Building A Repository of Software Components: A Formal Specifications Approach

R.T. Mittermeir

Institüt für Informatik
Universität Klagenfurt
A-9022 Klagenfurt, Austria
e-mail: roland@samos.ifi.uni-klu.ac.at

R. Mili, A. Mili

University of Ottawa
Department of Computer Science
Ottawa, Ont. K1N 6N5 Canada
e-mail: {rmili,amili}@csi.uottawa.ca

Abstract

In our correspondence to the fifth Workshop on Software Reuse [1], we had discussed the design and preliminary implementation of a repository where software components can be stored and retrieved automatically, using a formal-specification approach. In this paper we report on our progress, by describing the first prototypes we have for this repository, as well as preliminary assessment of their performance.

Keywords: formal specification, program correctness, program refinement, theorem proving, software repositories, storage and retrieval of software components.

Workshop Goals: assessing how our work fits in the overall problem of software reuse; identifying potential partners for developing our system on a larger scale, and with sharper focus.

Working Groups: reuse and formal methods, tools and environments, reuse process models, domain analysis and engineering, reuse handbook.

1 Background

The approach advocated in this paper results from the combination of two research efforts:

- An effort by the first author to build a software archive as a stratified multidimensional classification structure for software components.
- An effort by the second and third authors to investigate various features of the specification process and the specification product, as well as the construction and verification of programs.

In a recent joint publication [2] the first and third authors make a proposal for constructing a database of software components, based on the following premises:

- Entries of the database have the form of $\langle specification, program \rangle$ pairs, where the *program* component is correct with respect to the *specification* component.
- The set of database entries is structured by means of the refinement ordering between the specification components of the entries.
- The *retrieve key* for retrieval operations on the database take the form of a specification, and seeks to determine all the programs of the database that are correct with respect to the key, by matching the key against the specification components of database entries.

Our design rationale is that by representing database entries with arbitrarily abstract specifications and by letting the match between a store key and a retrieve key be defined by the refinement ordering (rather than strict equality), we deal effectively with the diversity of software components and the complexity of organizing such components in a database.

In this position paper we report on our progress in implementing this software repository, using C under $Unix$, and an experimental theorem prover (*Otter*, ©Argonne National Laboratory). Among the Functions we have implemented we mention: *Storage* of software components; *Retrieval* of software components; and *Approximate Retrieval* of software components, to be invoked whenever (exact) retrieval fails to produce results.

2 Position

2.1 Statement of the Position

We submit the position that the storage and retrieval of software components in an automated software repository is technologically possible, provided these components are described by formal specifications. This means in principle that software designers can use off-the-shelf software, in the same manner as hardware designers make use of off-the-shelf hardware to design hardware systems.

We qualify this position slightly, by adding that although the storage and retrieval is possible, it is not (yet) efficient: the theorem proving parts of our system remain something of a bottleneck (proofs can take unreasonably long).

To place our position in context, we add two premises:

- First, we realize that there is a great deal more to software reuse than the mere mechanics of storing and retrieving software components; more important than storing and retrieving components is the ability to design components that are worthy of reuse; also more important is the ability to define design methodologies that make use of reusable components. On the other hand, as several authors point out [3, 4, 5, 6], software reuse is as much (if not more) a managerial and organizational challenge as it is a technological challenge.

- Second, our pattern of software reuse depends on programs being specified formally. In an organization that does not practice formal methods, it may or may not be cost effective to formally specify software components for the sole purpose of integrating them in a software repository. On the other hand, one might argue that the use of formal methods is cost-effective on its own merits [7].

To motivate our position (and illustrate the extent of its validity) we give below some details about the implementation of our system.

2.2 Substantiating the Position

In this section we give very summary details about the implementation of our system. First, we discuss its overall architecture.

In any database system, the key to efficient retrieval is the availability of an ordering between the database entries. Typically, database entries are totally ordered, i.e. for any two distinct entries, one is necessarily smaller than the other (for the given ordering). For software components, we could not identify such a *total* ordering relation; hence, as a substitute, we defined a *partial* ordering relation, namely the refinement ordering. This gives our repository a lattice-like structure, where nodes represent specifications; and programs are attached to specifications in such a way that each program is attached to the highest specification with respect to which it is correct. This integrity constraint must be maintained by all database operations. As an example of a database structure, see in figure 1 the database obtained by storing a number of Pascal compilers, whose definition is given in [2]. Each node in this graph represents a specification; to each node we attach programs that are correct with respect to the specification that this node represents (these are not shown in the figure).

Storage and retrieval operations consist of navigating through a graph structure, matching the (storage or retrieval) key against nodes of the graph. Each node of the graph is represented by a *Unix* file which contains an *Otter* formula that defines the specification at hand, as well as information on neighboring nodes in the graph, and information on programs that are attached to this node.

The navigation through the graph is carried out by traditional graph traversal algorithms —which we have implemented in *C* under *Unix*. The matching of the (storage or retrieval) key against the current node is carried out in two steps: first we generate an *Otter* theorem that states that the key is a refinement of the current node; then we submit this theorem to the *Otter* theorem prover and await its result. The theorem prover and our *C* program interact to determine the result and act accordingly.

Whenever exact retrieval fails, the user has the option of invoking approximate retrieval. Approximate retrieval is defined in terms of a measure of functional proximity between specifications, which measures how close two specifications are by assessing how much information they have in

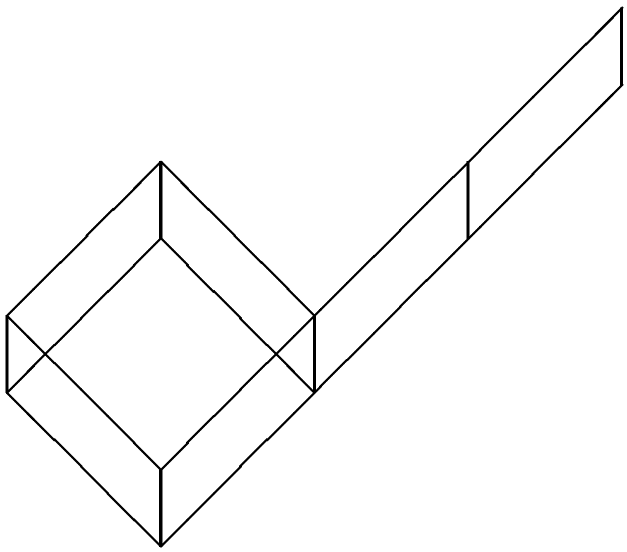

Figure 1: A Repository of Pascal Compilers

common. We have determined that a lattice operator, namely the meet (greatest lower bound) captures precisely the amount of information that two specifications have in common. The algorithm of approximate retrieval for a given retrieval key k proceeds by computing the meet of k with nodes in the graph, and identifying those noodes that maximize the meet with k (i.e. that have most common information with k). In the compiler example, the graph that we obtain by computing the meet of all nodes with k, for some retrieval k which is given in [2], is shown in figure 2. Note that this is obtained from the original graph by collapsing its two layers: whenever two nodes are collapsed, we can conclude that they differ by functional properties that are irrelevant as far as k is concerned; i.e. as far as k is concerned, one node is as good as another. In practice, this algorithm has given results that are quite satisfactory to the intuition.

3 Comparison

Our work deals with the problem of storing and retrieving software components using a formal specification approach. As such, it must be compared with alternative methods of storage and retrieval of software components as well as alternative methods of software reuse based on formal specifications.

Alternative methods of storage and retrieval of software components include the facetted approach, due to Prieto-Diaz and Freeman [8], the linguistic approach, due to Hall et al [9], and the cognitive approach, due to Maiden and Sutcliffe [10]. All these approaches are inspired from retrieval techniques of library science, and carry the usual drawbacks of natural language methods.

Alternative methods of using formal specifications for the purpose of software reuse include work

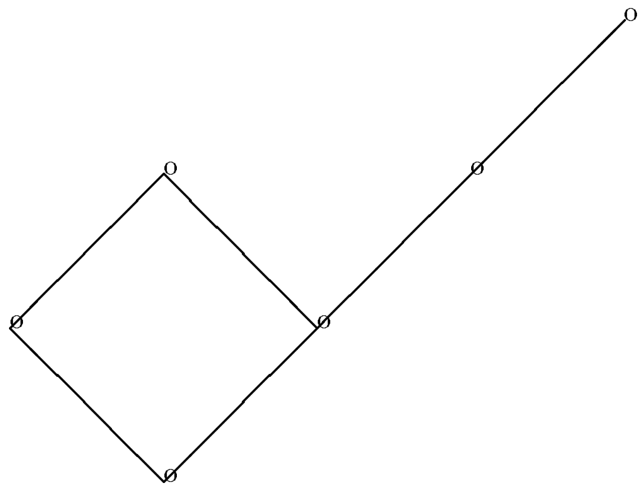

Figure 2: Ordering Nodes by their Proximity to k.

by Moineau and Gaudel [11] as well as work by Srinivas and Goldberg [12]. We differ from both of these proposals by the formal specifications background that we use: ours is relational, dealing primarily with functional descriptions of software components, whereas the alternative proposals use algerbraic specifications, dealing with descriptions of software components as state bearing modules. Furthermore we differ from the work of Srinivas by the fact that we concentrate on reusing software *products*, whereas they focus on reusing software *processes* (e.g. previously recorded design scenarii).

References

[1] R. M. A. Mili and R. Mittermeir, "A formal approach to software reuse: Design and implementation," in *Proceedings, Fifth Workshop on Software Reuse*, 1992.

[2] A. M. N. Boudriga and R. Mittermeir, "Semantic-based software retrieval to support rapid prototyping," *Structured Programming*, vol. 13, pp. 109–127, 1992.

[3] P. Collins, "Considering corporate culture in institutionalizing reuse," in *Proceedings, Fifth Workshop on Software Reuse*, 1992.

[4] S. Fraser, "Reuse by design- a team approach," in *Proceedings, Fifth Workshop on Software Reuse*, 1992.

[5] M. Griss, "A multi-disciplinary software reuse research program," in *Proceedings, Fifth Workshop on Software Reuse*, 1992.

[6] K. Wentzel, "Software reuse- it's a business," in *Proceedings, Fifth Workshop on Software Reuse*, 1992.

[7] J. Guttag and J. Horning, *Larch: Languages and Tools for Formal Specification.* New York, NY: Springer Verlag, 1993.

[8] R. Prieto-Diaz and P. Freeman, “Classifying software for reusability,” *IEEE Software*, vol. 4, no. 1, pp. 6–16, 1987.

[9] C. B. P.A.V. Hall and J. Zhang, “Practitioner: Pragmatic support for the reuse use of concepts in existing software,” in *Software Reuse, Utrecht 1989* (L. Dusink and P. Hall, eds.), pp. 97–108, Springer-Verlag, 1991.

[10] N. Maiden and A. Sutcliffe, “Reuse of analogous specifications during requirements analysis,” in *Proceedings, sixth International Workshop on Software Specification and Design*, (Como, Italy), pp. 220–223, 1991.

[11] T. Moineau and M. Gaudel, “Software reusability through formal specifications,” in *Proceedings, First International Workshop on Software Reusability* (J. C. R. Prieto-Diaz, W. Schaefer and S. Wolf, eds.), no. Memo Nr 57 in UniDo, (Dortmund), 1991.

[12] Y. Srinivas and A. Goldberg, “Replay of derivation histories in kids,” in *Proceedings, Fifth Workshop on Software Reuse*, 1992.

4 Biography

Roland Mittermeir is professor at the Institut für Informatik at the Universität Klagenfurt, Austria. Among his early work on software reuse is the design of the "Software Base"-Concept, now implemented in the AUGUSTA environment. Recently, he has also worked on implementing reuse concepts in medium sized software companies. Prof. Mittermeir holds degrees from the Wirtschaftsuniversität Wien and the Technische Universität Wien.

Rym Mili holds a *doctorate of specialty* from the University of Tunis, Tunisia, and is currently working towards a PhD at the University of Ottawa under the supervision of Professor Robert L. Probert. Her research interests are software specifications, software testing, and software reuse.

Ali Mili holds a PhD in computer science from the University of Illinois at Urbana Champaign. His research interests are software specifications, program construction and software reuse.