. Murray. GBB: A generic blackboard development system. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1008–1014, Philadelphia, Pennsylvania, August 1986.

[HL87] K. Huff and V. Lesser. *A Plan-based Intelligent Assistant that Supports the Process of Programming.* Technical Report 87-09, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, September 1987.

[Ich80] J.D. Ichbiah. *Reference Manual for the Ada Programming Language.* United States Department of Defense, July 1980.

[Leb87] M. Lebowitz. Experiments with incremental concept formation: UNIMEM. *Machine Learning*, 2(2), September 1987.

[Par72] D. Parnas. On the criteria to be used in decomposing a system into modules. *Communications of the ACM*, 15(12), December 1972.

[Par79] D. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[Par86] D. Partridge. Engineering artificial intelligence software. *Artificial Intelligence Review*, 1(1):27–41, 1986.

[San78] E. Sandewall. Programming in an interactive environment: the Lisp experience. *Computing Surveys*, 10(1), 1978.

[SB82] W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, July 1982.

[She84] B. Sheil. Power tools for programmers. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, McGraw Hill, Inc., 1984.

[SKW85] D. Smith, G. Kotik, and S. Westfold. Research on knowledge-based software environments at kestrel institute. *IEEE Transactions on Software Engineering*, SE-11(11), November 1985.

[SMC74] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, May 1974.

[Wol85] A. Wolf. *Language and Tool Support for Precise Interface Control.* Technical Report 85-23, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, September 1985.