Classifying Components by Behavioral Abstraction

Roland T. Mittermeir, Heinz Pozewaunig
Institut für Informatik-Systeme, Universität Klagenfurt
Klagenfurt, Austria
email: {mittermeir, hepo}@ifi.uni-klu.ac.at

Abstract

We present an approach for classifying and retrieving reusable software based on exemplary descriptions of its functionality. The functionality of reusable components is described by a set of tuples with each tuple representing a characteristic input-output (or stimulus-response) transformation of the component at hand. With careful choice of these characteristic tuples, information equivalent to conventional formal specifications is given, except that (re-)users not versed in formal specification can provide such queries.

The concept just introduced depends on the discriminative potential of the tuples provided and on the equivalence of behavior representations by the librarian and by the (re-)user. The former can be assured by the librarians professionalism, the latter by an adequate user-interface of the system supporting a dialog, based on what the system has to offer rather than on guesses about what the (re-)user is looking for.

1 Motivation

We depart from the assumption that in black box reuse the user is not interested in retrieving a particular piece of code from a library of reusable components, but that his/her interest is the functionality needed. This functionality can be – and usually is – described by natural language descriptors or, on a more detailed level, by formal specifications. However, the former are in general too weak to select an appropriate component, and the latter are in general too complicated to write down for the requester.

Based on the analogy of checking out books from open-stock libraries where catalogs provide just rough guidance and the final decision of the reader is made after browsing through some books suitable for the question at hand, we bank on the operational semantics of software. In contrast to [1] we do not use randomly generated data. We rather focus on discriminatory tuples.

2 Specifying by Tuples

2.1 Preliminaries

As discussed in [2] we have to distinguish between the (re-)users intent and the specific representational form used to describe this intent. Likewise, the component sought can be described at various levels of abstraction and in various representational forms.

Considering the component itself, three broad categories stick out for its description:

- *Textual algorithmic description:* This is the conventional description of a component's source code, written in some compilable programming language. In general we may assume this textual description to be procedural code, ready for execution after compilation/interpretation.
- *Predicative (intensional) description:* This refers to formal specifications, irrespective of the language used. While specifications are text too, they are not linear text but a set of predicates to be interpreted by some reasoning mechanism irrespective of the order in which they are written down.
- *Tabular (extensional) description:* Whatever a program might compute are input-output-transformations of some sort. Hence, instead of describing these mechanisms by means of indicating their semantics (specification) or how they are to be derived (program) one could also explicitly write them down in a (potentially infinite) table.

These categories to describe functionality apply for both, the provider and the requester of a component. Here, we depart from the assumption, that in general, it is easier to give some characteristic input-output mappings of a functionality needed than to describe it by writing its complete specification or its complete program. We also depart from the assumption, that it is easier to read a specification (or even a program) than to write it. Therefore, we propose a method where requesters of a reusable component select this component from a repository on the basis of characteristic input-output-tuples.

For this method to be sound, we assume that each component stored in the library is represented not only in its source-code representation, but that it also has a formal specification as well as a set of such characteristic tuples attached. To assume existence of a formal specification seems fair when dealing with high quality components. The effort spent in building it will pay off during high frequency reuse later. Requiring tuples attached is specific to this approach. The details of how these characteristic tuples are built in the first place and how the correspondence between tuples considered characteristic by the requester and those considered characteristic by the librarian are established is subject of the following sections.

2.2 Characteristic tuples

A specification defines a relation between input and output of a component (if needed, state information can be externalized). This relation can also be given in extensional form. As such, a component's specification can be seen as the universal relation defined over the cross product of all types figuring in its signature. This universal relation has an obvious partition into those tuples that do conform to the specification (belong to its domain) and those tuples that have their input-part in those portions of the "from-set" that do not belong to the domain covered by the specification. Within the domain, further partitioning of the set of tuples conforming to the specification can be made, so that each of these partitions can be characterized by some characteristic tuple.

If this approach seems too simple minded, one should note that domain partition testing rests on exactly this principle to distinguish between software that conforms to its specification and faulty software, i.e. software not conforming to this specification. At this point, it seems arbitrary, which tuple is used to characterize its particular sub-space. Testing theory shows, that there are particular positions (e.g. boundary values) that have particular discriminative power. We might follow this argument, specifically if we do not want to recreate those tuples but just rely on those already provided by the test suite a component had to pass. We will return to this issue with further arguments in section 4.4 where we focus on the discriminative power of tuples in the search process.

Not all test cases used for distinguishing correctly behaving components from other ones suite our purpose. In figure 1 tuples for testing the step function are depicted as diamonds and circles. Whereas all tuples are valid test cases, albeit only diamonds are suitable to distinguish between the step-function and the exponential one, which is unfortunately also characterized by the circle tuples. As "good" discriminating tuples discriminate in the library context as well as in the testing context, black-box test data can be used to identify such semantical hot-spots and no particular knowledge of the library structure is needed at first glance. For later tuning of the library structure, the librarian can rely on the executability of software.

3 Obstacles

While the concepts just described seem to be adequate to describe software from the librarians (component provider) perspective, it seems problematic from the requesters view. In order to obtain a match between the characteristic tuples provided by the librarian, the requester would have either to "guess" the appropriate tuples or to write his own specification of the system from which discriminatory test data could be derived. Both arguments seem to defeat the aim of the whole venture.

To overcome these obstacles, one can either fall back on the executability of software [1, 3] or reverse the role of requester and system. Since we assume that some descriptive methods have been used already for coarse screening of the library, the requester now needs to zero in only on a rather limited set of eligible components.

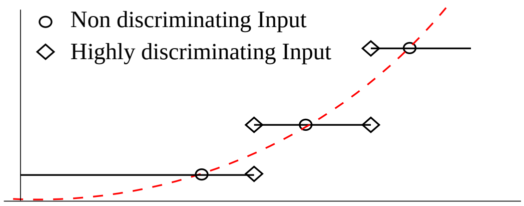

Figure 1: Quality of discriminating input

For them, the number of discriminating tuples will usually not be too excessive. Since we further assume, that (most of) the discriminative tuples have been provided already from the test suite and that the test suite has been established with professional care, we rather want to bank on this asset. If these tuples are organized in such a way that the discriminatory tuples are presented on a step by step basis such that the requester can walk down a path on a search-tree (actually: a lattice) of available components, the approach becomes feasible.

In doing so, contrarily to behavior sampling, our concept changes the role of the requester. Now, not he is responsible for providing the system with meaningful example values, but the system itself presents interactively characterizing tuples. In choosing the behavior which solves the problem in question best, the requester specifies the component incrementally.

This incremental specification of a component is not to be confused with a full fledged specification one would write for somebody to build this component from scratch. It serves rather to distinguish a given component (or set of components) from other components stored in the library but not conforming to the requesters specification. In general these tuples should suffice. However, based on the instrumentation of the library, the requester can always try to test-execute the component with further data of his/her choice.

4 Browsing Behavioral Descriptions

4.1 Principles

In general, libraries (and software libraries too) are organized according to some hierarchical schema. Whether this is keyword based [4], or whether there is a lattice of formal specifications [5], seems immaterial. Hence, to avoid an information overkill of the requester, behavioral abstractions presented by the system have also to be organized in some hierarchical way. As we have indicated already, we presume that the following considerations take place only after some semantic based, coarse description has been applied. Only the integrated cooperation of orthogonal concepts supports the reuser in a promising and satisfying manner.

On this base, specifically in fine grained search, a reuse library is primarily divided into segments Σ. The separation criterion is that the components belong to a common problem domain and have an equivalent (generalized) signature [6, 7, 8]. The equivalent interfaces of components belonging to the same segment are needed in order to ensure that their extensional specification is a subset of the universal relation for this particular signature. On the other hand, there may exist other segments, specified by the same signature, but attached with other semantic attributes (Statistic components and financial components should reside in different segments). Within these segments all components are structured due to the refinement criterion provided by the given tuples.

We assume that the set $\mathcal{C}_\Sigma$ is the set of all components in a segment Σ. A candidate component $c \in \mathcal{C}_\Sigma$ is a collection of one or more algorithms a with identical functionality (E. g., c represents all ascending sorting algorithms on character lists).

At this point we want to stress, that a certain algorithm a can also be related to two or more components! Consider sorting algorithms, which are in general implemented as parameterized programs accepting input of different data types, provided with an ordering relation for these types. Therefore, such algorithms are related to different components in different segments according to their respective signature.

Furthermore, $\mathcal{T}_\Sigma$ is the set of all tuples in a segment Σ, thus all example tuples $t \in \mathcal{T}_\Sigma$ are in accordance to the interface specification of Σ. The power-set of $\mathcal{T}_\Sigma$ is denominated as $\Pi(\mathcal{T}_\Sigma)$, and respectively, the power-set of $\mathcal{C}_\Sigma$ is denoted as $\Pi(\mathcal{C}_\Sigma)$. For the sake of simplicity, in the following discussion we do not mention the subscript for the current segment, Σ, if the meaning is clear.

A *component retrieval function* is defined as a function $cr : \Pi(\mathcal{T}) \to \Pi(\mathcal{C})$.Provided with a set of tuples as example behavior, the result of *cr* is a set of components C. Appropriately enlarging T will increase its discriminative power, thus narrowing the cardinality of C. Thus, an iterative scheme is described, so that in the end, the result of *cr* will be a unitary set C, containing a single component $c \in C$ as result of the search process. The pragmatics of *cr* is described as follows:

If a subset of $T \in \mathcal{T}$ denotes a subset of components $C \in \mathcal{C}$, we say that T *characterizes* C. The empty set characterizes every component in Σ, since the empty set has no discriminative power. Also, if the cardinality of C is 1 and there exists no other $t \in \mathcal{T}$ as example of the sole component c in the set C, we say that T *fully characterizes* C (in fact T characterizes $c \in C$). In this sense every set T is part of a description and a partial specification for the components in C. If the cardinality of C is 0, then either there is at least one tuple t illegally added to the description set T (T contains contradictory entries, resp. there are contradictions on the search path), or the additional tuple t is a valid additional discriminator, but there is no component satisfying the specification refined by t. As long as the cardinality is greater than 1 further tuples must be added to raise the characterizing potential of T. Hence to decrease to the number of components in C, the number of tuples in T must be incremented. Also it must be considered, if there are more than one elements in the component set C and no more $t \in \mathcal{T}$ can be added to T, without (1) generating the empty set, or (2) decreasing the number of candidates, then the repository seems underspecified.

If such a situation occurs (meaning that no remaining tuple in $\mathcal{T} \setminus T$ adds describing capabilities to T), (1) the components remaining in C are indistinguishable in the sense of input-output behavior, or (2) important discriminating tuples are missing in $\mathcal{T}$.

4.2 Search Hierarchy

In order to provide an efficient search strategy, the tuples t uniquely identifying one c have to be stratified. This stratification uses the fact, that the various sets T_i describing component specifications C_i are in general not disjunct. The overlap between these sets can be used to build a search hierarchy. To define this hierarchy, we note, that the unpartitioned set $\mathcal{C}$ will be retrieved by the empty set of candidate tuples, $T^0 = \emptyset$.

Adding one (or several) tuples to T^0 raises its discriminative power, so that $\mathcal{C}$ will be split into (disjunct) subsets C_i^1 with each C_i^1 containing components c conforming to T_i^1 with $T_i^1 \subset T^1$. Further refinement of T_i^1 by adding more tuples $t \in \mathcal{T}$ further raises its discriminative power. Let's refer to those additional tuples by τ_i^1, then $T_i^2 = T_i^1 \cup \tau_i^1$ and T_i^2 assumes on C_i the same role T^1 assumed on the full set of not discriminated components C. The result of this discrimination will be component-sets C_{ij}^2 with each component in C_{ij}^2 conforming to T_{ij}^2. The reader can immediately verify, that this describes level-sets in a tree and the procedure sketched for the levels next to the root can be iteratively expanded till the leaf nodes of the tree are reached. Hence, we will focus in the further discussion just on the immediate parent child relationship in this search tree. As the τ_i^l are the sets leading to further discrimination within a given set C_i, the tuple-sets T_{ij}^l are distinct only in those parts they obtain distinct to T_{ij}^{l-1} from τ_i^l. One might also note, that the set leading to the initial split, T^1 could as well be referred to as τ^0 (seen from a procedural point).

We call a pair (C_i, T_i) a candidate κ_i. For a candidate the set T_i characterizes all elements of C_i. Within the tree structure thus established, the following relations must hold between a father candidate p and its n successors s_i, $1 \leq i \leq n$:

- $(\bigcup_{i=1}^n C_{s_i}) = C_p$ means, that the component-set of all direct offsprings build the component set of the father candidate.
- $\forall c, \forall i \neq j : (c \in C_{s_i} \Rightarrow c \notin C_{s_j})$ means, that no offspring candidate shares a component with its sibling. An direct implication of this postulation is $\bigcap_{i=1}^n C_{s_i} = \emptyset$.
- $\bigcap_{i=1}^n T_{s_i} = T_p$ means, that every successor adds one or more new input-output tuples to its example value set T_s, which helps to distinguish the component set from its father component set.
- $\forall t, \forall i, j : (t \notin T_p \wedge t \in T_{s_i} \wedge t \in T_{s_j} \Rightarrow i = j)$ means, that the items in the tuple-set of a candidate but not in the tuple-set of the ancestor of this candidate have key-property. The reader might note, that these are tuples taken from the set of additionally discriminating tuples τ_p.
- $\bigcup_{i=0}^n \tau_{k_1 \dots k_n}^i = T^n$, thus, the tuples collected over those parts of the various incremental discriminatory sets on the path from the root of the search tree to the candidate at hand form exactly the set of discriminatory tuples of this candidate.

4.3 An example

Here the signature f : `char` $\rightarrow$ `bool` designates a segment containing the following component set $\mathcal{C}$ from the ANSI C-standard library (modified for readability):

```
1  bool isalnum(char c);    7  bool islower(char c);
2  bool isalpha(char c);    8  bool isupper(char c);
3  bool iscntrl(char c);    9  bool ispunct(char c);
4  bool isdigit(char c);   10  bool isspace(char c);
5  bool isgraph(char c);   11  bool isxdigit(char c);
6  bool isprint(char c);
```

The requester searches for a predicate checking if a given character is a hexadecimal digit. Initially, the set $\mathcal{C} = C$ contains all eleven components of the segment, whereas the tuple set T is empty. The system offers two options to the requester, τ_1^0 =`('0', T)` and τ_2^0 =`('0', F)`. Obviously, the latter option does not meet the demand. Therefore, `('0', T)` is added to the initial empty set T and the component set C is reduced by 2, 3, 7, 8, 9, 10, generating the candidate $\kappa_1 = (\{1, 4, 5, 6, 11\}, ('0', T))$. . In the next step the system offers the examples `('a', T)` and `('a', F)`, once more with the first example as correct offer. Hence, function `isdigit` should be excluded by the searcher and all other functions are added to C_2. In the next iteration (examples `('Z', T)`, resp. `('Z', F)`) the choice of the second alternative clearly identifies the component `isxdigit`, which provides the effect the reuser wanted.

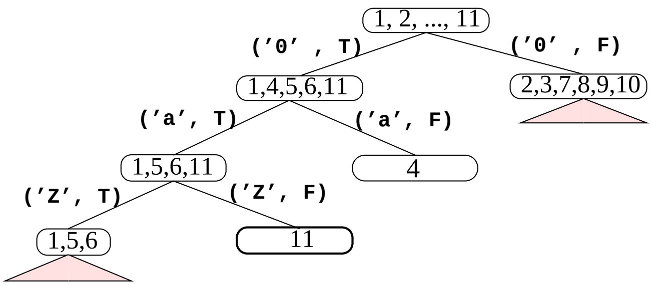

Figure 2: Classification Tree

Figure 2 shows the tree structure of this example in detail. The set C for every candidate is depicted as rectangular, whereas the set T is build as union of all tuples from the current candidate back to the root of the classification tree. After three iterations in interacting with the system, the reuser identifies the component in question without the need for specifying the semantics exactly. The resulting candidate is $(\{11\}, \{$`('0', T)`, `('a', T)`, `('Z', F)`$\})$ which is indeed an element of the component retrieval function cr.

4.4 Discussion

The illustrative example has been kept very simple from many perspectives. One simplistic assumption was to use just binary partitions based on simple tuples. In general, the discriminative set will consist of more than a single tuple, and thus, the librarian has to make a conscious judgment in optimizing depth and breadth of the search tree. This decision will involve the selection of particular tuples in the overlapping areas of characterizing tuples (T_p, if considered from a bottom-up perspective). This bottom-up perspective will be necessary, when considering maintenance of the library by including new candidates to a library defined and in use already.

The librarian, in contrast to the test expert, might also be worried about the executional cost of a tuple. Tuples should not only be selective, they should also be easy to compute and to verify. This argument holds because even if the repository has pre-stored results, the requester has to be able to verify that he/she actually wants the results stipulated by the various discriminatory tuples presented at a given level.

A further consideration, beyond the scope of this paper, is treatment of interactive software or software encapsulating state information. Here, the concept of precanned tuples as presented here, does not hold. The general principle of the ideas presented in this paper do apply though.

5 Conclusion

An approach to classify software on the basis of tuples, representing snapshots of its "petrified" behavior has been presented. The approach aims to allow for software description and software retrieval "by example".

References

[1] A. Podgurski and L. Pierce. Retrieving Reusable Software by Sampling Behavior. *ACM Transactions on Software Engineering and Methodology*, July 1993.

[2] R. T. Mittermeir, H. Pozewaunig, A. Mili, and R. Mili. Uncertainty Aspects in Component Retrieval. In 7^{th} *Int. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems – IPMU98*, pages 564–571, July 1998.

[3] R. J. Hall. Generalized Behaviour-based Retrieval. In *Int. Conf. on Software Engineering – ICSE93*, Baltimore, MD, May 1993. IEEE Computer Society Press.

[4] W. B. Frakes and P. B. Gandel. Representing Reusable Software. *Information and Software Technology*, 32(10):653 – 663, December 1990.

[5] R. Mili, A. Mili, and R.T. Mittermeir. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions On Software Engineering*, 23(7):445 – 460, July 1997.

[6] A. Moormann Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333 – 369, October 1997.

[7] R. T. Mittermeir and E. Kofler. Layered Specifications to Support Reusability and Integrability. *Journal of Systems Integration*, 3(3/4):273–302, sep 1993.

[8] Y. Chen and B. H. C. Cheng. Formalizing and Automating Component Reuse. In 9^{th} *Int. Conf. on Tools with Artificial Intelligence – TAI 97*, pages 94 – 101, Newport Beach, California, November 1997.