REBOUND: A Framework for Automated Component Adaptation

John Penix
NASA Ames Research Center
Code IC, MS 269-3
Moffet Field, CA 94035 USA
jpenix@ptolemy.arc.nasa.gov

Abstract:

The REBOUND adaptation framework organizes a collection of adaptation tactics in a way that they can be selected based on the components available for adaptation. Adaptation tactics are specified formally in terms of the relationship between the component to be adapted and the resulting adapted component. The tactic specifications are used as matching conditions for specification-based component retrieval, creating a ''retrieval for adaptation'' scenario. The results of specification matching are used to guide component adaptation. Several examples illustrate how the framework guides component and tactic selection and how basic tactics are composed to form more powerful tactics.

Keywords: software architecture, formal specification, software reuse

Background

The Automated Software Engineering Group at NASA Ames Research Center focuses on the development and application of formal methods for synthesis, verification and reuse of safety and mission critical software. The group's program synthesis work is based on the construction of programs from reusable component libraries using automated theorem proving techniques [14, 13]. Our verification research investigates the application of formal methods (both model checking and theorem proving) to verify reusable software architectures and design patterns for AI-based autonomous systems and concurrent real-time systems [21]. The addition of Bernd Fischer and Johann Schumann [5, 6, 7, 25] as well as Jonathan Whittle [28, 29] in late 1998 will position the group to make strong advances in the application of formal methods to software reuse.

Position

Evidence of the need for adaptation in software reuse is evident in the wide spread programming language support of data-type generalization and parameterization. These methods take a specialization approach to reuse, where a component is designed abstractly and specialized at reuse time (either statically or dynamically). While these specialization techniques have permitted the development of reusable code, they focus on implementation level artifacts. Therefore, these techniques cannot avoid the limitations of concrete component reuse as described by Biggerstaff [3]: the reuse of small generic components does not provide enough functionality to impact the cost of a system, while the specialization required to construct a large component or framework limits its applicability. This causes a library scalability problem, where the size of the library must grow combinatorially as additional features are supported [3].

Biggerstaff provides a convincing argument that solving the library scalability problem requires moving from the specialization model of reuse to a generational or compositional model of reuse. (Similar views are expressed by Batory [2] and Kiczales [12].) The model Biggerstaff suggests is a library of *factors* that are combined together to result in the component to be used. Current technology supports a layers of abstraction (LOA) approach to this solution, where factors are implemented as functions and composition is done using run-time function calls [2]. Biggerstaff discredits the use of module interconnection languages because they prevent optimization by placing boundaries between components.

In the context of architecture description languages, module or component boundaries are not necessarily just conceptual boundaries, but may correspond to physical boundaries in the system. In dynamic architectures [17], component interactions are not known until run-time, removing the possibility of static optimization even between homogeneous, localized components. It follows that an LOA-optimization approach to combining subcomponents into a larger system is inappropriate in the context of software architecture. Factored reuse at the architecture level requires a more elegant solution to component composition, one that does not require integration of implementation level artifacts. The loss in efficiency is offset by the fact that architecture level compositions can be more powerful than those available from an LOA approach.

Approach

We have developed a framework for automated component adaptation and composition , dubbed REBOUND (REuse Based On UNDerstanding). The function and interface aspects of a component are separated and then composed at reuse time to generate a component with the correct function and interface combination. Composition takes the form of adaptation, supported by tactic specifications that describe structures used to adapt components. The tactic specifications are used to generate matching conditions that describe adaptable components. Specification-based component retrieval [5, 20] can then identify adaptable components using these matching conditions.

The goal of the adaptation framework is to guide the search for a solution based on existing library components. This search is over the space of all possible designs that can be constructed from the components and tactics in the library. The exploration of the search space must be limited due to the potentially large number of architecture and component combinations. The framework guides the search to avoid designs that are incorrect or require components not in the library.

We distinguish between several kinds of adaptation within REBOUND:

1. Type Adaptation - specialization of an abstract type or behavioral type substitution.
2. Interface Adaptation - alters the interaction style of the component.
3. Behavior Adaptation - changes the function of the component either by composing it with other components or replacing a subcomponent.

The effects of adaptation tactics are specified in terms of formal relationships between the component to be adapted and the resulting adapted component. It is possible that a component may require any combination of these three kinds of adaptations. This is supported by composing tactic specifications to make more powerful tactics.

Type Adaptation

Type adaptation occurs when an abstract or generic type is specialized. This kind of adaptation can normally be identified and carried out by signature matching tools [30]. It must be considered separately here, because of the potential to combine type adaptation with other kinds of adaptation.

Interface Adaptation

Interface adaptation determines how the component to be reused is bound to the problem specification. Interface adaptations range from simple type conversions to *wrappers* that encapsulate sophisticated control structures. A wrapper is an architecture that contains one component. By altering a component's interface, it changes the way that it interacts with its environment. One kind of interface adaptation is type conversion [24]. This differs from type substitution in that the source and destination type are not (necessarily) related by the type hierarchy. A type conversion operator is used to convert between the two types.

Matching conditions for component retrieval can be generated by specializing a generic `Type Conversion Wrapper` specification with specific type conversion operators. In the REBOUND framework, common type conversion operators are identified during domain analysis. The more general case of finding or synthesizing a component that implements the proper type conversion is covered by behavior adaptation. The concept of the type conversion wrapper can be generalized to specify more powerful wrappers. If fact, any architecture specification that has all but one component instantiated can be considered a wrapper specification. However, for the purposes of component adaptation, it is important to stick to simple, intuitive tactics to: 1) increase their potential to be reused and 2) simplify the process of selecting and specifying wrappers during domain analysis.

Behavior Adaptation

Behavioral adaptation tactics are applied in a incremental and constructive manner, using the architectural constraints as a guide while selecting components to plug in [19]. Maintaining the validity of the constraints with each component selection, guarantees a correctly instantiated system.

The goal of selecting architectures for behavior adaptation is to have a potential adaptation strategy for a range of component matching conditions. The choice of specific architectures should be driven by common problem decomposition tactics from the application domain. For example, sequential composition of filters is a common way to break up a digital signal processing system.

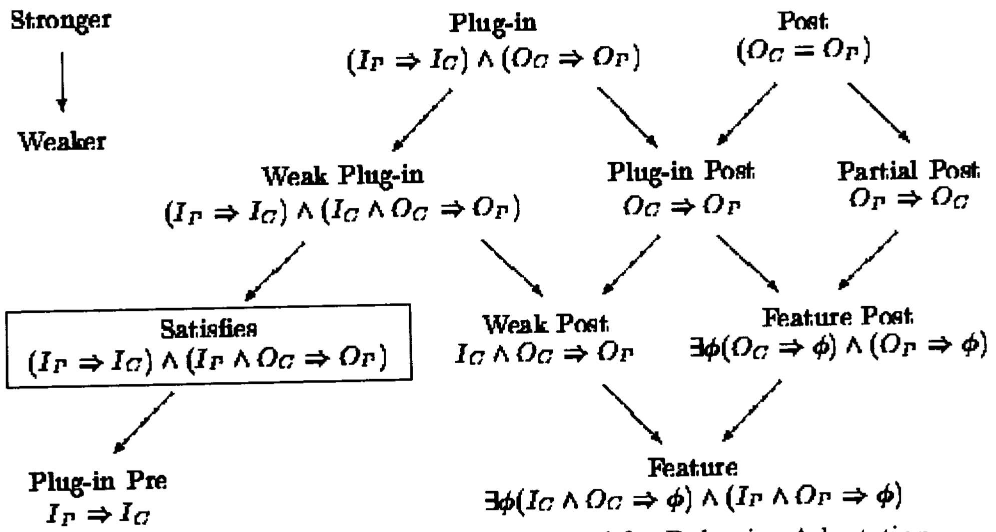
Stronger
Weaker
Plug-in
$(I_P \Rightarrow I_C) \wedge (O_C \Rightarrow O_P)$
Post
$(O_C = O_P)$
Weak Plug-in
$(I_P \Rightarrow I_C) \wedge (I_C \wedge O_C \Rightarrow O_P)$
Plug-in Post
$O_C \Rightarrow O_P$
Partial Post
$O_P \Rightarrow O_C$
Satisfies
$(I_P \Rightarrow I_C) \wedge (I_P \wedge O_C \Rightarrow O_P)$
Weak Post
$I_C \wedge O_C \Rightarrow O_P$
Feature Post
$\exists\phi(O_C \Rightarrow \phi) \wedge (O_P \Rightarrow \phi)$
Plug-in Pre
$I_P \Rightarrow I_C$
Feature
$\exists\phi(I_C \wedge O_C \Rightarrow \phi) \wedge (I_P \wedge O_P \Rightarrow \phi)$

Figure 1: Lattice of Specification Matches Used for Behavior Adaptation

The lattice of specification matches used to guide behavior adaptation is shown in Figure 1. This lattice is extended from the one described in our previous work [19, 20]. The highlighted Satisfies match indicates a component that can be reused to solve a problem. The goal of behavior adaptation is to alter the behavior of a component so that it matches under a condition at least as strong as Satisfies. This is accomplished by associating a composition architecture and a heuristic for instantiating the architecture with each matching condition.

Table 1 shows which matches each architectures is associated with and the instantiation rule for the match. Justification of the rules can be see by attempting to apply the architectures in the different situations [18]. For example, if a component matches under Plug-in Pre, it does not help to combine it with another component using the parallel or alternate architectures; the ''missing'' component is identical to the original problem. However, putting the component in the first position of a sequential architecture allows the derivation of a missing component specification that accepts the valid outputs of the first component and produced valid problem outputs.

Architecture	Match	Instantiation Rule
Sequential	Plug-in Post Weak Post Feature Post	Plug component into second position and derive specification for component to satisfy missing precondition
	Plug-in Pre	Plug component into first position and derive specification for component to satisfy missing precondition
Parallel	Feature	Combine components with features that total to problem features
Alternate	Weak Post	Derive specification for component to handle missing cases
	Feature	Combine components with features that total to problem features

Table 1: Adaptation Architectures Associated with Matching Conditions

Discussion

The framework embodies several techniques that limit search by avoiding designs that will either not provide the correct functionality or not terminate in existing library components. The main technique is the generation of matching conditions based on specialized adaption tactics. This leads to the selection of problem decomposition strategies that generate subproblems corresponding to library components. Problem decomposition strategies that do not lead to existing components (and are therefore dead-ends from a reuse perspective) are not discarded.

The framework limits the application of automated reasoning by confining it to two situations: 1) verifying component matches and 2) generation of subproblem specification during behavior adaptation. In addition, the search over the solution space is guided by the relationship of components to the problem, similar problems as determined by interface adaptation tactics, and the heuristics for instantiation the behavior adaptation tactics. Together, these tactics direct the search toward solutions that reuse components in library.

In the case where a large number of components are retrieved for a query, the match hierarchy can be use to select components for adaptation. In general, the closer a match is to Satisfies match, the sooner it should be selected for adaptation. The components requiring interface adaptation should only be considered after components that match the same way, but do not require interface adaptation.

A limitation of the framework is that the size (meaning level of abstraction or granularity) of the components in the library determines the size of the problems that can be solved by the system. The problem must be of similar size as the components in the library, because it is immediately compared to the components in the library. The adaptation tactics may allow the bridging of one or two abstraction levels (for example, the combination of a type conversion and a sequential composition might be considered two abstraction levels). However, large problems might require several levels of decomposition before reaching the abstraction level of the components. This limitation could be relaxed by allowing a few rounds of purely top-down problem decomposition before attempting the bottom-up component retrieval and adaptation. However, this would increase the search space proportional to the number of problem decomposition alternatives considered.

Comparison

The main contribution of this work is the development and evaluation the use of specification matching results to select component adaptation strategies. This builds upon the large body of research that has investigated specification-based (or deductive) component retrieval [5, 6, 7, 9, 10, 15, 16, 20, 25, 31]. The framework extends the traditional type abstraction/specialization adaptation paradigm [8] by adding interface and behavior adaptation.

Behavior adaptation is an extension of the work done on Kestrel's Interactive Development System (KIDS) [26, 27, 11]. In KIDS, the structure of specific algorithms such as global search or divide and conquer are represented algorithm theories. The generalization in REBOUND is that adaptation tactics are specified in terms of subcomponent problem theories rather than operators, allowing the construction of hierarchical systems.

The Inscape [23] environment developed by Perry uses a formal model of software interconnection [22]. Predicates are used to define preconditions and postconditions and obligations for functional components. Predicates are propagated throughout the system to support analysis and evolution, but not verification. The emphasis is on pragmatic use of specifications, therefore, the specification language is limited to conjunctions of predicates. The Inscape semantic interconnection model has recently been integrated into the GenVoca software system generators [1]. The REBOUND framework can be distinguished from these systems in several ways. First, the specification language is type first-order logic. Second, there is a distinction made between components and adaptors/architectures. This structure aids in the application of heuristic knowledge in guiding the construction of a system. The relationship between GenVoca and REBOUND is a point that deserves further study.

References

1 Don Batory and Bart J. Geraci. Validating component compositions in software system generators. In Murali Sitaraman, editor, *Fourth International Conference on Software Reuse*, pages 72-81, Orlando, Florida, April 1996. IEEE Computer Society Press.

2 Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca model of software-system generators. *IEEE Software*, pages 89-94, September 1994.

3 Ted J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *3rd International Conference on Software Reuse*, pages 102-109, Rio de Janeiro, Brazil, November 1994.

4 Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability - Concepts and Models*, volume 1. ACM Press, 1989.

5 B. Fischer, J. Schumann, and G. Snelting. Deduction-based software component retrieval. In *Automated Deduction - A basis for applications, Volume III Applications*. Kluwer, 1998.

6 Bernd Fischer and Johann Schumann. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proc. CADE-14 Workshop on Automated Theorem Proving in Software Engineering*, July 1997.

7 Bernd Fischer and Johann Schumann. SETHEO goes software engineering: Application of ATP to software reuse. In *Proc. CADE-14*, July 1997.

8 J. Goguen. Parameterized Programming. *IEEE Transactions on Software Engineering*, SE-10(5):528-543, 1984.

9 Jun-Jang Jeng and Betty H. C. Cheng. Specification matching: A foundation. In *Proceedings of the ACM Symposium on Software Reuse*, Seattle, Washington, April 1995.

10 L. Jilani, J. Desharnais, M. Frappier, R. Mili, and A. Mili. Retrieving software components by milimizing functional distance. unpublished, 1997.

11 Richard K. Jüllig. Applying formal software synthesis. *IEEE Software*, pages 11-22, may 1993.

12 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997.

13 M. Lowry and J. Van Ballen. `Meta-Amphion`: Synthesis of efficient domain-specific program synthesis systems. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 2-10, Boston, MA, November 1995. IEEE Computer Society Press.

14 Michael Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood. A formal approach to domain-oriented software design environments. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, pages 48-57, September 1994.

15 A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement based system. In *Proc. 16th Int'l Conf. on Software Engineering*, pages 91-100, Sorrento, Italy, May 1994.

16 A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering*, 23(7):445-460, July 1997.

17 Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, Kyoto, Japan, April 1998.

18 John Penix. *Automated Component Retrieval and Adaptation Using Formal Specifications*. PhD thesis, University of Cincinnati, April 1998.

19 John Penix and Perry Alexander. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, pages 535-542. Knowledge Systems Institute, June 1997.

20 John Penix and Perry Alexander. Efficient specification-based component retrieval. *Automated Software Engineering*, 1998. To appear.

21 John Penix, Perry Alexander, and Klaus Havelund. Declarative specification of software architectures. In *Proceedings of the 12th International Automated Software Engineering Conference*, pages 201-209. IEEE Press, nov 1997.

22 Dewayne E. Perry. Software interconnection models. In *Procedings of the 9th International Conference on Software Engineering*, 1987.

23 Dewayne E. Perry. The Inscape environment. In *Proceedings of the 11th International Conference on Software Engineering*, 1989.

24 James M. Purtilo and Joanne M. Atlee. Module reuse by interface adaptation. *Software: Practice & Experience*, 21:539-56, June 1991.

25 Johann Schumann and Bernd Fischer. NORA/HAMMR: Making deduction-based software

component retrieval practical. In *Proceedings of the 12th IEEE International Automated Software Engineering Conterence*, pages 246-254, Incline Village, NV, November 1997. IEEE.

26 Douglas R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024-1043, 1990.

27 Douglas R. Smith and Michael R. Lowry. Algorithm Theories and Design Tactics. *Science of Computer Programming*, 14:305-321, 1990.

28 A. Whittle, J. Bundy and H Lowe. Supporting programming by analogy in the learning of functional programming languages. In *Accepted for Poster Presentation at The 8th International Conference on AI in Education (AIED)*, 1997.

29 J. Whittle and E Melis. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 1998. to appear.

30 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.

31 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.

Biography

John Penix is a research scientist in the Automated Software Engineering Group at NASA Ames Research Center. He received a PhD in Computer Engineering from the University of Cincinnati, Cincinnati, Ohio in 1998. His dissertation, "Automated Component Retrieval and Adaptation Using Formal Specifications", received the Distinguished Dissertation Award from the University of Cincinnati Department of Electrical and Computer Engineering and Computer Science. His research interests lie in the intersection of software reuse, software architecture, automated reasoning and formal verification. John is a member of IEEE and ACM SIGART and SIGSOFT.

John Penix
Mon Aug 31 17:23:20 PDT 1998