Technical Report

# A Slicing Approach for Parallel Component Adaptation

Brandon Morel and Perry Alexander

ITTC-FY2003-TR-29120-01

November 2002

Project Sponsor:
EDAptive

# A Slicing Approach for Parallel Component Adaptation

Brandon Morel and Perry Alexander
*Information and Telecommunications Technology Center*
*University of Kansas*
*2335 Irving Hill Rd*
*Lawrence, Kansas 66045*
{*morel, alex*}*@ittc.ku.edu*

## Abstract

*Software reuse has received considerable attention as a technique for aiding software designers. One mechanism for increasing the efficiency of reuse is component adaptation, particularly when designing large and intricate systems. Component adaptation composes reusable components in an architecture to produce a new system. Specification slicing is a method for decomposing complicated systems into manageable sub-components, where each sub-component captures an independent behavior of the system. This paper outlines specification slicing and its application for reuse using the parallel adaptation architecture.*

## 1. Introduction

Developing large and complex software systems is a difficult and challenging task. Software reuse has received significant attention as a technique for improving the quality and reliability of such systems. A number of specification-based reuse systems [6, 11] have been developed with attractive results.

For reuse to be effective, a designer must be able to find a component in a library that satisfies the requirements of a design. However, a designer can not always expect a library to contain a component that matches his/her specific needs, especially if the desired functionality is complex.

Penix [10] proposed *adaptation-based reuse*, a process for adapting existing components by placing them in an architecture to satisfy a system design. From a top-down perspective, a system design is decomposed into a collection of interconnected components. The collective properties of the interconnected components form the properties of the system. A specification-based retrieval engine can be used to retrieve existing components to be reused in the architectural design of the system.

This paper demonstrates how specification slicing can be used for parallel adaptation-based reuse. A parallel adaptation architecture is a parallel composition of independent components such that the collective behavior satisfies the behavior of a system. Specification slicing is used to decompose systems into independent subsets of disjoint specifications, where each specification performs a subset of the behavior of the system. The use of specification slicing for parallel adaptation will be shown to produce a solution to a design problem that otherwise could not be realized with traditional retrieval techniques.

In the next section we briefly present an overview of specification-based retrieval. This is followed by an introduction on adaptation architectures. Specification slicing is presented in section 4. An example of an adaptation architecture using specification slicing is shown in section 5. In section 6 we evaluate our technique. Section 7 discusses related work, followed by concluding remarks and future directions in section 8.

## 2. Specification-based Retrieval

A formal specification states the behavior of a component without stating the implementation details. Each component has a formal specification that interfaces the implementation of the component. Using formal specifications over implementations allow automated theorem-provers to verify logical relationships between two components.

A formal specification of a component follows the DRIO [13] model structure:

$$\forall x \in D, \exists y \in R | I(x) \Rightarrow O(x, y)$$

`D` and `R` are the domain and range respectively. `I` is a set of pre-conditions that define the *legal inputs* to the component. The pre-conditions constrain the domain to the values that have a defined output. `O` is a set of post-conditions that define the *feasible outputs* for each legal input based on the

```
facet comppad(
  inseq :: input sample_sequence;
  len :: input nat;
  leftpadseq :: output sample_sequence;
  rightpadseq :: output sample_sequence) ::
dspdomain is
begin
  pre:   len > 0;
  post1: leftpadseq' =
           append(zeroseq(len), inseq);
  post2: rightpadseq' =
           append(inseq, zeroseq(len));
end facet comppad;
```

**Figure 1. Rosetta specification of a zero padding component**

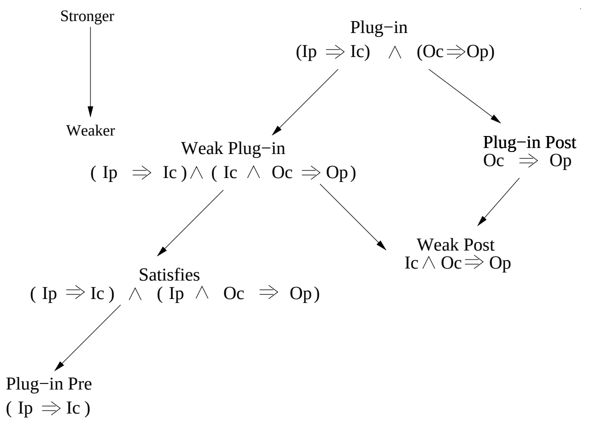


**Figure 2. Lattice of specification match conditions**

D×R relation. If the pre-conditions hold then the component must end in a state such that the post-conditions are true.

The DRIO component models are written in Rosetta [1]. Rosetta is a systems level design language for modeling heterogeneous systems. A Rosetta facet describes the requirements and behavior of a particular aspect of a system. Rosetta is a vast language, but only a subset is needed to describe a DRIO model. Facet parameters declare the domain and range of components. Facet terms define pre- and post-conditions over the domain and range.

Figure 1 specifies a component using Rosetta. The input and output types define the domain and range respectively. The pre-condition `pre` defines legal inputs to the component; the post-conditions `post1` and `post2` define the valid outputs of the component, forming:

$$I_{comppad} = pre$$
$$O_{comppad} = post1 \wedge post2$$

Several specification-based retrieval engines [6, 11] have been developed using automated theorem-provers to verify that a component specification matches a problem specification. Zaremski and Wing [16] established a number of match conditions for assessing specification reuse. Figure 2 shows a portion of these match conditions.

A problem specification is a DRIO specification that specifies the behavior of a system yet to be implemented. A library exists that contains component specifications. A specification-based retrieval engine formally verifies that a match condition exists, if any, between the problem specification and a component specification.

If a component C formally *satisfies* a problem P, then the implementation of C can be reused to implement P. C satisfies P if C accepts all legal inputs to P and the valid outputs of C must be valid outputs of P, given the legal inputs. *Weak plug-in* and *plug-in* are stronger match conditions of *satisfies*. The *plug-in pre* match condition implies that a component meets only the pre-condition requirements. The *plug-in post* and *weak post* match conditions imply that a component meets only the post-condition requirements.

## 3. Adaptation Architectures

It is naive to assume that a component will exist in a library that satisfies a large and complex problem. It is more feasible to retrieve a component that has a subset of the properties of the problem. The component can be adapted to obtain the properties of the problem by placing the component in an architecture with other components. An architecture is simply a collection of interconnected components.

Penix [10] identifies several behavioral adaptation tactics. Adaptation tactics are based on the match conditions of the components available for reuse. Each adaptation tactic applies an architecture construction method to the components. Adaptation continues until the problem is satisfied or the solution can not be realized.

In a parallel adaptation tactic, two or more components are composed in parallel. Figure 3 shows a problem specification and the parallel architecture to satisfy the problem. The problem has independent post-conditions, denoted as O$_{P1}$ and O$_{P2}$ where $O_P = O_{P1} \wedge O_{P2}$.

In a bottom-up approach, components are first retrieved from the library and the architecture is evaluated based on the component's behavior. Assume component A is retrieved, which only satisfies one of the post-conditions, O$_{P1}$. Component A can be adapted by placing it in a parallel architecture with another component (to be retrieved, namely component B) which satisfies the missing behavior.

In a top-down approach, the problem is decomposed into independent sub-problems, i.e. $I \rightarrow O_{P1}$ and $I \rightarrow O_{P2}$.

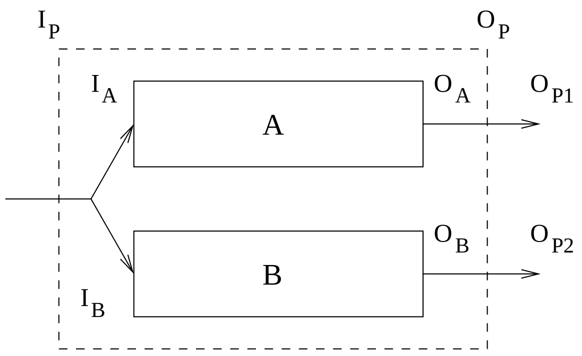


**Figure 3. A parallel adaptation architecture**

Retrieval is performed on the sub-problems. The matching components are composed in a parallel architecture.

## 4. Specification Slicing

Program slicing [14] is a decomposition process used to isolate a subset of program behavior. A program slice is a sub-program that contains only those statements and variables that affect or are affected by a slicing criterion. A slice criterion is a set of variables that are of interest at some point in the program. Program slicing has generated a breadth of applications at the implementation level of software design, including debugging [15], maintenance [7], and reuse [3].

Program slicing was applied at the specification level by Oda and Araki [9]. They define a technique for slicing Z specifications. A slice contains a portion of the statements in the specification that constrain the value of a variable. We apply specification slicing to the DRIO models written in Rosetta. The goal will be to use specification slicing to decompose a problem specification by isolating the independent behaviors; a retrieval engine will be used to locate components that satisfy the slices.

A specification is written as a tuple (D, R, I, O) representing the sets of domain variables, range variables, pre-conditions, and post-conditions. The specification in figure 1 can be represented as the tuple ({inseq, len}, {leftpadseq, rightpadseq}, {pre}, {post1, post2}).

If a term p potentially affects term q, then term q is data dependent on p. Post-condition terms are data dependent on other post-conditions if they both constrain a range variable. Similarly, pre-condition terms are data dependent on other pre-conditions if they both constrain a domain variable. The functions for data dependency are defined:

$$dDepend(q,p:O):bool = \exists r \in R | constrains(q,r) \wedge constrains(p,r)$$

$$dDepend(q,p:I):bool = \exists d \in D | constrains(q,d) \wedge constrains(p,d)$$

```
$INIT : R_s \leftarrow criterion,$
$O_s \leftarrow \{o | \forall o \in O, \exists r \in R_s, constrains(o,r)\}$
$D_s \leftarrow \emptyset, I_s \leftarrow \emptyset,$
$1. O'_s \leftarrow O_s \cup \{o | \forall o \in O, \exists x \in O_s, dDepend(o,x)\}$
$2. Repeat\ step\ 1\ until\ O'_s = O_s$
$3. I'_s \leftarrow I_s \cup \{i | \forall i \in I, \exists x \in O_s, cDepend(x,i)\}$
$4. I'_s \leftarrow I_s \cup \{i | \forall i \in I, \exists y \in I_s, dDepend(i,y)\}$
$5. Repeat\ step\ 4\ until\ I'_s = I_s$
$6. D'_s \leftarrow D_s \cup \{d | \forall d \in D, \exists o \in O_s, requires(o,d)\}$
$7. D'_s \leftarrow D_s \cup \{d | \forall d \in D, \exists i \in I_s, constrains(i,d)\}$
$8. D'_s \leftarrow D_s \cup \{r | \forall r \in R, \exists o \in O_s, requires(o,r)\}$
$9. R'_s \leftarrow R_s \cup \{r | \forall r \in R, \exists o \in O_s, constrains(o,r)\}$
```

**Figure 4. Rosetta specification slicing algorithm**

If a term p potentially determines if term q is applied, then term q is control dependent on p. In a DRIO model, the pre-conditions control the application of the post-conditions. A post-condition is control dependent on a pre-condition if the pre-condition constrains the legal inputs required to compute the feasible outputs. The function for control dependency is defined:

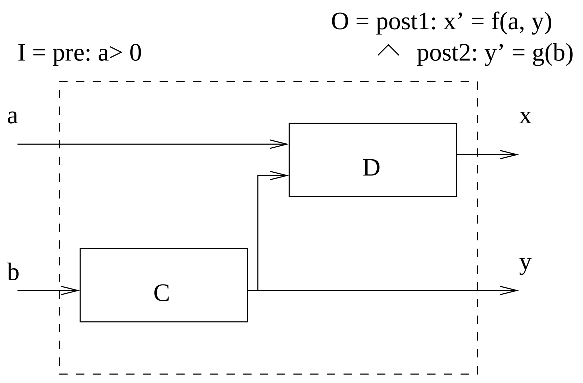
$$cDepend(o:O, i:I):bool = \exists d \in D | constrains(i,d) \wedge requires(o,d)$$

A specification slice is represented by a tuple ($D_s$, $R_s$, $I_s$, $O_s$). The slice criterion is a set of range variables. A specification slice is computed with the algorithm in figure 4. Initially, the post-conditions that affect or are affected by the criterion are assigned to $O_s$, then all post-conditions that are data dependent on post-conditions in $O_s$ are added in steps 1 and 2. Next all the pre-conditions that potentially determine if the post-conditions in $O_s$ are applied are added to $I_s$ in step 3. Then all pre-conditions that affect or are affected by the pre-conditions in $I_s$ are added to $I_s$ in steps 4 and 5. Finally, the domain and range variables involved in $O_s$ and $I_s$ are selected in steps 6 through 9.

$R_s$ represents all the range variables that affect or are affected by the criterion. If the specification was re-sliced with $criterion' \leftarrow R_s$, the same slice would be obtained. A slice may contain the original specification or an empty specification. An empty specification has no pre-conditions or post-conditions, i.e. pre: true and post: true.

A specification may require the value of a range variable but does not constrain the range variable (step 8). Figure 5 shows an example of such a case.

**Definition 4.1** *A "criterion partition" is the disjoint subsets of the range variables such that*
*1) the union of the subsets equals the range*



| Criterion | Slice |
|---|---|
| {x} | ({a, y}, {x}, {pre}, {post1}) |
| {y} | ({b}, {y}, {}, {post2}) |

**Figure 5. An slice with a range variable as input**

*2) a subset is not empty*
*3) a subset does not affect or is not affected by another subset, meaning the subsets are all independent*

A criterion partition is the same as a traditional partition of a set if all the range variables (R) are independent. A criterion partition generates a slice partition.

**Definition 4.2** *A "slice partition" is the partition of slices generated by a criterion partition. A slice partition must also be disjoint since a criterion partition is disjoint, meaning each slice is independent of all other slices in the set (one slice does not influence and is not influenced by another slice).*

A criterion may contain several range variables. If the variables are all interdependent, then the criterion is referred to as the smallest criterion.

**Definition 4.3** *The "smallest criterion partition" contains the disjoint subsets of smallest criterion.*

In terms of the *Stirling number of the second kind* [5], the smallest criterion partition corresponds to S(n, n), where S(n, k) states the number of partitions of an n-set into k blocks. Here "n" is the number of disjoint smallest criterion.

The algorithm in figure 6 is used to generate the smallest criterion partition. $C_F$ represents the range variables. A single range variable is selected from $C_F$ and is used as a slicing criterion. The slicing algorithm is used to generate $R_s$, which represents the smallest criterion that generates that particular slice. By removing $R_s$ from $C_F$ and repeating, the smallest criterion partition, $C_{scp}$, can be obtained.

```
$INIT : C_F \leftarrow R,$
$criterion \leftarrow \emptyset$
$C_{scp} \leftarrow \emptyset, S_{ssp} \leftarrow \emptyset,$
$1.criterion \leftarrow \{r | \exists r \in C_F\}$
$2.(D_s, R_s, I_s, O_s) \leftarrow slice(criterion)$
$3.S'_{ssp} \leftarrow S_{ssp} \cup \{(D_s, R_s, I_s, O_s)\}$
$4.C'_{scp} \leftarrow C_{scp} \cup \{R_s\}$
$5.C'_F \leftarrow C_F - R_s$
$6.Repeat\ step\ 1\ until\ C_F = \emptyset$
```

**Figure 6. Smallest criterion partition/smallest slice partition algorithm**

The algorithm is also used to generate the smallest slice partition, $S_{ssp}$.

**Definition 4.4** *The "smallest slice partition" contains the disjoint subsets of smallest specification slices generated from the smallest criterion partition.*

Each slice in the smallest slice partition represents the smallest independent behavior of the specification, referred to as the smallest independent slice.

**Definition 4.5** *The "smallest independent slice" is a specification slice that can not be further sliced. The criteria variables for generating the smallest independent slice are interdependent.*

Each slice specification in a partition is submitted to a retrieval engine. If a matching component is found for each slice, then the partition represents the collection of components to connect in parallel.

Component specifications in the library are not guaranteed to be in the smallest independent form, thus the smallest slice partition does not guarantee a complete solution. Therefore every possible slice partition must be generated (generated by every possible combination of the smallest criterion). The number of partitions of an n-set follow the *Bell numbers, $b_n$* [2, 12], which increases exponentially. If none of the slice partitions generate a complete solution, then there does not exist a parallel composition architecture to solve the problem.

Specification slicing can be used to automate the top-down parallel adaptation architecture by slicing the independent behavior of the problem. Each slice represents an independent behavior; the parallel composition of the slices satisfies the behavior of the problem. The interconnections of the components matching the slice specifications is a variable mapping from the problem to the components.

Specification slicing has applications in reuse-of-the-large and reuse-of-the-small. Figure 3 shows a large prob-

```
facet cross_equal(
  R :: input signal;
  L :: input signal;
  hileft :: output signal;
  hiright :: output signal;
  midleft :: output signal;
  midright :: output signal;
  locenter :: output signal) :: analog is
begin
  pre:  true;
  post1: hileft’ = hpfilter(delay(L));
  post2: hiright’ = hpfilter(delay(R));
  post3: midleft’ =
           hpfilter(lpfilter(equalize(L)));
  post4: midright’ =
           hpfilter(lpfilter(equalize(R)));
  post5: locenter’ =
           lpfilter(equalize(mix(L, R)));
end facet cross_equal;
```

**Figure 7. Rosetta specification of a 3-way crossover/equalizer**

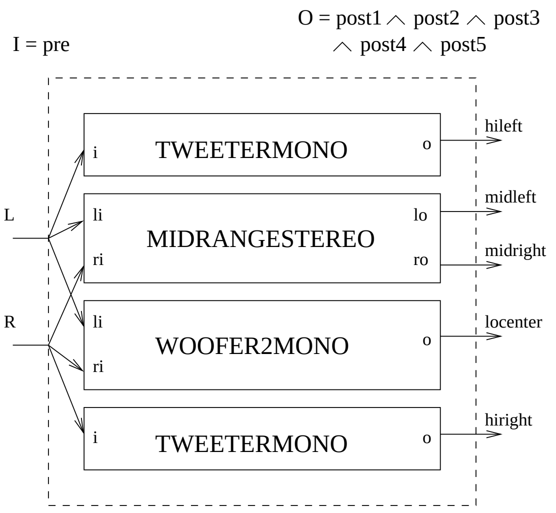
lem decomposed and smaller components reused. Conversely, a large architectural component can be decomposed into its reusable sub-components.

## 5. Example

In this section we present an example to illustrate our slicing algorithm for the parallel adaptation architecture. Figure 7 shows a problem specification for a 3-way crossover/equalizer [8] system. Figure 9 contains the library of available components. Clearly there does not exist a single component in the library to satisfy the problem. It will be shown that a collection of components can be retrieved and composed in a parallel architecture to satisfy the problem.

The algorithm in figure 6 is used to generate the smallest criterion partition and the smallest slice partition. The smallest criterion partition is ({hileft}, {hiright}, {midleft}, {midright}, {locenter}). The slice partition is shown in figure 8. The set of smallest criterion contains five elements. The Bell number, $b_n$, for n = 5 is 52, meaning there are 52 possible partitions. In the worst case 52 different slice partitions have to be tested for retrieval and adaptation, fortunately the number of unique specification slices to undergo retrieval is equal to the size of the power set of the smallest slice partition, or $2^n$-1 = $2^5$-1 = 31 specifications.

Retrieval is performed on the smallest slice partition. Components for only three of the specifications can be reused. Instead of blindly searching through all possible slice partitions, heuristics can be applied to test only unique



| | Criteria | Slice | Match |
|---|---|---|---|
| smallest criterion partition | {locenter} | {{R, L}, {locenter}, {pre}, {post5}} | woofer2mono |
| | {hileft} | {{L}, {hileft}, {pre}, {post1}} | tweetermono |
| | {hiright} | {{R}, {hiright}, {pre}, {post2}} | tweetermono |
| | {midleft} | {{L}, {midleft}, {pre}, {post3}} | nil |
| | {midright} | {{R}, {midright}, {pre}, {post4}} | nil |
| | {midleft, midright} | {{R, L}, {midright, midleft}, {pre}, {post3, post4}} | midrangestereo |

**Figure 8. Crossover/equalizer parallel architecture**

```
facet tweetermono(
  i :: input signal;
  o :: output signal) :: analog is
begin
  pre:  true;
  post: o′ = hpfilter(delay(i));
end facet tweetermono;

facet tweeterstereo(
  li :: input signal;
  ri :: input signal;
  lo :: output signal;
  ro :: output signal) :: analog is
begin
  pre:  true;
  post1: lo′ = hpfilter(delay(li));
  post2: ro′ = hpfilter(delay(ri));
end facet tweeterstereo;

facet woofermono(
  i :: input signal;
  o :: output signal) :: analog is
begin
  pre:  true;
  post: o′ = lpfilter(equalize(i));
end facet woofermono;
```

```
facet woofer2mono(
  li :: input signal;
  ri :: input signal;
  o :: output signal) :: analog is
begin
  pre:  true;
  post: o′ = lpfilter(equalize(mix(li, r)));
end facet woofer2mono;

facet midrangestereo(
  li :: input signal;
  ri :: input signal;
  lo :: output signal;
  ro :: output signal) :: analog is
begin
  pre:  true;
  post1: lo′ =
          hpfilter(lpfilter(equalize(li)));
  post2: ro′ =
          hpfilter(lpfilter(equalize(ri)));
end facet midrangestereo;
```

**Figure 9. Analog component library**

partitions. Clearly the slice partition ({hileft}, {hiright}, {midleft, midright}, {locenter}) should be tested next since neither {midleft} nor {midright} had matching components. This slice partition does indeed generate a parallel composition architecture to satisfy the problem. The parallel architecture is composed in figure 8.

## 6. Evaluation

The largest gain from our slicing technique is the ability to increase reuse using an adaptation architecture. The example in the last section showed that the traditional single-component-reuse did not provide a solution, whereas our technique provided a solution by reusing a collection of components.

Our slicing technique can be easily integrated into existing specification-based reuse systems. After a problem specification is sliced, each specification slice is given to the retrieval engine. The results are then used to construct the architecture.

Specification slicing also gives designers all the benefits of program slicing at the specification level. Specification slicing allows designers to track dependencies [9] and perform testing and debugging activities [4].

The drawback to our slicing approach is the verbose generation of specification slice partitions. As the number of smallest criterion increase, the number of slice partitions grow exponentially. In a significantly large problem specification the sheer number of specification slices generated may drowned the retrieval engine from finding a solution in a reasonable amount of time. Different heuristics are being explored to avoid searching every possible slice partition.

## 7. Related Work

Penix [10] presents a framework for adaptation tactics using architectures. In his work, problem and component specifications are assigned domain features using a classification technique. A component for reuse that has some of the features of the problem is adapted with other components with the missing features in a parallel architecture. The collective features of the components in the parallel architecture matches all the features of the problem. This assumes the features are always fine grained. For example, a highpass filter and a lowpass filter would both have the feature `filter`, but the two filters are vastly different.

Oda and Araki [9] developed specification slicing and was further explored by Chang and Richardson [4]. Chang and Richardson developed methods for static and dynamic slicing of Z specifications. They applied specification slicing toward testing, validation, and debugging. We apply static specification slicing for reuse.

Zhao [17] developed a slicing approach for architecture description specifications for reuse-of-the-large. A large description of a software system is described in an ADL using a collection of elements (components and connectors); slicing is used to extract and reuse elements or collections of elements in other system designs.

## 8. Conclusions and Future Work

Software reuse is an invaluable technique for developing software systems. For reuse to be profitable, architectures must be constructed using smaller components to solve larger and more intricate problems.

In this paper, specification slicing was implemented as a method for parallel component composition. The slicing approach isolates the independent behaviors of a problem, subsequently the slices increase the chances of reuse in a library containing basic components. An example showed a problem that could not be solved using one component, rather a solution was achieved using a parallel adaptation architecture.

The slicing method for parallel composition has been implemented in a system called SPARTACAS. SPARTACAS uses several adaptation architectures, in addition to the parallel architecture, and has been successfully applied to solve a digital down-converter problem. Reuse metrics, such as time, recall, and precision, are currently being investigated.

## References

[1] P. Alexander, D. Barton, and C. Kong. *Rosetta Usage Guide*. The University of Kansas / ITTC, 2335 Irving Hill Rd, Lawrence, KS, 2000.

[2] E. T. Bell. Exponential numbers. *American Math Monthly*, 41:411–419, 1934.

[3] G. Canfora, A. Cimitile, A. D. Lucia, and G. D. Lucca. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance*, pages 424–433, Victoria, Canada, 1994.

[4] J. Chang and D. J. Richardson. Static and dynamic specification slicing. In *Proceedings of the Fourth Irvine Software Symposium*, Irvine, CA, April 1994.

[5] J. H. Conway and R. K. Guy. *The Book of Numbers*. Springer-Verlag, New York, 1996.

[6] B. Fischer and J. Schumann. SETHEO goes software engineering: Application of ATP to software reuse. In *Proc. CADE-14*, July 1997.

[7] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *IEEE Transaction on Software Engineering*, 17(8):751–761, 1991.

[8] S. Linkwitz. A three-enclosure loudspeaker system with active delay and crossover: Part 2. Speaker Builder, March 1980.

[9] T. Oda and K. Araki. Specification slicing in formal methods of software development. In *Proceedings of the 17th Annual International Computer Software and Applications Conference*, pages 313–319. IEEE Computer Society Press, November 1993.

[10] J. Penix. Rebound: A framework for automated component adaptation. In *Proceedings of the 9th Annual Workshop on Software Reuse*, January 1999.

[11] J. Penix, P. Baraona, and P. Alexander. Classification and retrieval of reusable components using semantic features. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 131–138, Nov. 1995.

[12] G. C. Rota. The number of partitions of a set. *American Math Monthly*, 71:498–504, 1964.

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

[14] M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[15] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[16] A. M. Zaremski and J. M. Wing. Specification matching of software components. In *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Oct. 1995.

[17] J. Zhao. A slicing-based approach to extracting reusable software architectures. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering*, pages 215–223, 2000.