Finding Reusable UML Sequence Diagrams Automatically

William N. Robinson and Han G. Woo, *Georgia State University*

Software analysts create many artifacts, and until recently, these have been cumbersome to reuse. Reuser is a CASE tool that lets analysts automatically retrieve related UML artifacts for reuse. Its underlying graph-based concept-clustering technique performs well in structured domains.

Reuse in software-intensive systems is ubiquitous in practice. In general, *software reuse* is defined as "the process of creating software systems from existing software."[1] Software reuse does more than improve productivity in software development; it also increases the quality of the resulting software systems because it uses validated artifacts.

Requirements engineers benefit from reuse, even though it's a manual process relying on human recall in most situations. (See the "Artifact Reuse" sidebar for more information.) Analysts can recall prior specifications, adapt them to the current problem, and consolidate the reusable parts to create new requirements specifications.[2] However, requirements reuse lacks tool support. In his survey paper, Axel van Lamsweerde says, "Surprisingly enough, techniques for retrieving, adapting, and consolidating reusable requirements have received relatively little attention in comparison with all the work on software reuse. … The work in this area has not made sufficient progress to date to determine whether such approaches may be practical and may scale up."[3]

Our Reuser project seeks to address this tool gap by seamlessly assisting analysts as they reuse UML (Unified Modeling Language) artifacts. In particular, it automates artifact retrieval.

Finding the best match

Finding the best match is the first, and most important, step in reuse. Given an initial artifact, *art*, a matching function, μ, determines the best match, *m*, from a library of designs, *L*: m = μ(*art*, *L*). It's difficult to define a good matching function—especially for *use cases*, with their informal, irregular textual descriptions. In practice, a use case consists of the name, a short description, and a narrative of events among objects. Finding a good use case based on an ini-

0740-7459/04/$20.00 © 2004 IEEE

Artifact Reuse

Software analysts create many artifacts and often recall prior work as they recreate new artifacts. For example, when presented with an e-commerce problem, they create yet another e-commerce solution. This process needs to improve.

Design by patterns is a popular approach in which analysts create designs by instantiating predefined patterns from books or from computer-aided software engineering tools. But analysts find their own personalized designs most useful, and these are not generalized into patterns. Rather than instantiate a pattern, analysts copy and adapt their existing artifacts. Until recently, such artifact reuse has been cumbersome.

To demonstrate, let's walk through the process of defining an online sales system. Many analysts begin a sequence diagram similar to that shown in Figure A. The illustrated fragment is a common sequence diagram, but this is just the beginning. Analysts will also typically develop use case text, a class diagram, and a more complex sequence diagram. Having developed such systems before, many analysts retrieve related UML artifacts and copy and edit them as necessary. This manual strategy works well for a small set of predefined designs. However, the strategy breaks down when the design library contains many designs produced by different analysts. It becomes difficult and time-consuming to know the library's scope as well as to find the best-matching artifacts.

Some automated design retrieval approaches exist. However, they require specific design annotations, which most analysts are unwilling to add. Our REUSER tool lets analysts automatically retrieve related artifacts for reuse. Unlike other reuse approaches, this approach relies on the artifact structure and less on extra-artifact annotations or name similarity.

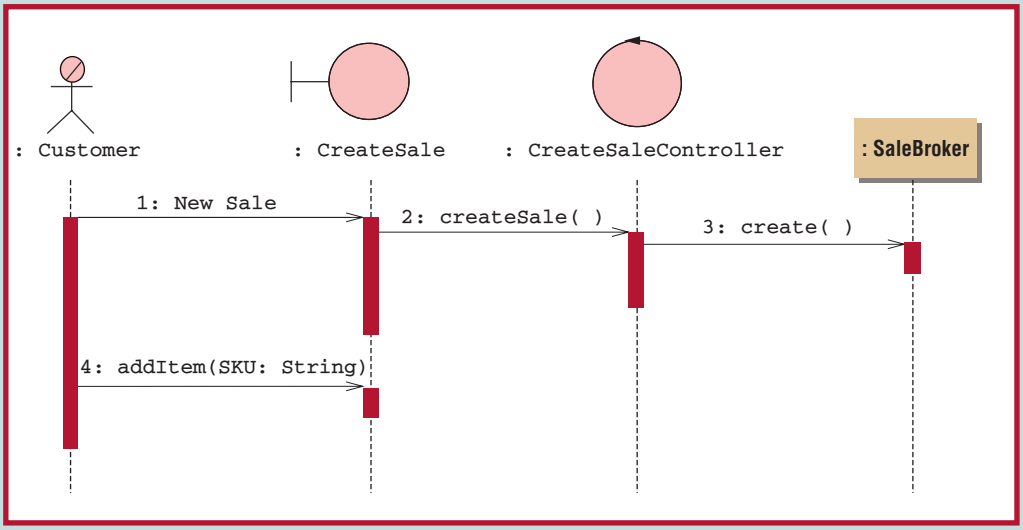

Figure A. An initial sequence diagram used to query a design library.

tial, incomplete use case typically means comparing use case narratives. Unfortunately, both humans and machines are rather poor at such natural-language comparisons. Consequently, matching functions considers two types of commonalities: concepts and relationships.

Concept matching

This is the most common way to find the best match. For example, the Crews (Cooperative Requirements Engineering With Scenarios) project structures use case scenarios using a *facet-based* classification system—the library designer predefines the facets.[4] Once properly annotated, an analyst searches the scenario library, selects an appropriate scenario, and adapts it for the current specification task. In a similar approach, an analyst calculates *syntactic* similarity between use case scenarios. Thomas Alspaugh and colleagues provide a weighted similarity metric using UML use case actors and extra-UML properties of goal, purpose, and viewpoint.[5] An analyst can adjust the attribute weights to suit the comparisons (for example, detecting commonality among scenarios). Maurits Blok and Jacob Cybulski focus on the event flows of a use case sequence diagram.[6] Using the WordNet lexicon to classify an event, their tool represents a use case as a vector of event flow descriptors. It calculates the similarity of the use case vectors using an information-retrieval technique. Motoshi Saeki[7] uses the semantics of UML's «`extends`» and «`uses`» relationships to find similar or common parts of use cases.

Relationship matching

These techniques extend the comparisons among the attributed concepts to include concept relationships. These approaches focus on the graphs' structure defined by concepts (as

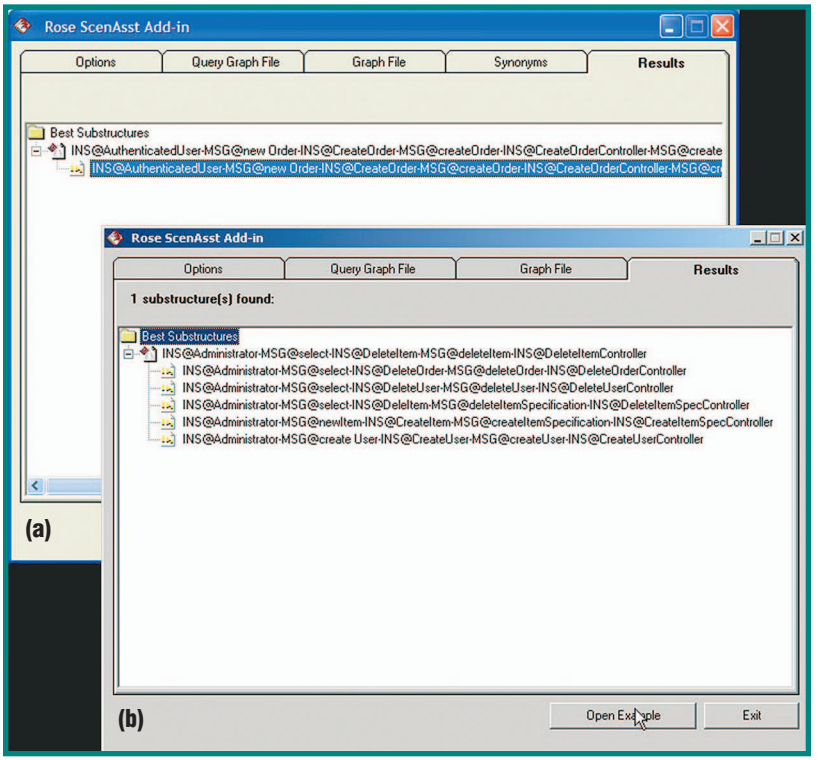

Figure 1. (a) The best-matching sequence diagram for the query shown in Figure A. (b) Matching sequence diagrams for another, unrelated query.

vertices) and relationships (as edges). The resulting similarity calculations are not summations across concept comparisons. Rather, we calculate whole graph similarities—for example, totaling the number of vertex or edge differences. The ReqColl (*Req*uirements *Coll*ector) project represents requirements as a conceptual graph in which graph similarities define requirement similarity.[8] Others use this same basic approach, including TARA (Tool Assisted Requirements Analysis),[9] IRA (Intelligent Reuse Advisor),[10] and KAOS (*k*nowledge *a*cquisition in automated *s*pecification of software).[2] The lauded Programmers Apprentice project also uses graph matching; however, it relies mostly on common programming constructs.[11] In contrast, today's reuse projects increasingly rely on extra annotations to support their matching algorithms. For example, GBRAM (Goal-Based Requirements Analysis Method) requires use cases to include goal and purpose annotations.[5] Consequently, this limits reuse to artifacts that include such annotations.

Relationship matching can be effective, even without extra annotations. In many domains, artifacts have regular structures that have similar uses. Molecular biology, image analysis, computer-aided design, and Web pages include common structures amenable to graph-based concept-learning and clustering techniques.[12] For example, given a partial computer circuit's graph representation, structure matching can find similar completed circuits from a library.[12] This approach can especially benefit analysts, as they often loath extra annotations because their value isn't observed until some future reuse request, which might never arise. So, a matching method that works on standard artifacts is a boon: analysts can reuse their own artifact libraries as well as any other libraries they'd like to access. Moreover, the approach works with the structure of both object-level design and class-level requirements scenarios.

You can apply relationship matching to UML. To find a reusable artifact, a tool calculates match scores based on the designs' structures, represented as graphs. Our research shows this approach works well for commonly structured UML models.

Reuse by structural relationships

You can apply our *reuse by structural relationships* approach in the following manner:

1. An analyst defines or selects an artifact library.
2. The tool calculates structure scores for the artifacts in the library.
3. To query the library, an analyst gives the tool a partially completed structure. (The tool calculates the structure score for the partial structure, searches the artifact library, and returns the *n* best matches.)
4. The analyst adapts the retrieved artifact to the current problem context.

This reuse approach applies to any UML artifact, including UML extensions such as stereotypes. To demonstrate the approach, we defined the REUSER tool. It's implemented as a Rational Rose add-in (including 2000–2003 versions); however, REUSER can work with any UML tool provided that you can encode the UML structures as directed graphs.

UML structures

REUSER assists the reuse of UML artifacts, including use cases, class diagrams, and sequence diagrams. In this article, we focus on sequence diagram reuse. Figure A (in the sidebar), for example, shows a partially completed sequence diagram passed to the tool. Figure 1a

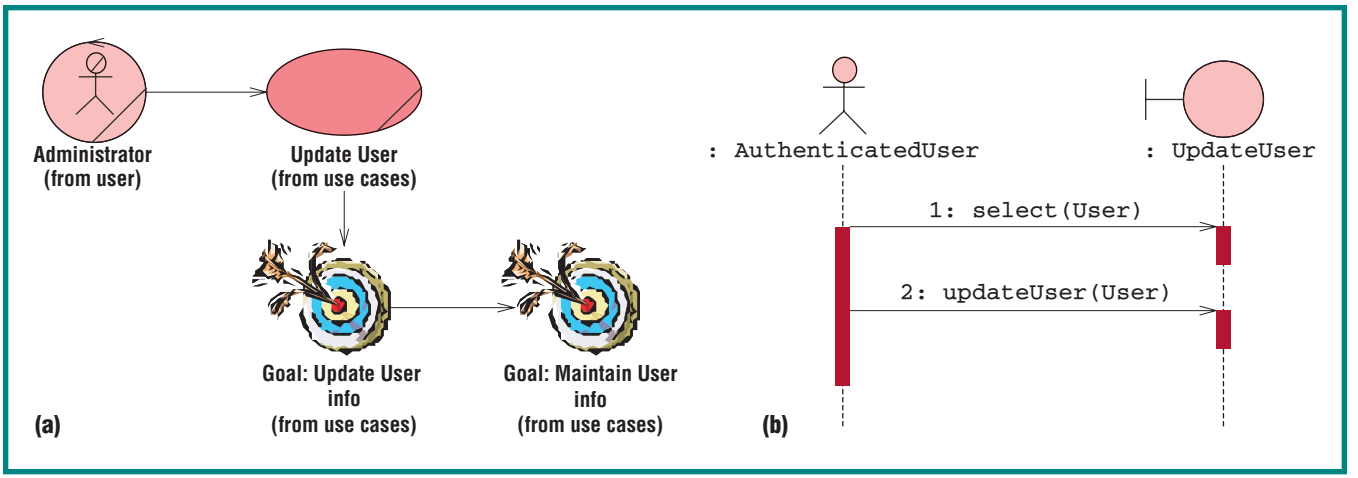

Figure 2. A UML (a) use case and goal structure and (b) sequence diagram.

illustrates the REUSER tool-finding matches for Figure A's query. An analyst can browse the list of the best matches and directly open each artifact from the list. The selected match must then be adapted to satisfy the current problem. The figure shows sequence diagrams displayed as a text string of their objects (prefixed with `INS`, denoting instance) and messages (prefixed with `MSG`). Matched artifacts are shown as indented below their initiating query. Figure 1b illustrates that REUSER can find multiple artifacts for a single query.

Figure 2 shows that you can reuse artifacts using UML extensions. REUSER automatically analyzes UML extensions. Figure 2a shows an extended use case diagram. In the diagram, the Administrator is a UML actor with the Rational Unified Process stereotype of business worker; similarly, the Update User use case makes use of a RUP stereotype. Figure 2a also shows two goals that the use case supports: Update User Info and Maintain User Info. These are displayed with a user-defined stereotype.

Figure 2b shows a simple sequence diagram for the use case. The Authenticated User object in the sequence diagram is an instance of a subclass of the Administrator class from the use case diagram. The two diagrams are interrelated and represent an initial use case sketch. Together, they form an artifact fragment, which is used to query the artifact library.

While viewing Figure 2's use case diagram, an analyst can ask REUSER to find reusable use cases. The tool will use both of Figure 2's structures to find matching use cases. Conversely, if the analyst begins with the sequence diagram, REUSER will return related sequence diagrams—again, based on both structures. In general, an analyst can select a subset of related artifacts, and REUSER will return the best-matching artifacts based on their structural correspondence.

Automated encoding

Rather than analyze the UML artifacts directly, REUSER analyzes a directed graph representation. REUSER translates UML artifacts into its internal representation by translating UML metamodel elements into a directed graph. By translating metamodel types into vertex types, the approach applies to all UML versions, including version 2.0 (see www.omg.org). UML metamodel associations become edges in the internal graph, while all other UML metamodel elements become vertices. Using vertex prefixes maintains as part of the translation UML metamodel element types—including UML extensions such as stereotypes. This results in a UML artifact's minimal graph representation.

Consider a UML artifact. Classes, objects, messages, stereotypes, and so on are encoded as vertices; UML associations and links become edges. A class instance, for example, becomes a vertex prefixed with `INS`, and a class is prefixed with `CLS`. A message vertex, prefixed with `MSG`, is located between its sending-object and receiving-object vertices. Edges represent UML metamodel associations, including

- A link between a sending object and a message
- A link between a message and a receiving object
- The instantiation relationship between an object in a sequence diagram and a class in a class model

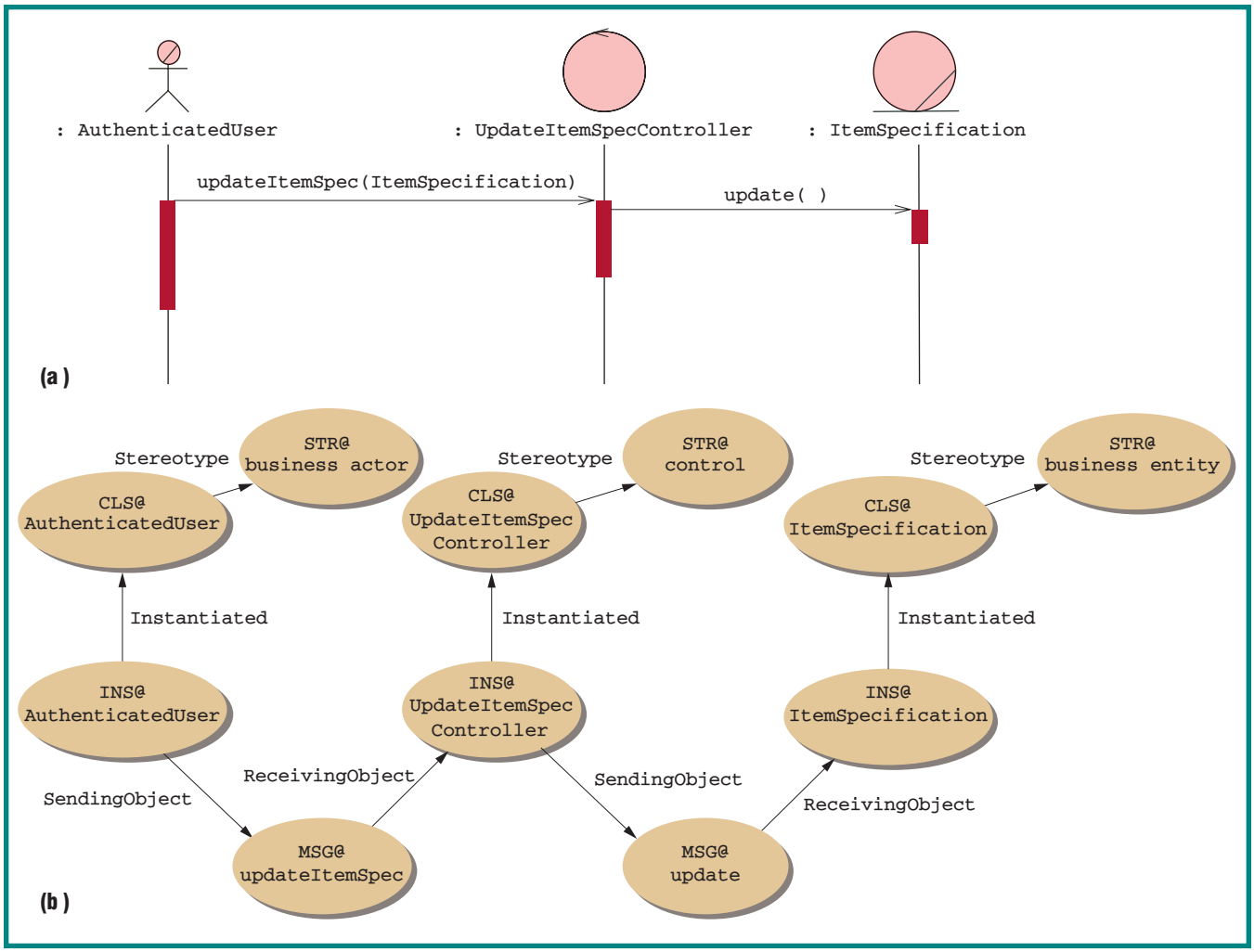

Figure 3. (a) A sequence diagram in a (b) Reuser graph format.

Figure 3 shows how Reuser encodes objects in a UML model as vertices and edges. The vertices have prefixes for classes (`CLS`) and their attributes (`ATR`), methods (`MTH`), associations (`ASC`), and stereotypes (`STR`). Class instances are prefixed with INS, and messages among them are prefixed with MSG. The edges support UML metamodel relationships (for example, `Stereotype`, `Method`, `Association`, `Inheritance`). Reuser similarly encodes UML 2.0 metamodel elements such as sequence diagram reuse references and alternatives (for example, `REF@A_Ref`, `ALT@Seq_#11`).

Automated matching

Reuser employs the Subdue algorithm to find matches for a given query graph, and it compares labeled directed graphs.[12] To form a match between two graphs, at least one node label must match. In UML, this typically begins with a matching stereotype name and is refined with matching names for some use cases, classes, or objects. A label synonym list, provided by the analyst, permits equivalence between different labels. As Figure 2 shows, the matches don't need isomorphic structures. In fact, close matches are most helpful because they provide new substructures that you can reuse. The algorithm is polynomial; however, the worst-case behavior is often avoided by applying structure-limiting constraints to the search.

The search algorithm finds "interesting" and repetitive substructures in graphs. It replaces each repeating substructure with a pointer to its minimal substructure, thereby compressing the graph. Repeating this process results in a minimal classification lattice in which lower-level concepts are included in the higher-level concepts. Inexact matching lets you substitute partially matching substructures. A threshold de-

```
Subdue ( graph G, int Beam, int Limit )
  queue Q = { v | v has a unique label in G }
  bestSub = first substructure in Q
  repeat
    newQ = {}
    for each S ∈ Q
      newSubs = S extended by an adjacent edge from G in all
        possible ways
      newQ = newQ » newSubs
      Limit = Limit - 1
    evaluate substructures in newQ by compression of G
    Q = first Beam substructures in newQ in decreasing order
        of value
    if best substructure in Q better than bestSub
    then bestSub = first substructure in Q
  until Q is empty or Limit ≤ 0
return bestSub
```

Figure 4. SUBDUE's classification algorithm.[12]

termines when two structures are similar enough to match. The analyst sets the threshold parameter, which ranges from 0 to 1, where 0 finds an exact match. The threshold must be less than the `transformation cost/structure size`, where `transformation cost` is the number of graph transformations required to make the structures isomorphic.

Given an artifact fragment as a query, the search algorithm can find similar artifacts by

- Classifying the artifact library into a concept lattice
- Comparing an input artifact query against the concept lattice
- Retrieving the *n* closest matching artifacts

Figure 4 shows SUBDUE's classification algorithm. Its beam search strategy searches for structures that best match the input structure. To classify an artifact library, a graph's uniquely labeled vertex initializes the search queue. Classification occurs through a substructure search. To expand the search, SUBDUE extends each vertex, `S ∈ Q`, by including its adjacent edges and associated vertex. A compression method determines a match. It replaces all instances of a classified structure observed in the input structure with a single vertex; an exact match compresses the input graph to one vertex. The more the algorithm compresses the input structure, the better the match—according to the Minimum Description Length principle.[12] You can limit the search by breadth, `Beam`, and number of expansions, `Limit`.

You can also use the classification algorithm to find matching artifacts. Rather than starting with an arbitrary vertex, search begins with the query structure. This biases the algorithm to expand structures that resemble the query structure and results in a list of structures that best match the query structure.

UML is particularly amenable to relationship matching because of its syntax, metamodel-supported extensions, and commonly applied idioms. For example, a Web-based UML model often includes Web-stereotyped definitions in common class and interaction patterns. As the simple model in Figure 4 shows, its graph structure of 16 typed vertices and 15 typed edges most resembles functions involving actor-interface-controller idioms. Through stereotyped patterns, analysts are effectively modeling with domain-specific languages in which similar structures have similar functions. Conversely, relationship matching is less likely to succeed when the artifacts have varied functions with few structural variations, and they are not segmented by their structures, such as sublanguages or idioms.

Evaluation

In this section, we evaluate the structural relationship reuse approach's effectiveness by means of various methods and summarize the results.

Retrieval case study

We conducted a case study, relying on an artifact library comprising 308 classes and 85 use cases in domains similar to order processing. We didn't use synonym lists because we

Table 1

Retrieved similar substructures for a partial query

Threshold	Query	
	Create Sale	Update User
0.1	None	None
0.2		
0.3		2 instances from Update User*
0.4		9 instances:
0.5	1 instance from Create Order*	2 from Update Item
		4 from Update Order
		1 from Create User
		2 from Update User*

* indicates best match

had sufficient matches rooted in the stereotype names and in some common class names. In the study, we presented partially completed use case designs to Reuser. A test set of 27 classes and seven partial sale-processing use cases were constructed as queries. For example, the sequence diagrams of Figures A and 3 were queries. To test Reuser, we requested matches for each of the seven partially completed sequence diagrams.

Table 1 summarizes the retrieval results for the two use cases: Update User (Figure 3) and Create Sale (Figure A). For the Create Sale query, Reuser found the best match sequence diagram for the Create Order use case at a 0.5 threshold. The query and the retrieved sequence diagrams share a similar structure, although the object and message names differ.

For the Update User query, Reuser discovered exact matches from two different versions of the Update User sequence diagram at a 0.3 threshold. The threshold 0.4 to 0.5 matching also discovered the same structure, yet suggested a broader range of nine similar cases. For example, the match found a similar sequence diagram that instantiates both the Update Item and Update User use cases. Overall, Reuser appears to retrieve similar sequence diagrams when presented with partial-sequence diagrams.

The preliminary case study's results suggest that Reuser effectively and efficiently finds good matches—for simple and complicated queries. However, to rule out author bias in our evaluation, we impaneled experts.

Expert panel

Five experts performed experimental tasks with Reuser. The experimental tasks consist of completing two partial sequence diagrams, one complicated and the other simple.

We selected the experts from faculty members in a US information systems department. They had on average 36.7 months of system analysis and design experience. They identified themselves as very knowledgeable about use cases (averaging 6.25 out of 7), sequence diagrams (5.25 out of 7), and the Rose case tool (5.75 out of 7).

Each expert created good sequence diagrams using Reuser. They reported that it took 21 minutes to complete a complex sequence diagram and seven minutes for a simple sequence diagram.

The experts confirmed our hypothesized belief that using Reuser affects quality and time efficiency for scenario authoring. Each expert quickly identified similar sequence diagrams that suited the experimental tasks. Additionally, each question for perceived usefulness and intention to use received high scores in a seven-point Likert scale (means ≥ 6). These results suggest that Reuser's approach is effective and that practitioners might adopt it. However, we can't draw any strong conclusions from the expert panel results because of the small sample size and lack of a comparison group.

Lab experimentations

We performed lab experiments to further validate the research. For independent variables, we employed a 3×2 factorial design with repeated measure (within subject). The three experimental groups included the control group, with no tool support or design library; group A, with a design library; and group B, with tool support and a design library. We gave each group two tasks of varied complexity.

Analysis of the preliminary experimental results indicates that the experimental groups performed as expected: group B was better than group A, which was better than the control group. Thus, Reuser appears to offer good support for retrieving reusable design artifacts.

Our results show that Reuser's structural relationship approach works. Functionally, it retrieves related, reasonably reusable artifacts. Additionally, analysts seem to accept the simple approach: while viewing a partially constructed artifact, you can simply ask the tool to find related ar-

tifacts. The underlying structure-based classification approach has repeatedly performed well in structured domains.[12]

However, with its polynomial-time algorithm, REUSER doesn't perform well with large query structures. In such cases, analysts can obtain good results by reducing the query size—for example, by leaving out a final subsequence from a sequence diagram. Expert analysts can obtain even better results by modifying the parameters to the underlying SUBDUE algorithm, such as search breadth, limit, and target structure size.[12]

Other approaches to reuse suffer similar, or worse, problems of scale. All approaches define a similarity metric; however, most often the approaches only use the metrics to group artifacts. Fewer reuse approaches use their metrics to define an efficient concept lattice. Moreover, most approaches rely on extra-artifact annotations to guide matching. A library concept lattice constructed from a library-specific metric might be an efficient approach. However, few organizations will pay for such customization, and few analysts are willing to work with the extra-artifact annotations.

Although REUSER's approach works, we see opportunities for future improvements such as

- Parameter and input tuning
- Weighted structure matching
- Library lattice visualization
- Pattern extraction

You can optimize query parameter values, such as match threshold, for a given library and patterns of query usage; you can automatically derive such values through sensitivity analysis.

Although REUSER supports dynamic customized queries, it doesn't support weighting among the criteria. To define a custom query, an analyst can include "custom" elements in the query structure; similar library structures will match. However, an analyst might deem certain elements or associations central to the query. Future work will support weighted query graphs.

We based REUSER's automatic classification on structural design commonalities. Thus, you can mine patterns from the library lattice. You might consider the more abstract and common patterns as empirical design patterns. Comparing such empirical patterns with the many descriptive patterns currently espoused could produce interesting results. In the meantime, analysts can improve their process's quality and efficiency by applying REUSER's structural relationship retrieval approach.

References

1. W. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, no. 2, 1992, pp. 131–183.
2. P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks," *Proc. 3rd Int'l Symp. Requirements Eng.* (RE 97), IEEE CS Press, 1997, pp. 26–37.
3. A. van Lamsweerde, "Requirements Eng. in the Year 00: A Research Perspective," *Proc. 22nd Int'l Conf. Software Eng.* (ICSE 2000), IEEE CS Press, 2000, pp. 5–19.
4. C. Rolland et al., "A Proposal for a Scenario Classification Framework," *Requirements Eng.*, vol. 3, no. 1, 1998, pp. 23–47.
5. T. Alspaugh et al., "An Integrated Scenario Management Strategy," *Proc. IEEE 4th Int'l Symp. Requirements Eng.* (RE 99), IEEE CS Press, 1999, pp. 142–149.
6. M.C. Block and J.L. Cybulski, "Reusing UML Specifications in a Constrained Application Domain," *Proc. 5th Asia Pacific Software Eng. Conf.* (APSEC 98), IEEE CS Press, 1998.
7. M. Saeki, "Patterns and Aspects for Use Cases: Reuse Techniques for Use Case Descriptions," *Proc. 4th Int'l Conf. Requirements Eng.* (ICRE 2000), IEEE CS Press, 2000.
8. K. Ryan and B. Mathews, "Matching Conceptual Graphs as an Aid to Requirements Re-use," *Proc. IEEE Int'l Symp. Requirements Eng.* (RE 93), IEEE CS Press, 1993, pp. 112–120.
9. A. Finkelstein, "Re-use of Formatted Requirement Specifications," *IEEE Software Eng. J.*, vol. 2, no. 5, 1988, pp. 186–197.
10. N. Maiden and A. Sutcliffe, "Exploiting Reusable Specifications through Analogy," *Comm. ACM*, vol. 35, no. 4, 1992, pp. 55–64.
11. C. Rich, "Knowledge Representational Languages and Predicate Calculus: How to Have Your Cake and Eat It Too," *Proc. 2nd Nat'l Conf. Artificial Intelligence* (AAAI 82), AAAI Press, 1982, pp. 193–196.
12. I. Joyner, D.J. Cook, and L. B. Holder, "Discovery and Evaluation of Graph-Based Hierarchical Conceptual Clusters," *J. Machine Learning Research*, Oct. 2001, pp. 19–43.

About the Authors

William N. Robinson is an associate professor in the Department of Computer Information Systems at Georgia State University. His research interests include requirements engineering, model-driven systems development, and e-commerce. He received his PhD in computer science from the University of Oregon. He is a member of the IEEE, the ACM, and the Association for Information Systems. Contact him at Computer Information Systems Dept., J. Mack Robinson College of Business, Georgia State Univ., Atlanta, GA 30302-4015; wrobinson@gsu.edu; http://cis.gsu.edu/~wrobinso.

Han G. Woo is a doctoral candidate in the Department of Computer Information Systems at Georgia State University. His research interests include requirements engineering, process knowledge management, and data mining. He received his MS in management information systems from Seoul National University. Contact him at the Dept. of Computer Information Systems, Georgia State Univ., 910, 35 Broad St., Atlanta, GA 30303; hwoo@student.gsu.edu.