Intelligent Component Retrieval via Automated Reasoning

Bernd Fischer,[*] Michael Lowry[+] and John Penix[+]
NASA Ames Research Center
RIACS*/Code IC+, MS 269-2
Moffet Field, CA 94035 USA
{fisch,lowry,jpenix}@ptolemy.arc.nasa.gov

Abstract

Effective application of AI techniques requires an understanding of how the representation vs. reasoning tradeoffs impact the domain of interest. In this paper we evaluate these tradeoffs for the software engineering problem of automatically retrieving components from a library. We generalize our experience in building several automated component retrieval and automated program synthesis systems. We then develop a framework in which we can compare the tradeoffs taken in the various approaches and speculate as to how to effectively combine aspects of the approaches for future applications.

Introduction

Software component technologies such as Java Beans, CORBA and Microsoft COM are quickly becoming industry standards for developing software that can be easily integrated, reconfigured and updated (Englander & Loukides 1997; Krieger & Adler 1998; Meyer 1999; Seetharaman 1998).

Component-based technologies have a potential impact on all phases of software development and maintenance. Component-based development reduces software integration costs by allowing component interfaces to be defined early in the development cycle when it is less expensive to make software modifications. They allow well-defined system architectures which localize the effects of software modifications and reduces quality degradation during maintenance. Testing and maintenance can be done at the component level, simplifying and automating the maintenance effort.

Last, but not least, component technologies can increase the potential for reuse within and among applications due to well-defined component interfaces, explicit architectures, and dynamic binding. However, it is well known that language features alone do not lead to successful software reuse. Software reuse adds difficult challenges to design that, if not adequately addressed, can lead to disastrous results such as the Ariane 5 explosion (Jézéquel & Meyer 1997).[1]

[1]While the Ariane example has become somewhat cliché, within NASA it hits too close to home to be forgotten.

We believe that the barrier to successful software reuse is a knowledge problem. Reuse places an emphasis on bottom-up design and therefore relies heavily on analysis capabilities to determine whether components are reusable. In supporting this analysis in software reuse, the emphasis has been on representing the knowledge required for analysis without regard for how it impacts the reasoning process, and hence the quality of the solution. This has limited the impact that automation can have on reuse, by limiting the level of assurance and quality of reuse analysis.

To understand the specific technical difficulty in automating reuse, it is helpful to decompose reuse into three activities: retrieval, evaluation and adaptation. Retrieval involves specifying a query and locating potential reuse candidates. Evaluation is determining the relationship between a retrieved component and the desired component. Adaptation requires making changes to a component to meet reuse requirements.

Effective automation of these activities using one representation is difficult because each activity requires different knowledge about a component. Retrieval benefits from an abstract classification of component function that supports efficient comparison. For automated evaluation to be useful, it must provide a designer with both a precise relationship and a high level of assurance. Hence, evaluation depends on a precise and detailed description of component behavior. Adaptation requires knowledge about the structure of a component and the functions of its sub-components. In all, the choice of a component representation scheme determines what software reuse activities can be automated effectively.

While source code has been the focus of most software reuse efforts, automated reuse requires information not described in source code. Source code provides a description of *how* a component performs its function. For purposes of retrieval, we are interested in precisely *what* this function is. This semantic gap makes it difficult to understand the function of the code and recognize a potentially reusable component. Formal specification languages provide the expressiveness and precision necessary to capture *what* the function of a component is. *Specification matching* (Rollins & Wing 1991; Moorman Zaremski & Wing 1997) applies theorem proving to evaluate relationships between specifications. Given a formal definition of reusability, specification matching can be used to evaluate the reusability of a component with respect to a requirements specification. In addition, automated reasoning can be used to determine what changes are necessary to reuse a component and guide component adaptation (Penix & Alexander 1997; Smith 1982).

Specification-Based Retrieval

Due to the overhead of automated theorem proving, specification matching is too computationally expensive to test a large number of components in an acceptable amount of time (Schumann & Fischer 1997; Mili, Mili, & Mittermeir 1997; Zaremski 1996). To attain practical response times, specification matching must be limited to evaluating a small number of components retrieved by a separate mechanism. An alternative to limiting specification matching is to limit the expressibility of the specification language, making retrieval more efficient (Perry 1987). However, this lowers the level of assurance during evaluation because it takes into account fewer aspects of a component's behavior. Approaches that do not distinguish retrieval from evaluation will either have inefficient retrieval or weak evaluation.

Our goal in supporting component retrieval is to identify a subset of the component library using the *semantics* of the component and problem specifications. The method must preserve (or effectively approximate) the reusability relationships between components using an efficient matching algorithm.

Information retrieval methods are evaluated by the two criteria of precision and recall (Salton & McGill 1983). Both are calculated from the set *REL* of *relevant* components which satisfy a given relevance condition with respect to the query and *RET*, the set of *retrieved* components which actually pass the filter. The *precision* p is defined as the relative number of hits in the response while the *recall* r measures the system's relative ability to retrieve relevant components:

$$p = \frac{|\,\mathrm{REL} \cap \mathrm{RET}\,|}{|\,\mathrm{RET}\,|} \qquad r = \frac{|\,\mathrm{REL} \cap \mathrm{RET}\,|}{|\,\mathrm{REL}\,|}$$

Ideally, both numbers would be 1 (i.e., the system retrieves all and only matching components) but in practice they are antagonistic: a higher precision is usually paid for with a lower recall.

A retrieval algorithm can also be considered as a filter which tries to reject irrelevant components only. We need some metrics to evaluate this filtering effect more precisely. We use the *precision leverage*

$$\delta_p = p \cdot \frac{|\,\mathcal{L}\,|}{|\,\mathrm{REL}\,|}$$

(where $\mathcal{L}$ is the full library) to denote the ratio between the precisions of the filter's input and output set, respectively. The factor $|\,\mathcal{L}\,| \,/\, |\,\mathrm{REL}\,|$ is also called the *general retrieval factor*; it is the inverse of the library's own precision. We also use the *fallout*

$$f = \frac{|\,\mathrm{RET} \backslash \mathrm{REL}\,|}{|\,\mathcal{L} \backslash \mathrm{REL}\,|}$$

which is the fraction of non-matching components which pass the filter as well as the *reduction* which is just the number of rejected components relative to the size of the filter's input. Finally, we define the *relative defect ratio* by

$$\delta_e = \frac{|\,\mathrm{REL} \backslash \mathrm{RET}\,|}{|\,\mathcal{L} \backslash \mathrm{REL}\,|} \cdot \frac{|\,\mathcal{L}\,|}{|\,\mathrm{REL}\,|}$$

as the relative number of rejected matching components in relation to the precision of the filter's input. Thus, a relative defect ratio greater than 1 indicates that the filter's ability to reject only irrelevant components is even worse than a purely random choice.

Existing Retrieval Techniques

Retrieval by Successive Filtering

An intuitive idea for limiting the number of components subject to specification matching without limiting the expressibility of the specification language is to pipe the candidate components through a series of filters of increasing complexity and (thus hopefully) strength. Fast but incomplete (and possibly even unsound) filters are used to identify non-matches upstream and thus prevent the relatively slow but complete and sound prover-based filters downstream from drowning in irrelevant components.

A typical filter inspects components one at a time, makes a local decision and either rejects the component or passes it through to the subsequent filter. In this setup, the rejection filters should be *recall preserving* (i.e., reject only components which are actually not relevant to the query) because matching components which are ruled out upstream never arrive at downstream filters and, consequently, can never be recovered again. On the other hand, the precision of the intermediate results then increases at each filtering step.

Filters which rely on a sound but incomplete proof procedure (and in our context all provers can be considered to be incomplete because they have to operate with limited resources) are recall preserving if their typical behavior is inverted, i.e., if the failure to find a proof is interpreted as "pass through". For filters which rely on unsound proof procedures, an engineering problem is to balance the loss of recall against the achieved reduction factors.

The NORA/HAMMR-system (Schumann & Fischer 1997; Fischer, Schumann, & Snelting 1998) is based on such a filter pipeline. It employs different sound and unsound rejection filters before the actual proof is attempted. We have experimented with two different techniques, rewrite-based simplification and model checking. Table 1 shows the results for these filters and their combinations achieved for a test library of ca. 120

list processing components; the numbers are averages based on ca. 120 queries with a total of 14.000 proof tasks. (Cf. (Fischer, Schumann, & Snelting 1998) for a more detailed description of the different filters and the test library.)

The first filter, $\mathcal{R}_{\text{domain}}$, is sound; it implements a set of rewrite-rules including the standard simplifications of first-order logic with equality and some domain-specific rules tailored towards the *list*-domain. Since the rewrite system is sound, this is a recall-preserving, zero-defect rejection filter, i.e., $r = 1.00$ and $\delta_e = 0.0$. Its actual filtering performance is already quite good. With a timeout of 0.5 seconds per proof task it filters out almost two thirds of the invalid tasks, resulting in a more than two-fold precision increase; even better results are achieved with longer timeouts.

The second filter, $\mathcal{R}_{\text{item}}$, builds on $\mathcal{R}_{\text{domain}}$; it contains additional rewrite rules to eliminate all quantifiers over the *item*-type of *list*. These rules are based on the generally unsound abstraction that the *item*-domain only contains a single element. In the experiments, however, this turned out to be a good engineering compromise because it finds only spurious "proofs" but no spurious counterexamples. Hence, its recall for the test library is still 100%. Moreover, this filter scales up to larger timeouts better than the sound $\mathcal{R}_{\text{domain}}$-filter. With an individual timeout of 5 seconds per task it still achieves average response times of roughly one minute per query. Under these time constraints,[2] the filter almost triples the precision of the answer set by filtering out more than 75% of the non-matches.

The final rejection filter used here, $\mathcal{M}_{2,1}$, applies MACE (McCune 1994), a first-order model checker based on the Davis-Putnam procedure, over an abstracted structure comprising two *list*-elements and again a single *item*-element. Due to the abstraction, this filter is also unsound and in contrast to the $\mathcal{R}_{\text{item}}$-filter it is not even well engineered. For short timeouts, both recall and precision are worse than those of the rewrite-based filters and the filtering effect is low. For longer timeouts, the filtering effect and the precision improve dramatically but at the expense of unacceptably low (for a rejective pre-filter) recall values. This is also reflected in the relatively high defect ratios which are approximately 0.6, independent of the timeout.

The last two entries of Table 1 contain results for filter pipelines comprising the combination of the two rewrite-based and all three filters, respectively. They reveal the desired "pipelining effect", i.e., the performance of the entire pipeline represents a significant improvement over the individual filters, in particular w.r.t. the fallout which can be reduced to almost 15%. However, they also reveal that the overall pipeline performance is still determined by the weakest filter. The addition of the unsound model checking filter leads to a dramatic loss of recall which is only partially offset by the slight improvement in precision and fallout. But the model checking filter's performance clearly benefits from the pre-selection of the other upstream filters, and its defect ratio drops significantly.

In NORA/HAMMR, these pre-filters are followed by a final confirmation filter applying automated theorem provers in order to increase the precision. Here, the effect of pre-filtering is even more dramatic: the pipeline becomes both faster *and* better. The reduced fallout allows longer individual timeouts for the prover (20 secs. in the experiments) which directly translates into a higher recall; at the same time, the sound $\mathcal{R}_{\text{domain}}$-filter also increases the recall. These effects add up to a final recall level of almost 90% with *guaranteed* precision (i.e., $p = 1.00$) and an average response time per query of approximately 10 minutes.[3]

Retrieval by Feature-Based Indexing

Another approach to automating specification-based reuse is to use indexing based retrieval techniques (Penix, Baraona, & Alexander 1995; Penix 1998a; Penix & Alexander 1999). In this approach, the emphasis is on making retrieval efficient. This is done by using a feature-based indexing scheme to classify components. The classification scheme consists of a set of domain dependent feature descriptors (abstract predicates) that define when a component should be assigned a specific feature. If one of these conditions can be inferred from the component specification then the component is assigned the appropriate feature. The formalization of the scheme permits automated classification of the specifications via theorem proving.

The output of the classification phase is a set of features that are used as a query to the component library. Retrieval is based on syntactic comparison of feature sets. The library retrieval mechanism returns components that have feature sets similar to the query. The components returned by the retrieval mechanism are passed on to a more detailed evaluation that uses specification matching to determine component reusability.

This semantic classification system was implemented using the HOL theorem proving system (Gordon 1989). Several precautions were taken to reduce the overhead of automated reasoning during classification (Penix 1998a). The feature sets for the library components are calculated off-line, a special purpose proof tactic was constructed to solve the classification proofs and inductive proofs were reduced by burying the induction into domain theory lemmas. These precautions result in an incomplete proof procedure. In addition, the reasoning in terms of the abstract indexes is both incomplete and unsound. These limitations should not be considered critical flaws until it is clear of the effects are on the practical performance of the system (Doyle & Patil 1991). In the context of component retrieval, loss of both soundness and completeness is not catastrophic, but effects the precision and recall of the retrieval al-

[2]This means (almost) interactive responses with the next hardware generation

[3]Cf. previous footnote.

	$\mathcal{R}_{\text{domain}}$		$\mathcal{R}_{\text{item}}$		$\mathcal{M}_{2,1}$		pipeline	
$T_{\max}$ (sec.)	0.5	5.0	0.5	5.0	0.5	5.0	2 * 0.5	3 * 0.5
T_{task} (sec.)	0.24	1.48	0.15	0.53	0.21	0.63	0.35	0.43
σ_T	0.21	2.10	0.16	1.28	0.23	1.00	0.36	0.48
T_{query} (sec.)	28.63	175.59	18.01	63.62	24.41	74.64	41.10	50.71
r (%)	100.00	100.00	100.00	100.00	66.65	45.59	100.00	66.65
p (%)	30.06	35.37	35.43	37.49	19.52	56.81	38.83	39.71
δ_p	2.32	2.73	2.73	2.89	1.50	4.38	2.99	3.06
f (%)	34.70	27.25	27.15	24.84	40.99	5.17	23.47	15.09
δ_e	0.00	0.00	0.00	0.00	0.60	0.61	0.00	0.43
red (%)	56.83	63.31	63.41	65.42	55.68	89.58	66.61	78.21

Table 1: NORA/HAMMR: Rejection Filters and Pipelining

gorithm. The impact of incomplete classification was evaluated by experimentally measuring its effect on retrieval performance (Penix & Alexander 1999).

The retrieval system was evaluated using the same component library as in NORA/HAMMR. The retrieval mechanism came very close to implementing the classification scheme in most cases: of the 63 components, the expected feature sets were derived for all but two specifications. Precision and recall averaged 30% and 70%, respectively, which is comparable to existing methods. The response time of the classification system during the experiments ranged from 0.15 to 0.66 seconds with an average of 0.35 seconds. This is well within the acceptable response time for an interactive system and is an improvement over existing filter-based approaches as NORA/HAMMR.

What is lacking from the approach is a systematic method for choosing the features used to classify the library. Formalizing an abstract concept that is shared among several components is difficult and picking useful ones is even harder. During several experiments, the intended intuitive meaning of a feature was shown to be incorrect by the system. Recognizing the utility of hierarchical and complementary features is a good start toward building better schemes. However, tool support would be necessary to scale the method to larger libraries and more sophisticated classification schemes.

Retrieval Framework

Traditionally, component retrieval methods are evaluated by distinguishing between relevant and retrieved components. Our experience indicates that this distinction is not adequate in the case of specification-based retrieval because the retrieval step relies on specification matching which is undecidable. To evaluate these systems it becomes necessary to distinguish between the relationship that *should* hold between a query and retrieved components and the components that are actually retrieved by a system in practice. Therefore, we further divide retrieval into two separate concepts, *matching* and *finding*, which correspond to the specification and implementation of a retrieval algorithm, respectively.

Distinguishing between relevance, matching and finding allows better evaluation of how representation and reasoning alternatives effect the performance of the retrieval system. In particular, it improves our understanding of where and why loss of recall and precision occur. Table 2 shows how our existing approaches are recasted within this framework.

Making the step from relevance to matching condition involves casting the relevance condition into the context of the query language. In the case of the filter-based method, the query language is first order logic, so refinement can be fully captured. This indicates that this stage is both sound and complete, which corresponds to preservation of precision and recall, respectively. However, in the index-based method, the set of possible features (the indexing/classification scheme) determines the relationship between relevance and matching. If adequate features do not exist, then it is possible that the refinement relationship can not be represented in all cases. In fact, the feature matching relation is symmetric while refinement is not. Therefore, in general, it is only possible for feature matching to approximate refinement. The classification scheme defines the nature of the approximation and how weak/strong the arrows are.

As mentioned above, the relationship between match condition and find condition is determined by the practical issues of properly implementing the match condition. In the filter-based system, an automated theorem prover is used to attempt to verify that the refinement relation (or any other match condition) holds between the query and a candidate component. The proof procedure is sound but incomplete. Failure to complete valid proofs corresponds directly to failure to retrieve a matching component. Therefore, in the filter-based system the incompleteness leads to a potential loss of recall. The soundness of the proof procedure ensures that precision is 1.

In the index-based system, the prover is used to automatically generate the feature-based representation of the components and queries. The incompleteness of the proof procedure can result in features not being assigned to components and queries. This can effect both the soundness and completeness of retrieval. If a com-

Method	*Relevance Condition*		*Match Condition*		*Find Condition*
Filter-based	Refinement	$\Leftrightarrow$	Refinement	$\not\Rightarrow$ $\Leftarrow$	Prover
Index-based	Refinement or Approximation	$\not\Rightarrow$ $\not\Leftarrow$	Semantic Features	$\not\Rightarrow$ $\not\Leftarrow$	Prover and Set Comparison

Table 2: Comparison of Retrieval Techniques

ponent or query is missing a feature that they should both share, then the system may fail to retrieve the component, affecting recall. If a component or query is missing a feature that would distinguish them as *not* matching, then the component may be retrieved erroneously, affecting precision.

Applications

Building Better Indices

Our framework has shown how the quality of index-based methods critically depends on the quality of the index itself, particularly on the discriminative power of the used features. Building better indexes is thus a prerequisite for scaling index-based retrieval up to larger libraries and bigger components but three constraints complicate this process:

- The quality of the index should be expressed in different terms than recall and precision in order to separate indexing from retrieval.
- The quality of the index should become visible immediately after the classification phase and before the proper retrieval phase starts in order to improve poor indexing immediately.
- The discriminative power of a feature may depend on the entire classification schema because feature-based retrieval is defined over feature *sets* and not over single features.

The techniques we have developed for specification-based browsing (Fischer 1998) can also be used to improve the schemas used for index-based retrieval.

Combinations of Indexing and Filtering

Feature-based indexing and successive filtering are two different techniques to achieve the same goal—a reduction of the number of candidate components subject to specification matching—but they can effectively be combined to improve the quality of specification-based retrieval even further. Two different combinations are possible.

First, filtering can be used to support the classification phase whose accuracy determines recall and precision of feature-based indexing. In effect, the entire classification phase can be considered as a retrieval task, although with reversed roles: each library component is used as a query into a "library" of features, all retrieved features are assigned to the component. For large, fine-grained classification schemas, this reversed retrieval task shares many deductive characteristics with the original retrieval task (e.g., high fraction of non-theorems, large axiom sets, etc.) and even the time issues become more important. This combination is straightforward and requires no change in the setup of the filtering process.

Second, an index can also be used within an initial pre-filter to reduce the number of emerging proof tasks "at the source". This combination, however, requires some modifications because the feature-based indexing method is not recall-preserving. The root cause for this loss of recall is (again) that the failure to find a proof for the validity of a feature is identified with its invalidity. This identification is in practice not correct because the automated reasoning step is incomplete.

This problem can be avoided if the feature set assigned to each component is divided into the *positive* features Φ^+ which are defined as in (Penix & Alexander 1997; Penix 1998a) and the *negative* features Φ^- which can be *proven not* to hold. Hence, $f \in \Phi^-(c)$ iff $\vdash pre_c \wedge post_c \Rightarrow \neg f$. Obviously, the positive and negative feature sets of a non-trivial and implementable component c (i.e., $\exists \vec{x}, \vec{y} \cdot pre_c(\vec{x}) \wedge post_c(\vec{x}, \vec{y})$) are always disjoint. It is then easy to show that a component c can (for the relevance condition of refinement) not be relevant if it has complementary feature to the query q, i.e., $\Phi^+(c) \cap \Phi^-(q) \neq \emptyset$ or $\Phi^-(c) \cap \Phi^+(q) \neq \emptyset$. These conditions can be used to improve recall as well as precision.

Scaling Up to Bigger Components

One of the major limitations for applying deductive component retrieval in practice is the ability to handle "components" more complex than procedures. In theory, specification matching conditions for procedures can be directly extended to the class/module level (Jeng & Cheng 1994; Zaremski 1996). However, there are many practical problems with this approach. First, the number of potential procedure matches that must be carried out increases sharply: for every component match there may be many potential mappings between procedures that must all be checked. The *pre/post*-specification style does not scale well as a query language for large complex classes and the extensions do not address the case where a group of tightly coupled classes are required. Finally, the pre-post method of specification does not work for components that may be required to operate in a multi-threaded environment, as is the case for Java.

We believe that our framework can be used to help address these practical issues of scaling up specification-based component retrieval. The first thing to notice is that the relevance conditions for classes and groups of classes are neither obvious or unique. For example, consider the Java Beans component model where components are described in terms of attributes, generated events and observed events. Syntactically, the Java type system will provide a refinement hierarchy based on the sets of attributes and events in a Bean interface. However, semantic refinement must be defined in terms of some notion of simulation of observable behavior. This definition might differ, for example, in terms of whether an event observation "must" or "might" cause the generation of another event in the future. In practice, it becomes necessary to relax this types of constraint in the context of multi-threaded execution. Further research is required to identify and formalize definitions of relevance for more complex components.

A practical definition of relevance should also take into account the explicit support of customization that is built into many component models. With respect to knowledge representation, this can be addressed by modeling components using parameterized specifications that indicate how specialization of visible attributes (the parameters) affect the behavior of the component (Penix, Alexander, & Havelund 1997; Penix 1998b). Initial studies have shown that deductive synthesis techniques have potential for supporting the reasoning tasks for component adaptation via parameter specialization (Penix & Alexander 1997; Penix 1998a)

The same issues affect the specification of matching conditions and queries as well. Many behavioral properties that can be informally described in terms of events are not easily expressed in terms of pre- and postconditions of their methods. For example, in a query language for retrieving Java Beans, properties should be expressed in terms of attributes and events, despite the fact that both are realized using Java method calls. The problems is that many properties of interest can involve relationships between method calls and attributes that cannot be expressed succinctly (if at all) in terms of pre- and postconditions.

Deductive Synthesis

By careful attention to the interaction of representation and computational complexity, we have been able to develop semantic-based component retrieval tools whose precision and recall exceed those of traditional retrieval algorithms. Furthermore, specification-based evaluation (matching) greatly reduces the cognitive load on the end-user in determining whether a component is suitable for her needs. This approach can be used not just in a manually-driven reuse context, but also in the context of an automated process for system generation through component composition.

A reference point of comparison is with the Amphion system (Stickel *et al.* 1994), which uses resolution theorem-proving to automatically synthesize a system by automatically assembling components given specifications of desired system behavior. In one application domain, the synthesis of space science observation systems composed of routines from JPL's NAIF toolkit, the number of routines in the library was roughly comparable to the list processing domain described earlier. Amphion follows the classical resolution-based refutation paradigm of synthesizing a program by generating "witness terms" for existential variables in the specification that would make the specification a theorem with respect to the background domain theory. In essence, a program was built up by successively extending the witness terms while the unresolved portion of the specification was decreased. The background domain theory consists of axioms about the application domain and axioms encompassing pre- and post-conditions of the components of the domain

The major technical difficulty in developing Amphion was in tuning the theorem prover strategy to enable reasonably efficient synthesis. Without tuning, Amphion exhibited time performance that empirically was exponential in the size of system specifications (Lowry *et al.* 1994). On average, specifications larger than thirty literals took longer than an hour to synthesize a program with an un-tuned theorem proving strategy. A relatively straightforward strategy of ordering partial solutions (descendents of the goal, or specification, clause) roughly by the number of literals that remained at the level of the abstract specification language achieved good results (Lowry *et al.* 1994). Further domain-specific tuning of the strategy achieved synthesis times on the order of seconds for programs consisting of ten to forty components. Empirically, there was still a shallow exponential in the required synthesis time with respect to the size of a specification. In practice, the performance was acceptable, but the strategy required returning whenever the domain theory was modified or extended with non-definitional axioms.

In later work, automatically generated decision procedures (Lowry & Van Baalen 1995) improved upon the hand-tuned strategy and addressed the problem of manual re-tuning of the theorem-proving strategy. For the purpose of this discussion, the decision procedures, in essence, replaced some of the axioms that caused exponential growth. Usually, the axioms that were replaced described mathematical aspects of the domain. From a search perspective, one role of the decision procedures was to replace eager instantiation of existential variables with lazy evaluation that delayed instantiation until enough constraints had been accumulated to generate only instantiations that would lead to valid solutions.

The specification-based retrieval research described in this paper has the potential to define a new set of decision procedures for deductive synthesis from components. General-purpose theorem proving to reason about the axioms describing the behavior of components in the library could be replaced with specialized decision procedures based on these retrieval algorithms. This would eliminate the majority of axioms in the domain theory, and lead to substantially less search. To implement the interface to deductive synthesis, the match condition would need to be generalized to a partial match condition, since no one component would match the entire specification. The partial match condition would define an ordering relation on retrieved components, and this ordering relation would be incorporated into the strategy of the theorem-prover. Perhaps most important, this provides a means of scaling up from simple components described through functional pre- and post-conditions to complex components. It provides a means of separating the reasoning about the individual components from the reasoning about the possible compositions of components.

Comparison

Traditionally, specification-based component retrieval is discussed in the more general context of information retrieval. (Mili, Mili, & Mili 1995; Mili, Mili, & Mittermeir 1998) develop a coherent terminology which can be seen as a first formalized approach to understanding component retrieval. In this approach, specification-based retrieval is seen as using specifications as the *surrogate representatives* to be used for retrieval. However, the effects of this abstraction step on recall and precision are not discussed in detail.

(Atkinson 1998) also develops an abstract framework to discuss component retrieval in more general terms. It uses Object-Z to recast the retrieval process as a data abstraction and shows how a variety of existing approaches (including NORA/HAMMR) can be considered as instances of this datatype. However, the focus of this work is on modeling structural commonalities of the retrieval mechanisms and not on the representation vs. reasoning tradeoffs and their effects on the domain metrics recall and precision as in our work presented here.

Conclusions

In this paper, we have presented the two most advanced specification-based component retrieval systems, NORA/HAMMR and REBOUND in a unified framework. This framework helps us to understand and evaluate how the various representation vs. reasoning tradeoffs involved in the systems' designs affect the established metrics of the general retrieval domain. It also allows us to identify weak points in the original approaches:

- In filter-based approaches as NORA/HAMMR the relevance and match conditions are (deliberately) identified but this (inadvertently) also coalesces the component retrieval and component evaluation phases.
- In feature-based indexing as in REBOUND incompleteness of the automated reasoning system may lead not only to a loss of recall (as expected) but also to a loss of precision.

While the inadvertent identification of retrieval and evaluation seems to be system-immanent and must thus be kept in mind, our framework shows us how the precision loss of feature-based indexing can be mitigated and, moreover, how both approaches can safely be combined.

Our future work will focus on the implementation of the described framework applications, in particular the different combinations of indexing and filtering. We will also continue our investigations on scaling specification-based retrieval up to bigger components and integration with deductive synthesis approaches.

References

Atkinson, S. 1998. Modelling formal integrated component retrieval. In *Proceedings of the 5th International Conference on Software Reuse*, 337–346.

Doyle, J., and Patil, R. S. 1991. Two theses of knowledge representation: languages restrictions, taxonomic classification, and the utility of representation services. *Artificial Intelligence* 48:261–297.

Englander, R., and Loukides, M. 1997. *Developing Java Beans*. Java Series. O'Reilly.

Fischer, B.; Schumann, J. M. P.; and Snelting, G. 1998. Deduction-based software component retrieval. In Bibel, W., and Schmitt, P. H., eds., *Automated Deduction - A Basis for Applications*. Dordrecht: Kluwer. 265–292.

Fischer, B. 1998. Specification-based browsing of software component libraries. In Redmiles, D. F., and Nuseibeh, B., eds., *Proc. 13th Intl. Conf. Automated Software Engineering*, 74–83. Honolulu, Hawaii: IEEE Comp. Soc. Press.

Gordon, M. J. C. 1989. HOL: A proof generating system for higher-order logic. In Birtwistle, G., and Subrahmanyam, P. A., eds., *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag. 73–128.

Jeng, J.-J., and Cheng, B. H. C. 1994. A formal approach to using more general components. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, 90–97.

Jézéquel, J.-M., and Meyer, B. 1997. Design by contract: The lessons of ariane. *IEEE Computer*.

Krieger, D., and Adler, R. M. 1998. The emergence of distributed component platforms. *IEEE Computer* 31(3):43–53.

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

Lowry, M.; Philpot, A.; Pressburger, T.; and Underwood, I. 1994. A Formal Approach to Domain-

Oriented Software Design Environments. In *Proceedings of the 9^{th} Knowledge-Based Software Engineering Conference*, 48–57. Monterey, CA: IEEE Computer Society Press.

McCune, W. W. 1994. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, Argonne, IL, USA.

Meyer, B. 1999. On to components. *IEEE Computer.*

Mili, H.; Mili, F.; and Mili, A. 1995. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering* 21(6):528–562.

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

Mili, A.; Mili, R.; and Mittermeir, R. 1998. A survey of software reuse libraries. *Annals of Software Engineering* 5:349–414.

Moorman Zaremski, A., and Wing, J. M. 1997. Specification matching of software components. *ACM Trans. Software Engineering and Methodology* 6(4):333–369.

Penix, J., and Alexander, P. 1997. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, 535–542. Knowledge Systems Institute.

Penix, J., and Alexander, P. 1999. Efficient specification-based component retrieval. *Automated Software Engineering* 139–170.

Penix, J.; Alexander, P.; and Havelund, K. 1997. Declarative specification of software architectures. In *Proceedings of the 12th International Automated Software Engineering Conference*, 201–209. IEEE Press.

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

Penix, J. 1998a. *Automated Component Retrieval and Adaptation Using Formal Specifications.* Ph.D. Dissertation, University of Cincinnati.

Penix, J. 1998b. Compositional specification of software achitecture. In Perry, D., and Magee, J., eds., *Proceedings of the 3rd International Software Architecture Workshop.*

Perry, D. E. 1987. The Inscape environment. In *Proc. 11th Intl. Conf. Software Engineering*, 2–12. IEEE Comp. Soc. Press.

Rollins, E. J., and Wing, J. M. 1991. Specifications as search keys for software libraries. In Furukawa, K., ed., *Proc. 8th Intl. Conf. Symp. Logic Programming*, 173–187. Paris: MIT Press.

Salton, G., and McGill, M. J. 1983. *Introduction to Modern Information Retrieval.* New York: McGraw-Hill.

Schumann, J. M. P., and Fischer, B. 1997. NORA/HAMMR: Making deduction-based software component retrieval practical. In Lowry, M., and Ledru, Y., eds., *Proc. 12th Intl. Conf. Automated Software Engineering*, 246–254. Lake Tahoe: IEEE Comp. Soc. Press.

Seetharaman, K. 1998. The corba connection. *Communications of the ACM* 41(10):34–36. (Introduction to Special Section).

Smith, D. R. 1982. Derived preconditions and their use in program synthesis. In *Proceedings of the Sixth conference on Automated Deduction*, volume 138 of *Lecture Notes in Computer Science*, 172–193. Springer-Verlag.

Stickel, M.; Waldinger, R.; Lowry, M.; Pressburger, T.; and Underwood, I. 1994. Deductive composition of astronomical software from subroutine libraries. In Bundy, A., ed., *CADE 12, 12th International Conference on Automated Deduction*, volume 814 of *LNCS*, 340–55. Nancy, France: Springer-Verlag.

Zaremski, A. M. 1996. *Signature and Specification Matching.* Ph.D. Dissertation, Carnegie Mellon University.