

Retrieval as Synthesis: Feature-Based Retrieval and Adaptation Architectures

Perry Alexander, Cindy Kong and Brandon Morel

ITTC – The University of Kansas
Nichols Hall
2335 Irving Hill Rd
Lawrence, Kansas 66045
{alex,ckong,morel}@ittc.ku.edu

Abstract

One mechanism for achieving efficient component synthesis is retrieval and adaptation of existing solutions and architectures. This paper outlines two techniques, feature-based retrieval and adaptation architectures, that support this synthesis technique. Feature-based retrieval is a case-based reasoning derivative used to efficiently retrieve potential solutions from a component database. Adaptation architectures are small architectures used to efficiently package components for reused in a black-box fashion. Together these techniques provide an effective, rigorous synthesis technique for component-based systems.

Introduction

If we assume that software system construction parallels the construction of traditional systems, retrieving, adapting and reusing components in standard configurations may help address substantial numbers of design problems. Given a problem specification and a collection of components and system architectures, it should be possible to reuse both existing components and problem solving architectures. Additionally, high assurance of correctness in the resulting system can be obtained if the retrieval and instantiation of components can be achieved using formal, sound techniques. We propose that combining specification matching and adaptation architectures can be used to achieve such an end.

Specification matching enhanced by *feature-based retrieval* provides a mechanism for classifying and retrieving components in a rigorous manner. Ontologies for classifying components are constructed using formally defined features that are derived from component specifications. These features are used to select potential matches for consideration by a specification matching engine from a component database. By eliminating mismatches prior to specification matching, feature-based retrieval can substantially reduce the amount of time necessary to search a component database.

Adaptation architectures are special purpose architectures that allow a component to be reused by building a reuse

infrastructure around a retrieved component. The adaptation architecture situates the component in a subsystem that can be customized in well defined ways to solve a problem. By associating adaptation architectures with differences in desired and achieved specification matches, components that do not completely match a problem can be adapted and reused. Adaptation architectures provide quick, standard mechanisms for adapting existing components to new problems.

Feature-Based Retrieval

Formal component retrieval centers on establishing a relationship between a problem specification and a component that assures the component will be useful in addressing the problem. Because most components are implemented using operational representations that are not suitable for formal analysis, this relationship is typically established between a problem and a formal specification of a component. Appropriately, this process is frequently referred to as *specification matching*.

Abstractly, specification matching is implemented by: (i) defining a problem specification (s_p); (ii) defining a collection of component specifications (S_c); and (iii) searching for elements of S_c that satisfies a Boolean matching condition ($M(s_p, s_c)$). Thus, the problem becomes finding the set $\{s_c \in S_c \mid M(s_p, s_c)\}$, the of specifications from S_c that satisfy the match criteria. Although appealingly elegant, implementing specification matching in a brute force fashion is computationally impractical. Decision procedures must be designed that attempt to establish the match criteria between components. As typical components involve higher-order logic specifications, such decision procedures typically involve computationally expensive formal inference. The problem is further complicated because inference tends to be most expensive when the match condition cannot be proven. In a component database of reasonable size, failed cases will dominate resource consumption during the establishment of the match criteria.

Researchers have proposed several techniques to address the complexity of specification matching. For example, Penix and Alexander (Penix and Alexander 1999) have proposed *necessary filters* in the form of feature-based retrieval that removes the majority of components that cannot match the problem from the component database. Fischer (Fischer and Schumann 1997) has proposed the use of *competing matching engines* to allow simultaneous application of multiple property verification approaches.

The underlying principle behind feature-based retrieval is the use of traditional, indexing techniques to apply necessary conditions as a pre-filter for specification matching (Pearl 1984, Smith 1990). Given a collection of search spaces and a search goal, a sufficient condition identifies subspaces where a solution will be found, if one exists. Mathematically, if M is a match condition and ψ is sufficient condition, then we know $\psi \Rightarrow M$ and if ψ holds, M must hold. A necessary condition indicates search paths that cannot yield a solution by specifying a weaker condition than the match condition. Mathematically, if M is a match condition and ϕ is a necessary condition, then $M \Rightarrow \phi$. If ϕ holds nothing can be inferred about M , but if ϕ does not hold, M cannot hold. The necessary condition does not identify the subspace that should be searched, but does avoid subspaces that cannot yield solutions. An excellent example of a necessary condition check occurs when specifications or programs are checked for interface compatibility. If the interface of a component is compatible with a specification, there is no guarantee of the suitability of the component. However, if the interface of the component is not compatible with the specification, it can be guaranteed that the component will not satisfy the specification, negating the need for additional checking.

Feature-based retrieval defines a collection of necessary conditions that filter the component database. Unfortunately it is not always possible to structure a search space in a manner that supports defining sufficient conditions. Because the time complexity of specification matching is dominated by failed attempts, eliminating non-matching components has the potential to make specification matching a feasible component retrieval strategy. This result has been demonstrated empirically by preliminary work by the authors (Penix and Alexander 1999, Patil and Alexander 2000) and others (Fischer and Schumann 1997).

The implementation of necessary conditions as features is best understood by examining a commonly used matching criterion for software components called *satisfies match*. Given a problem specification s_p and a component specification s_c defined in the classical axiomatic style with preconditions i_p and i_c and post-conditions o_p and o_c respectively, we say that s_c *satisfies* s_p if the following condition holds:

$$s_c \text{ satisfies } s_p \equiv (i_p \Rightarrow i_c) \wedge (i_c \wedge o_c \Rightarrow o_p)$$

The *satisfies* condition is true when: (i) any legal input to the problem p is also a legal input to component c as defined by their respective preconditions ($i_p \Rightarrow i_c$); and (ii) when any legal output from c also a legal output from p as defined by their postconditions ($i_c \wedge o_c \Rightarrow o_p$).

The structure of *satisfies* is a conjunction of implications, each of which will have associated necessary conditions. If $i_p \Rightarrow i_c$ is being checked and ϕ is a necessary condition for the match, then $(i_p \Rightarrow i_c) \Rightarrow \phi$ must hold. By transitivity, $i_p \Rightarrow \phi$ must also be true if the condition is true. When checking for $i_p \Rightarrow i_c$ knowing that ϕ is implied i_c , c can be discarded if $i_p \Rightarrow \phi$ does not hold as the implication *cannot* hold because transitivity is violated. The necessary condition ϕ is called a *feature* because it represents a property that is exhibited by the problem and/or components. When checking the condition $i_p \Rightarrow i_c$ if the set of features associated with i_c is not a subset of the features associated with i_p then the match condition cannot be satisfied. The same argument follows for features associated with post-conditions and features associated with components in general.

Necessary condition filtering can significantly reduce the number of components involved in specification matching. However, efficiency is gained only when the time required to derive and compare features is less than the time required to attempt matching over components filtered by necessary conditions. Features associated with necessary conditions must be derived using inference techniques, putting at risk any gains from eliminating components from the search. Fortunately, features defined for *satisfies* depend only on one specification involved in the match. Features associated with components can be derived prior to component retrieval and used to index components in a traditional database. Features associated with the problem must of be derived at retrieval time, but represent only a fraction of the total inference requirements. (i) calculating problem specification features; (ii) retrieving components with matching features; and (iii) performing specification matching over the collection of retrieved components. Early prototypes indicate a significant gain in efficiency with little loss of precision or recall in the retrieval process (Penix and Alexander 1999, Patil and Alexander 2000).

Ongoing feature-based retrieval research includes: (i) exploration of new matching criteria; (ii) development of ontologies for component classification; and (iii) the investigation of efficient feature derivation and specification matching techniques. Current feature-based retrieval prototypes implement a limited collection of match criteria. Although *satisfies* is an exceptionally powerful criterion, other potential conditions must be explored to support adaptation, partial matches and the inclusion of heterogeneous components. The special interaction of feature derivation and the *satisfies* match condition must be extended to assure efficiency in other retrieval metrics.

The quality of ontologies used to classify components can profoundly affect retrieval efficiency, precision and recall. The collection of feature definitions used to filter components prior to retrieval defines an ontology for those components. Existing retrieval prototypes define primitive, *ad hoc* ontologies. Systematic methods for defining features and classifying components must be developed and empirically evaluated for precision and recall.

Specification matching efficiency is highly dependent on the quality of feature derivation and specification matching inference systems. Our current prototypes exclusively use the PVS specification and verification system for performing inference. Other inference tools including resolution-based provers, model checkers, equivalence checkers, and SAT algorithms must be explored in the context of feature-based retrieval.

Configuration and Adaptation Techniques

Successful component retrieval and reuse involves two tasks: (i) finding potential candidates; and (ii) adapting those candidates to solve the current problem. In the ideal case, every problem has an associated component in the component database. In practice, this is rarely the case due to the inability to predict the needs associated with new problems and the sheer size and complexity of such component databases. Retrieved components must be adapted by configuring parameters, instantiating and configuring architectures, and adapting component implementations. The proposed adaptation and configuration techniques will reuse “close” matches using differences between the desired and achieved match to guide configuration and adaptation processes.

Component adaptation techniques can be classified into two broad categories borrowed from software testing: (i) white box; and (ii) black box reuse. White box adaptation examines and attempts to alter the implementation of a component with the goal of achieving a different task. White box adaptation ranges in complexity from simple parametric adaptation through defining and setting parameters to altering code in software components or implementation in hardware components. Black box adaptation attempts to reuse a component without modification or knowledge of component implementation by building infrastructure around the component. Black box adaptation ranges in complexity from simple data conversion to elaborate harnesses or environment emulators.

Ongoing component adaptation research is investigating: (i) parameterization; (ii) architecture instantiation; and (iii) component adaptation through instantiation of adaptation architectures. Parameterization involves configuring predefined parameters to customize a retrieved component. Architecture instantiation is a form of parameterization where parameters represent components. To instantiate a parameter, the retrieval system is invoked to retrieve a component for the specific parameter based on its defined task within the architectures. Special cases of traditional architectures, adaptation architectures are special purpose architectures that situate a component in a usage environment. By instantiating other components in the adaptation architecture, the retrieved component is reused without structural modification. Differences between the desired match and the match achieved by the retrieved component will be used to: (i) specify parameter values; (ii) instantiate architecture components; and (iii) select and instantiate adaptation architectures.

Current component retrieval prototypes based on use of feature-based retrieval (Penix and Alexander 1999) and specification matching (Fischer and Schumann 1997, Zaremski and Wing 1995) use satisfies or plug-in for both feature based retrieval and specification matching. Moreover, these prototypes require an exact match between components and problems. In practice, many different matching criteria exist with satisfies representing only a single criteria for reuse. Figure 1 shows a matrix of various different specification matching criteria identified and classified by Zaremski and Wing (Zaremski and Wing 1995) and modified by Penix and Alexander (Penix and Alexander 1999). These matching criteria are organized in a lattice where each arrow represents implication. For example, achieving satisfies match implies that a plug-in pre match is also achieved. Moving down the lattice, matching criteria become increasingly weak and represent decreasingly close matches.

Parametric configuration is a process of adjusting known parameters to modify the behavior of a component. A parameterized component in the match hierarchy represents a family of behaviors resulting from specific component configurations. When a parameterized component is retrieved, the configuration system must utilize techniques to determine appropriate settings for adaptation parameters. Effectively, the configuration system generates the desired component from the retrieved component by instantiating adaptation parameters.

Architecture instantiation is a process of selecting values for components to form an aggregate problem solving systems. An architecture (Shaw and Garlan 1996) is an aggregation of component requirements that decomposes a problem into subsystems. Architectures describe each included component as well as interface requirements, interconnection requirements and how properties are calculated for the aggregate system. Architectures are populated by recursively retrieving components to instantiate architecture components utilizing requirements specified in the architecture. Configuration generates a collection of retrieval problems associated with the collection of components required to instantiate the architecture.

Adaptation architectures are small, special purpose architectures used to adjust the behavior of a component to achieve specific tasks. They may generate new retrieval problems, or they may use simple synthesis techniques to generate new components. To utilize adaptation architectures, the configuration system must know how the desired component and the retrieved component differ. The relative positions of the desired match and the achieved match in the specification lattice provide information about the difference that can be used by the configuration process. By understanding where the weaker match lies in the lattice with respect to the desired match, differences between the retrieved and desired components are defined. Adaptation architectures are associated with paths through the lattice. Thus, by knowing the path in the lattice between the desired and

achieved match, an adaptation architecture can be found and instantiated.

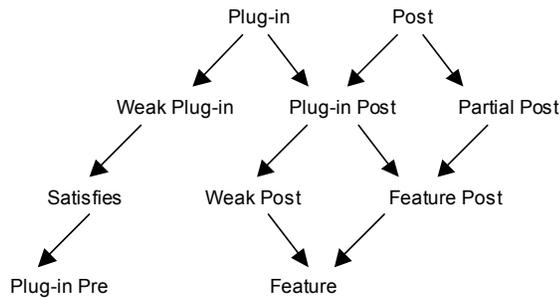


Figure 1 – Lattice of Specification Matching criteria (Zaremski and Wing 1995, Penix and Alexander 1999)

As an example of an adaptation architecture, consider the case where satisfies is the desired match result and the obtained match is weak-post. Note that weak-post is simply satisfies match with out the conjunct involving preconditions – only the postconditions are required to be compatible. Weak-post indicates that the retrieved component produces the correct output, but its precondition is violated by potential inputs to the desired component. The standard means for solving this problem is to design a converter that transforms problem inputs into suitable inputs for the retrieved component. Thus, the adaptation architecture associated with the link from weak post to satisfies is a two component batch-sequential architecture similar to the specification fragment in Figure 2. In this architecture, the retrieved component generates outputs and is used as the second component in the flow. The first component converts the input data to a form satisfying the second component’s precondition. A concrete example is using binary search as a general search technique. Binary search does not match using satisfies, but does using weak post because binary search requires a sorted input and in the general case, this cannot be guaranteed. Thus, a batch sequential architecture is used and the configuration system attempts to find a sorting component to place in front of the binary search component.

To implement reuse architecture use, we will identify standard mechanisms for addressing differences in retrieved and desired components. Each arc in the matching criteria lattice will be associated with an adaptation architecture for addressing differences in matching criteria. When a component is retrieved using a weaker match criterion, this architecture is then employed to adapt the retrieved component. We will identify useful matching criteria; place them appropriately in the lattice and associate reuse architectures with each lattice arc. Within the adaptation architecture, necessary components will either be retrieved in the same manner as traditional component or automatically generated using code synthesis techniques.

```

facet batch_seq(T::design type; x::input T;
                z::output T; f1, f2::facet) is
  a::meta.type(f1.z);
begin logic
  c1:f1(x,a);
  c2:f2(a,z);
  tc1:M__type(a) <= M__type(f2.x);
  tc2:T <= M__type(f1.x);
  tc3:M__type(f2.z) <= T;
end facet batch_seq;
  
```

Figure 2 - Example batch-sequential architecture

Summary

Feature-based retrieval and adaptation provide mechanism for efficiently finding and adapting existing components to solve a new problem. Feature-based retrieval uses necessary conditions to classify and filter potential candidate solutions by eliminating components from the specification matching process. The result is a more efficient process that does not waste computational resources on evaluating failed matches. Adaptation architectures indexed by the match criteria lattice provide mechanisms for adapting retrieved solutions in a black-box fashion. Deep understanding of retrieved components is not required as they are reused in the adaptation architecture in a black-box fashion.

References

Alexander, P. and C. Kong. 2001. “Rosetta: Semantic Support for Model-Centered Systems-Level Design,” *IEEE Computer* **34**(11):64-70.

Fischer, B., “Deduction-Based Software Component Retrieval,” Ph.D. Thesis submitted to Universitat Passäu.

Garlan, D., R. Monroe and D. Wile. 1997. “ACME: An Architecture Description Interchange Language,” *Proc. of CASCON’97*, 169-183.

Patil, M. and P. Alexander. 2000. “A Component Retrieval System Using PVS,” *Theorem Proving in Higher Order Logics*, Portland, OR 2000.

Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

Penix, J. and P. Alexander. 1999. “Efficient Specification-Based Component Retrieval,” *Automated Software Engineering* **6**(2):139-170.

Fischer, B. and J. Schumann, “NORA/HAMMR: Making Deduction-Based Software Component Retrieval

Practical,” *Proceedings of the CADE-14 Workshop on Automated Theorem Proving in Software Engineering*, July 1997.

Shaw, M. and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

D. Smith and Lowry, M., “Algorithm Theories and Design Tactics”, *Science of Computer Programming* **14**:305-321.

Smith, D., “Constructing Specification Morphisms,” *Journal of Symbolic Computation* **15**: 571-606.

Smith, D., 1995. “Top-Down Synthesis of Divide-and-Conquer Algorithms,” *Artificial Intelligence* **27**(1):43-96.

Zaremski, A. and J.M. Wing. 1995. “Specification Matching of Software Components,” 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering.