Classification and retrieval of reusable components using semantic features

Article · November 1995

DOI: 10.1109/KBSE.1995.490128 · Source: CiteSeer

CITATIONS
29

READS
507

3 authors, including:

John Penix
Google Inc.
65 PUBLICATIONS **1,831** CITATIONS
SEE PROFILE

Perry Alexander
University of Kansas
121 PUBLICATIONS **546** CITATIONS
SEE PROFILE

Classification and Retrieval of Reusable Components Using Semantic Features

John Penix, Phillip Baraona and Perry Alexander
Knowledge-Based Software Engineering Lab
Department of Electrical and Computer Engineering
University of Cincinnati
Cincinnati, Ohio 45221-0030
jpenix,pbaraona,alex@ece.uc.edu

Abstract

Automated assistance for software component reuse involves supporting retrieval, adaptation and verification of software components. The informality of feature-based software classification schemes is an impediment to formally verifying the reusability of a software component. The use of formal specifications to model and retrieve reusable components alleviates the informality, but the formal reasoning required for retrieval introduces questions of scalability. To provide scalability, current retrieval systems resort to syntactic classification at some level of abstraction, abandoning the semantic information provided by the specification. In this paper, we propose a methodology that shifts the overhead of formal reasoning from the retrieval to the classification phase of reuse. Software components are classified using semantic features that are derived from their formal specification. Retrieval of functionally similar components can then be accomplished based on the stored feature sets. Formal verification can be applied to precisely determine the reusability of the set of similar components.

1 Introduction

Reuse is a popular design methodology common to engineering disciplines. Its popularity arises primarily from two aspects: (a) cost reduction resulting from not designing a new solution; and (b) increased confidence in the solution because it has been successfully used. For reuse to be an effective problem solving methodology, the designer must be able to retrieve appropriate solutions, adapt a solution to fit the new problem, and evaluate the resulting solution.

As in other design domains, reuse is a popular software design methodology. However, automated software reuse support has been slow to emerge due to the difficulty of providing a useful representation for software components. The design representation must be able to efficiently support component retrieval, adaptation and verification. These activities require a high level understanding of component function that is not always reflected by a software component's structure or syntax. This semantic gap makes it difficult to recognize a potentially reusable component given a requirements specification.

The methodology presented in this paper allows the design environment to provide potential reuse candidates when given a problem specification. It is not intended to be a stand alone component retrieval mechanism, but to be integrated into a CASE environment that supports all three phases of software reuse. With formal verification in mind, formal specifications are used to describe problem requirements and potential solutions.

Our current goal is to support the design of digital signal processing (DSP) systems as architectures of interconnected components. Confinement to the DSP domain is helpful because most computations in this domain are performed over sequences of data. This common model of computation simplifies the classification of a component's function within the domain. Formal theories describing classes of component behavior in terms of this general model are provided by a domain expert. These theories are used in conjunction with a formal inference system to derive features for component retrieval. The cost of inference is reduced by storing features with candidate solutions and deriving features from the problem description only once.

In the next section, current methodologies are described and impediments to integrating them into an automated design support environment are discussed. This is followed by an overview of the use of formal specifications to verify the reusability of software com-

1068-3062/95 $4.00

ponents, including a description of the formal specification language used in our current application. Next, a method for classifying reusable components using a set of semantic features derived from a formal specification is presented. The use of feature sets to retrieve functionally similar components from a database is then described. Finally, related approaches to applying formal methods to software reuse are compared and contrasted, and conclusions are made.

2 Existing Methodologies

A popular method for describing libraries of reusable software components is a faceted classification scheme [1, 2]. Using a this methodology, components are classified by a set of attribute-value pairs, or *features*. The classification is performed by a domain expert, who is required to analyze the database of software components and classify them according to predefined terms. The domain expert's knowledge is implicit in the classification. To provide a basis for similarity calculations, the terms that represent the set of possible values for a feature are often related by a conceptual distance graph [1]. The informality and imprecision of these classification schemes complicates the automation of the overall reuse process. Automation of the classification process requires reverse engineering from source code. The imprecision of the classification scheme does not support formal component verification; reasoning about identically classified components requires source code analysis.

The use of formal specifications to augment software reuse has been proposed to solve these problems [3, 4, 5, 6, 7, 8, 9, 10]. There are many benefits to applying formal methods to software reuse. First, formal specifications provide an explicit representation of the function of a software component free from many implementation details. This is valuable because function is the primary point of interest when determining reusability. Next, the expressiveness of formal specification languages allows precision beyond that of faceted classification. Equivalent specifications perform equivalent functions. Finally, formal specifications and their associated formal system provide a basis for automated reasoning. A formal specification defines the function of a software component in terms of a domain model, a collection of axioms that define the data types and operations used in the system. Formal reasoning based on the domain model can be used to logically verify the reusability of a software component.

The high cost of theorem proving as a retrieval mechanism limits the scalability of reuse based on formal specifications. To alleviate this problem, existing systems rely on less precise, syntactic classifications at some level of abstraction. In exchange for the efficiency of imprecision, the functional information that is the basis for determining reusability is sacrificed. This paper describes a methodology for classifying and retrieving formally specified reusable software components based on functional semantics. To lower the overhead incurred by the use of automated reasoning during retrieval, we shift part of this burden to the classification process. A software component is classified by a set of semantic features that are formally derived from the component specification.

3 Formal Specifications and Software Reusability

A formal software specification describes the function of a piece of software free from most implementation details. It states what a piece of software does without stating how it does it. Determining the reusability of a software component requires modeling both the desired component as well as the library of existing solutions. Formal specifications can be used for both of these purposes.

Our current design needs can be described in terms of a problem specification [11]. Given the domain and range of the problem to be D_p and R_p, a problem specification has the form

$$\forall x : D_p, \exists z : R_p \cdot I_p(x) \Rightarrow O_p(x, z)$$

where $I_p(x)$ and $O_p(x, z)$ are the input and output conditions, respectively. The input condition specifies the set of domain values that a have a defined output, called the *legal inputs* to the problem. The output condition is a relation over $D_p \times R_p$ that specifies the functionally acceptable outputs for each legal input. This set of outputs is called the *feasible outputs* of the problem. The fact that this is a relation allows multiple feasible outputs for any one legal input. Because this type of specification describes the Domain, Range, Input and Output of a component it is referred to as a *DRIO* [12] model.

A software component, f_c, can also be described by a formal specification:

$$\forall x : D_c \cdot I_c(x) \Rightarrow O_c(x, f_c(x))$$

where $f_c : D_c \rightarrow R_c$. The only difference is that the feasible outputs are constrained to be well defined (by the function f_c) over the legal inputs.

In a formal view of software reuse, the ideal goal is to find an existing software component that *satisfies* a problem specification. If such a component

does not exist, then a similar component should be retrieved and adapted to satisfy the problem. A component *satisfies* a problem specification if, for any of the problem's *legal inputs*, the component results in one of the problem's *feasible outputs*. Formally, component C satisfies problem P if the following two conditions hold:

$$\forall x : D_p \cdot I_p(x) \Rightarrow I_c(x)$$

$$\forall x : D_p \cdot I_p(x) \wedge O_c(x, f_c(x)) \Rightarrow O_p(x, f_c(x))$$

The first condition states that any legal input to the problem will be a legal input to the component. The component specification assures that a legal input to the component results in a feasible output from the component. The second condition states that all feasible outputs of the component for a legal problem input are valid outputs of the problem. In the case of an illegal problem input, the behavior of the component is unconstrained, so any output is allowed. These statements provide a transitive proof that the problem's legal inputs result in one of the problem's feasible outputs. Given a problem specification and a component specification, reusability can be demonstrated by proving the two satisfaction conditions hold.

3.1 VSPEC

Our current application involves supporting the design and synthesis of digital signal processing systems using VHDL [13]. VHDL [14] is a hardware description language used for digital system design and simulation. To allow formal specification of VHDL components, VSPEC [15, 16], a Larch interface specification language for VHDL was developed. The Larch style of specification [17] is described as a two-tiered approach because specifications are written using two languages. The lower level is written using the Larch Shared Language (LSL). LSL is an algebraic specification language that is used to model abstract data types. Abstract types are specified by declaring sorts and operators that are analogous to types and functions in programming languages. The upper level of a Larch-style specification is the interface specification. Because calling conventions vary widely among programming languages, interface specifications are written in a language tailored to a specific programming language. The behavior of each component is defined using a vocabulary provided by an associated LSL specification.

A VHDL *entity* declares a digital component by defining the component's interface. However, in VHDL the function of the component is not defined in the entity structure. Instead, each entity has one or more associated architectures that define the behavior of a specific implementation of the entity. The VSPEC language annotates VHDL by adding clauses to support declarative specification at the entity level. The seven VSPEC clauses are:

- **requires** - specifies the input conditions for an entity.
- **ensures** - specifies the output conditions for an entity.
- **constrained by** - specifies performance constraints on an entity.
- **modifies** - specifies what the entity may alter.
- **based on** - associates VHDL data types with LSL sorts.
- **state** - defines a collection of variables that represents the internal state of the entity.
- **includes** - denotes shared language files that are referenced

In terms of component reuse, we are most interested in the **requires** and **ensures** clauses because they specify the function of the entity.

The following is an example of a VSPEC entity declaration for a search component:

```
ENTITY search IS
  PORT (input : IN recordArray;
        key : IN INTEGER;
        output : OUT recordType)
  REQUIRES sorted(input);
  ENSURES member(output, input)
          AND output.keyval = key;
  INCLUDES "record.lsl";
END search;
```

The **port** clause is the VHDL declaration of the component's input and output types. In this example, there are two inputs, a **recordArray** and an **integer**, and an output that is a **recordType**. The formal definition of the sorts and operators for records is given in **record.lsl**. The **requires** clause states that **input** must be sorted for proper execution of the entity. The predicate **sorted** is an operator defined in the file **record.lsl** that only returns true if the input array is ordered. The **ensures** clause guarantees that **output** is an element of **input** and that its **keyval** matched the search key.

From VSPEC, the *DRIO* model can be constructed using the following rules:

$D = d_1 \times d_2 \times \ldots \times d_n$ where each d_k is the sort (defined by the based on clause) representing the type associated with an **in**, **inout**, or **buffer** port or a state variable

$R = r_1 \times r_2 \times \ldots \times r_m$ where each r_j is the sort representing the type associated with an element in the **modifies** list

$I(x : D) = I_v(x : D)$ where $I_v(x : D)$ is the logical sentence defined by the **requires** clause

$O(x : D, z : R) = O_v(x : D, z : R)$ where $O_v(x : D, z : R)$ is the logical sentence defined by the **ensures** clause

Therefore, given a VSPEC problem specification and a VSPEC component specification, it is possible to determine the reusability of the component with respect to the specification. The next section describes a classification scheme that is used to provide a set of functionally relevant components for reuse verification.

4 Classification via Semantic Features

Within the formal model of software reuse, the goal of matching existing function to desired function is transformed into showing that a component specification logically satisfies a problem specification. This satisfaction proof can be carried out using automated theorem proving techniques. However, attempting this proof over a large database of reusable components is not practical. To provide scalability, the theorem proving activity must be restricted to a small subset of the database. Most existing systems use syntactic measures to determine this subset. Such a methodology works only when a specification's behavior is derivable from its structure. Thus, at high levels of abstraction, syntactic comparison necessarily fails. This section describes a semantic classification scheme that provides a set of reuse candidates.

4.1 Semantic Features

To efficiently compute semantic similarity, the problem and component specifications are classified by a set of $(attribute, value)$ pairs, called *features*. The features represent some aspect of the component's function in the following manner. Associated with every domain theory is a collection of feature definitions. Each feature definition is a parameterized theory stating when an instance of the feature should be associated with a specification. The theories capture various abstract views of the domain from which analogies can be made about component function. The theories represent the knowledge that the domain expert would use to classify the database by hand. Instead, the new role of the domain expert is to provide the feature definitions.

For example, within the DSP domain, the feature *Select* is defined as:

$$Select(S1, S2) \Leftrightarrow \exists x : S1, y : S2 \bullet isInput(x) \wedge isOutput(y) \wedge y \in x$$

Operationally, a feature's theory defines when it should be added to a specifications feature set. The two predicates, *isInput* and *isOutput*, are used with existential quantification to specify relationships between sorts in the domain and range. If there is a substitution such that these predicates hold, the remainder of the theory is instantiated and used as a proof goal. If the proof succeeds then the corresponding feature is added to the specification's feature set. Given the definition of *Select* above, a component having the feature $Select(intseq, int)$ indicates that the component has an input of type *int* that is a member of an input of the composite type *intseq*.

The set of features used to classify a reuse library is domain dependent. This allows abstractions to be made about the component functionality that occurs within the domain. These abstractions are captured in the feature definition theories and used for component classification. For example, a subset of the features that are currently defined for the digital signal processing domain are shown in Figure 1. In the digital signal processing domain, many objects are manipulated as sequences. This is reflected by the use of the sequence operators *length* and $\in$ in these feature definitions.

4.2 Feature Set Generation

The feature-based classification of a software component is derived from its specification using automated theorem proving. A domain theory that defines the system's sorts and operation is made available to the theorem prover as a set of rewrite rules. Given a feature definition, the domain and range of the specification are evaluated to see if a substitution of sorts exists to potentially satisfy the theory. For each feasible substitution the theory is instantiated with the variables of the appropriate sort. The theorem prover is then given the goal of showing that the input and output conditions imply the instantiated goal statements. If the goal is reached after a fixed number of forward inferencing steps, the associated feature is added to the feature set. Deeper information is discovered by selectively allowing the theorem prover to

$$
\begin{aligned}
Select(S1, S2) &\Leftrightarrow \exists x : S1, y : S2 \bullet isInput(x) \wedge isOutput(y) \\
&\quad \wedge y \in x \\
NonMember(S1, S2) &\Leftrightarrow \exists x : S1, y : S2 \bullet isInput(x) \wedge isOutput(y) \\
&\quad \wedge y \notin x \\
Build(S1, S2) &\Leftrightarrow \exists x, y.z : S1, y : S2 \bullet isInput(x) \wedge isOutput(y) \\
&\quad \wedge x = y.z \\
Permute(S1) &\Leftrightarrow \exists x, y : S1 \bullet isInput(x) \wedge isOutput(y) \\
&\quad \wedge \forall z : S \cdot z \in x \Leftrightarrow z \in y \\
Merge(S) &\Leftrightarrow \exists x, y, z : S \bullet isInput(x) \wedge isInput(y) \wedge isOutput(z) \\
&\quad \wedge \forall w : S \cdot w \in z \Rightarrow w \in x \vee w \in y \\
Route(S) &\Leftrightarrow \exists x, y : S \bullet isInput(x) \wedge isOutput(y) \\
&\quad \wedge x = y \\
Transform(S1, S2) &\Leftrightarrow \exists x : S1, y : S2 \bullet isInput(x) \wedge isOutput(y) \\
&\quad \wedge length(x) = length(y)
\end{aligned}
$$

Figure 1: Sample Signal Processing Features

perform more forward inferencing. Goals are generated in this manner for each feature definition in the system.

Consider the VSPEC specification of a search entity previously described:

```
ENTITY search IS
  PORT (input : IN recordArray;
        key : IN INTEGER;
        output : OUT recordType)
  REQUIRES sorted(input);
  ENSURES member(output, input)
          AND output.keyval = key;
  INCLUDES "record.lsl";
END search;
```

The *DRIO* form of this specification is:

$\forall$input : recordArray, key : integer
$\exists$output : recordType
sorted(input) $\Rightarrow$ member(output, input)
$\wedge$output.keyval = key

To construct a feature set for this entity the domain model axioms must be made available to the theorem prover. The domain model is defined in the LSL files that are listed in the **includes** clause. The domain theory axioms defined by the LSL are transformed into a set of rewrite rules and added to the rule base of the theorem prover. Each feature definition in the domain is then evaluated. Using the data processing features from Figure 1, the *isInput* and *isOutput* predicates for *Select* and *NonMember* are satisfied by the substitution (recordArray $\mapsto S1$, recordType $\mapsto S2$). In addition, the theory is type checked using the included LSL specification, `record.lsl`. In this case, the existence of the operator $\in$: recordType $\rightarrow$ recordArray is verified within the domain theory. After passing both of these checks, the remainder of each theory is instantiated with the appropriate variables and used as the consequent of a proof goal. Two goals are presented to the the theorem prover:

(sorted(input) $\wedge$ member(output, input) $\wedge$
output.keyval = key) $\Rightarrow$ output $\in$ input

(sorted(input) $\wedge$ member(output, input) $\wedge$
output.keyval = key) $\Rightarrow$ output $\notin$ input

Within the LSL domain theory, the **member** operator is defined in terms of $\in$. For the first feature, this operator definition will be expanded during forward inferencing and the goal will be satisfied. When this occurs, the feature (*Select*, recordType) is added to the component's feature set. For the second feature, the goal will not be resolved so the feature is not added to the feature set. Although *Select* and *NonMember* have opposite definitions the features are not mutually exclusive. It is possible for the features to coexist in a feature set via different variable substitutions.

Feature sets are generated and stored for all of the components in the reuse database. This classifica-

tion is done automatically using the method described above. One advantage to automatic classification is that the set of feature definitions can be changed and the database can be automatically re-classified. A new feature set is simply generated for each component in the reuse library. Informal classification schemes require a domain expert to reclassify each component in the database. In addition, the inference mechanism can be run longer when classifying database components because it occurs off-line before user interaction begins. Thus, features requiring more levels of forward inferencing may be discovered.

5 Retrieval

Given the feature set representations for a problem, we wish to retrieve a set of components having similar feature sets. There are two concepts used to compare features in feature sets: sharing and matching. Two components that contain a feature with the same attribute name are said to share the named feature. If the shared feature has equivalent values in both feature sets, it is a matching feature.

A shared feature indicates that some aspect of the functions performed by two components are analogous, but in terms of different domain sorts. The more features that two specifications share, the more similar their function is likely to be. Given the feature sets of two components, A and B to be A_{fs} and B_{fs}, respectively, this relationship is captured by their shared feature ratio:

$$S_f(A,B) = \frac{|\{f|\exists x,y \cdot (f,x) \in A_{fs} \wedge (f,y) \in B_{fs}\}|}{|\{f|\exists x,y \cdot (f,x) \in A_{fs} \vee (f,y) \in B_{fs}\}|}$$

A matching feature means that the components perform analogous functions over identical sorts. This attests to a stronger degree of functional similarity than a shared feature. The matching feature ratio is defined as:

$$M_f(A,B) = \frac{|\{(f,x)|\exists x \cdot (f,x) \in A_{fs} \wedge (f,x) \in B_{fs}\}|}{|\{(f,x)|\exists x,y \cdot (f,x) \in A_{fs} \vee (f,x) \in B_{fs}\}|}$$

The similarity of two specifications based upon their feature sets is the product of their shared feature ratio and their matching feature ratio.

$$Similarity(A,B) = S_f(A,B) \cdot M_f(A,B)$$

Similarity is used to find a small subset of reuse candidates from a large reuse library. The candidates then undergo verification to see if they satisfy the problem specification. By setting a similarity threshold and checking retrieving components whose similarity is higher, a set of potential solutions is found. By reducing the threshold, "riskier" solutions are considered. By setting the threshold to 1.0, only exact matches are found. The threshold value is varied based on the means available for adapting a potential solution to the current problem.

While specifications cannot generally be compared syntactically, features can. Thus, the similarity value of two specifications is determined much more efficiently than would be logical satisfaction or equivalence. Features for a problem are generated only once. By limiting the inference process, time required for this activity can be controlled by the user. Features for database elements are determined once and stored statically. Using selected features as static, database indices supports extremely efficient database retrieval requiring that full similarity be determined for only a subset of potential solutions.

6 Related Work

The retrieval and reuse of solutions in the manner described here is common to most case-based reasoning systems [18, 19]. The structure of this system is largely based on the retrieve-adapt-evaluate architecture of classical case-based reasoning systems. This paper deals primarily with the retrieval process. Additionally, the similarity metric is a simplification of the BENTON similarity metric [20] used to retrieve and compare design plans.

Jeng and Cheng [5, 6] use formal specifications to construct a hierarchical reuse library. The hierarchy is two-tiered; the lower lever is based on the notion of logical satisfaction while the upper level is based on specification syntax. Formal reasoning is used to retrieve components that are equivalent, more general, or more specific than a given query specification. Boudriga et al. [3, 21] support software reuse using a lattice of relational specifications. The mechanism for retrieving specifications from the lattice relies on theorem proving to compare specifications. There is no discussion of the effects of the theorem proving on the scalability of the methodology. In both of these systems a form of classification takes place when the component is placed into the reuse database structure. However, it is not apparent that this classification will sufficiently reduce the overhead of formal reasoning to provide scalability.

Zaremski and Wing [9, 10] apply signature and specification matching as a method of retrieving reusable components. Signature matching selects functions from the database that have similar input and output types as the desired function, allowing var-

ious sort substitutions. Specification matching then uses a *semi-automated* theorem prover to show various logical relationships (such as logical satisfaction) hold between the input and output conditions of the retrieved components and the query specification. This approach is similar to our own, in fact, the information used in signature matching can be represented in terms of semantic features. However, specification matching is a syntactic measure and does not take into account specification semantics.

The Inquire retrieval mechanism described by Perry and Popovich [7] supports retrieval based on specification predicates. A formal specification of a component's behavior is constructed from a set of predefined predicates. An inference mechanism is then used during the retrieval process to locate components that satisfy the various predicates. Predicates in the Inquire system are somewhat similar to semantic features. However, the set of predicates serves as the complete formal specification of a component. To provide consistency for retrieval, the predicates that can be used in specifications must be predefined for the problem domain. When using semantic features, only the feature definitions must be predefined. New abstract types (sorts and operators) can be created as long as they are related to the basic domain theory over which the features are defined. The authors do not discuss the scalability of predicated based retrieval when implemented with a fully descriptive specification logic.

Steigerwald [22] describes a method for automating the reuse of algebraic specifications. Normalized algebraic specifications are compared by constructing sample terms in one algebra and then mapped via a signature morphism into the second algebra. The results of reducing the sample terms are used to verify the consistency of the signature morphism. Steigerwald's work is also an attempt to make retrieval based on specifications affordable by minimizing the application of formal reasoning; semantic normalization is applied to the algebras to simplify the matching process. Similarly, limited theorem proving during the classification process leads to a more efficient component retrieval phase.

7 Conclusions and Future Work

The algorithm described here currently serves as the principle design methodology used by COMET [13] for transforming VSPEC problem specifications into VHDL implementations. It is implemented using REFINE [23] in the SOFTWARE REFINERY environment. The system currently retrieves both components and high level architectures for instantiation. The results are passed to hardware/software partitioning procedures and finally to hardware and software synthesis subsystems.

Developing pragmatic features and populating the reuse database are difficult tasks in any case-based reasoning domain. Feature sets for DSP components are currently being developed and implemented. Because DSP is a relatively well-defined, narrow domain, this process is substantially simpler than in broader domains. As VSPEC is increasingly used as a specification language, the reuse database will continue to grow. Because VSPEC use is independent of reuse activities and features can be automatically derived from it, any component specified using VSPEC can potentially be added to the reuse library.

The effectiveness and efficiency of retrieval mechanisms for software reuse are highly dependent upon the classification scheme used. Imprecise classification schemes provide a medium for comparing component similarities. However, more precise information is needed to guarantee reusability. Extremely precise classification, while guaranteeing retrieval of the component, complicates the comparison of component similarity.

This paper describes classification and retrieval of software components that bridges this gap. Components are classified using a set of semantic features, functional descriptors that are logically implied by the semantics of the component's formal specification. A feature set representation can be generated by a theorem prover using limited forward inference. Comparison of components via feature sets allows functional comparison without requiring formal reasoning. This methodology is scalable because of the limited set of components over which theorem proving is applied to determine reusability.

Acknowledgments

Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RAASP Technology Program, contract number F33615-93-C-1316.The authors would also like to thank the various anonymous referees who have provided many helpful comments during the development of this work.

References

[1] Eduardo Ostertag, James Hendler, Ruben Prieto Diaz, and Christine Braun. Computing Similarity in a Reuse Library System: An AI Based Approach. *ACM Transactions on Software En-*

gineering and Methodology, 1(3):205–228, July 1992.

[2] Ruben Prieto-Diaz and Peter Freeman. Classifying software for reuse. *IEEE Software*, 4(1):6–16, January 1987.

[3] Noureddine Boudriga, Ali Mili, and Roland Mittermeir. Semantic-based software retrieval to support rapid prototyping. *Structured Programming*, 13:109–127, 1992.

[4] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. LaSSIE: A Knowledge-Based Software Assistant. *Communications of the ACM*, 34(5):34–49, May 1991.

[5] Jun-Jang Jeng and Betty H. C. Cheng. Using Formal Methods to Construct a Software Library. In *Procedings of 4th European Software Engineering Conference, Lecture Notes in Computer Science*, volume 717, pages 397–417, September 1993.

[6] Jun-Jang Jeng and Betty H. C. Cheng. A Formal Approach to Using More General Components. In *Procedings of the 9th Knowledge-Based Software Engineering Conference*, pages 90–97, September 1994.

[7] Dewayne E. Perry and Steven S. Popovitch. Inquire: Predicate-Based Use and Reuse. In *Procedings of the 8th Knowledge-Based Software Engineering Conference*, pages 144–151, September 1993.

[8] Robert A. Steigerwald. *Reusable Software Component Retrieval Via Normalized Algebraic Specifications*. PhD thesis, Naval Postgraduate School, Monterey, Califorina, December 1991.

[9] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching, a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, April 1995.

[10] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. In *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.

[11] Douglas R. Smith. Top-down syntheses of divide and conquer algorithms. *Artificial Intelligence*, 27:43–96, 1985.

[12] D. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, Sept. 1990.

[13] R. Vemuri, H. Carter, and P. Alexander. Board and MCM Level Synthesis for Embedded Systems: The COMET Cosynthesis Environment. In *Proceedings of the First Annual RASSP Conference*, pages 124–133, Arlington, VA, August 15-18 1994.

[14] Institute of Electrical and Electronics Engineers, Inc., 345 East 47th St., New York, NY 10017. *VHDL Language Reference Manual*, 1994.

[15] Phillip Baraona, Perry Alexander, and John Penix. VSPEC: A declarative requirements specification language for VHDL. *Current Issuse in Electronic Modeling*, July 1995. to appear.

[16] Knowledge Based Software Engineering Laboratory, University of Cincinnati. *VSPEC Language Reference Manual*, 1994. In Preparation.

[17] J. Guttag and J. Horning. *Larch: Languages and tools for formal specification*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1993.

[18] Christopher K. Riesbeck and Roger C. Schank. *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Hillsdale, NJ., 1989.

[19] Roger C. Schank. *Dynamic Memory*. Cambridge University Press, Cambridge, 1982.

[20] Perry Alexander. Combining transformational and derivational analogy in Larch specification generation. In *Proceedings of The 6th International Conference on Software Engineering and Knowledge Engineering*, pages 131–138, Riga, Latvia, June 1995. Knowledge Systems Institute.

[21] Noureddine Boudriga, Fathi Elloumi, and Ali Mili. On the lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.

[22] Robert A. Steigerwald. Reusable component retrieval with formal specifications. In Larry Latour, Steve Philbrick, and Mark Stevens, editors, *Proceedings of the Fifth Annual Workshop on Software Reuse*, 1992.

[23] L. Abraido-Fandino. An overview of refine 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering*, Madrid, Spain, April 1987.