Self-descriptive Software Components

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

Abstract

Many technical problems involved with Component Based Software Development might be considered solved these days. Less so are mental reservations both, developers as well as their managers have with respect to the risks involved with building software from powerful building blocks.

This paper addresses the issue of trust developers might extend towards components. Besides psychological aspects this issue has a technical aspect. We will focus on the latter. After a brief review of mechanisms to describe software components and the relationships between descriptions and certifications, the paper zooms in on "self-descriptive" methods, i.e. methods directly involved with the executional properties of software.

The approach presented in detail is tuned towards fine-grained distinctions. It lends itself particularly to questions of version discrimination and software maintenance.

1 Motivation

My great-grandfathers horse-cart found its way home, even when the old man was too tired to direct the horses. My car will kill me, if I fall asleep on the highway. - Can we build software that resembles rather my great-grandfathers horses than my car?

We increasingly rely on complex technology in spite of the apparent deficiencies highlighted in the above sentence. Is there a chance to bring some of the reliability of these old systems back to modern technology, specifically to software, a technical artifact that is occasionally claimed to be "artificially intelligent"?

When aiming at such goals, we have first to analyze some of the differences between the system consisting of horse, cart, and driver as opposed to the system consisting of car and driver. Since we want to abstract from the driver and since the material the wagon or the car respectively are made of does not really contribute to this difference, the distinction has to be found in the horse. The car is a system without a *self*[1]. The combination horse and cart, however, has in the horse-part of this system at least some limited capability of recognizing its self as well as - in this example - the location of its self as opposed to the location where it ought to be in the evening.

We better don't stress this analogy to a point were it breaks. However, we might at least agree that from a reliability perspective, a component knowing about its self has merits over a component that is just there without such knowledge. Let us pause at the proverb "*To a person with a nail, any tool looks like a hammer*". This proverb implicitly assumes that the tool has no knowledge about its self. Hence, either a wrench or a pair of pincers might be used as hammer, even if they break during the operation. Lacking a self, they lack any capability of protecting themselves against misuse. Likewise they lack any capability of protecting their application environment against them being applied in an abusing manner.

In the rest of this paper we will focus on the issue to which extent information inherent in a component (a tool) can be externalized in such a way that it becomes instrumental in the software development process. In order to do so, we next focus on notions of external descriptions versus self-description of software components. Based on these reflections, we propose in section 3 the technique SbS which exploits the behavior of software components directly without taking the detour of externally added descriptions. While section 3 still focusses on software Reuse, section 4 highlights how SbS can be used to support other software engineering tasks.

2 Self-descriptiveness of Software

How about the *self* of software? Software is an artifact. Hence, it cannot have the kind of self we find with the horse or with other animals around us. It has not even a physical substance. However,

[1] Some psychologists might claim that the notion of *self*, specifically the self-awareness, is a unique property of humans. They characterize it as "the" major discovery during childhood. We are not going to argue about that in detail. But at least certain social behavior of animals makes us to believe that animals are capable of distinguishing between themselves and their peers in the cohort. Hence, some limited concept of self seems to exist.

in the context of semantic database research, the issue of self descriptive data has already been investigated [Roussopoulos, 1982]. In spite of the plethora of work on documentation and software comprehension, we are not aware that similar approaches have been followed in software engineering. Even the notion of conceiving software as just another form of highly structured data [Mittermeir and Oppitz, 1987; Mittermeir, 2001] did not bring us to the point of pursuing this as a research issue on its own. But in software reuse, the question as to whether the component someone finds in a repository actually is the component some external description claims it to be, is central to the overall success of large scale planned reuse.

As pointed out by Tracz [Tracz, 1988] developers will rely on reusable components only, if they can rest assured that reusing is not harming their success in the long run. Put in other words, they will use a component only, if they can trust that it satisfies the expectations put into this component, i.e. it behaves as described.

This brings us back to the question of how to describe reusable components. An extensive survey about how to describe reusable software [Mili *et al.*, 1998] lead us to conclude that most of the approaches discussed there are too weak to achieve both, high recall and high precision. This is not to be blamed on the methods analyzed there. As shown in [Mittermeir *et al.*, 1998] it is due to the inherent problem of describing software by abstractions, abstractions that might be based on abstraction principles different with the producer/librarian and re-user. We might interpret the advice that successful reuse is better in-house reuse or at least domain dependent reuse as an outgrow of this problem. In in-house reuse, we can walk to the colleague who produced the module and ask her/him about this piece. Then, the trust we confer to this colleague is transferred to his artifact, to the component.

In our recent research, we split the problem. The issue of adequately describing a component by a shallow analysis of verbal documentation and using its compressed form was pursued by Bouchachia [Bouchachia, 2001]. This of course falls short of the precision obtained by describing components by means of functional specifications [Mili *et al.*, 1997]. However, Bouchachia's method can capture more dimensions than just the functionality and it is established at far lesser expense. The precision lacking with comparing natural language texts is brought in at a later stage by describing components by behavioral specifications [Pozewaunig, 2001]. These behavioral descriptions are basically suitable for fine grained search. Hence, some prior screening has to sufficiently narrow the search space. In software retrieval, this might be considering a limiting factor to be compensated by coarser means for narrowing down the search space. However, before we lose focus in delving more deeply into the process of software retrieval, we better come back to the question of trust from another vantage point.

With any external description or specification of software, the re-user has to trust that the component behaves as specified. In the case of a function `sin(x)` it takes only little effort to check whether it is a trigonometrical function and whether it actually computes the sine or whether, by mistake, somebody had the wrong conceptual model and the function actually computes the cosine instead. In case of more complex components, such a check might be much more involved and certain flaws in the component might eventually go unnoticed. Hence, external descriptions on higher levels of abstraction are not more than well-meaning hints that the code most probably will live up to what the specification promises.

Behavioral descriptions do not share this deficiency. They consist of data tuples describing actual executional behavior of the component under consideration (see section 3.1). Thus, in order to check whether the description is faithful with respect to the component it describes, we just have to run it on the input part of the tuple and see whether the execution matches the tuples' output part. At least for functional components, the issue of trust in the correspondence between (external) description and the inherent description of the codes own functionality can be solved as simple as shown. For state-bearing components, which compute their output on their input and internal states, the issue is a little more involved, but still soluble [Pozewaunig, 2001].

Why can behavioral specifications overcome the problem of trust as posed above? They are just data-points linking input and output space of a computation. Thus, their level of abstraction is even below the level of abstraction of code. However, with these data-points we are close to some notion of *self* inherent in the code. Of course, it does not possess the self we witnessed with the great-grandfathers horse. However, code encompasses all the computations that fall in its domain.Since functionality can be described by predicates, by data tables, or by code alike. Relational specifications rest particularly on this representation invariance. Thus, attaching executional traces or tuples emanating from the formal testing process as descriptive means to software does not add information, it just serves to

- raise descriptive efficiency: The component does not need to be executed, the result of the execution is readily given already.
- point to interesting spots: As we know from testing theory, most input-output combinations can be sufficiently covered by just a few characteristic tuples, lying close to the border of sub-domains the component is specified for. For the remaining data points the components behavior will be analogous. Hence, these points are inherently uninteresting.

Among those arguments, apparently the second one is crucial, since it leads the re-user to those points where a component's behavior is critical. It seems fair that both, the producer and the re-user of a component agree on which data points defined over domain and range of a component are critical. If the producer missed some of them (from the re-users) perspective, the re-user can "ask" the component by executing it with the respective input vector. If the re-user missed some of them, that's too bad. However, these points would also have been missed in development from scratch.

Another important argument stems from the process of gathering descriptions which is performed by executing the component and analyzing the results. Thus, producing trustworthy descriptions is not a task of a human but of a specialized algorithm and it can be repeated at nearly no cost over and over again.

When exploiting behavior, which manifests itself in the form of test data, an accurate classification can be obtained, which is not blurred by natural language vagueness and/or the biases of humans describing the *self* of a piece of software.

3 Data Transformations as Knowledge Base

To bring effective behavioral descriptions to life, we extend the path which was already laid by the technique of behavior sampling of Podgurski and Pierce [Podgurski and Pierce, 1993]. In their work, they developed the idea to retrieve software by feeding examples of behavior to a retrieval system. In a first step, sample behavior in the form of input-output tuples are specified by the searcher. Next, the retrieval system executes its stored components and present those showing exactly the functionality specified by the example.In that way no specialized query language is needed to query the software repository and the components are described by a demonstration of a part of their behavior. The query is expressed directly in the context of the domain, thus, a software engineer must not switch from her/his mind-set to another one to access a repository.

One remark to that idea: Also for formal specifications, especially for relational specifications, data is already considered as a means to describe software, albeit on an abstract level. Relational specifications [Mili *et al.*, 1997] describe the behavior of a component by means of relations between input domain and output range. Hence, the idea to take data tuples as component descriptors, is not entirely new.

Some of the drawbacks coming with this approach are the problems when executing the components on the presented input. A ready-to-run executional environment for each component of the repository is mandatory and in the case of a long living repository many versions of different execution environments must be maintained then. In addition, an interaction with the retrieval system may render difficult, since one query-answer cycle is very time consuming, depending on the number of available components conforming to the signature as specified by the example data. A further drawback is the demand to specify exactly the behavior of a component in advance (which includes the provision of the correct signature). If this information is not known in advance a search becomes inevitably cumbersome or impossible, although a functionality as needed is available in the repository. As a direct consequence, browsing a repository is not supported. However, browsing is seen as an important means to learn about styles and already available solutions for a certain domain.

3.1 The SbS approach to description generation

In this section we present a description technique based on behavior sampling. We refer to this technique as *static behavior sampling*, for short SBS.

Whereas behavior sampling relies on the on-line execution of components, SBS calls on historical data about the component's executions. The majority of this data is provided by test data. From test data much information about characteristic behavior of components can be infered without having to analyze or execute the component directly.

In this work we focus on state-less components only, which are seen as functions (however complex) which produce deterministically output on input data. Such a pair of input and output data is called *data tuple*. Since both input and output may comprise many (perhaps complex) data elements, a data tuple dp is a pair of two vectors: one for the input, one for the output; $dp = (\vec{i}, \vec{o})$. Those tuples used for the purpose of characterizing software components solely are named *data points*.

Test data is aimed at revealing faults in software; the aim of testing is not to describe software. Consequently, when considering the input domain to a component containing equivalence classes, one member of that class suffices to test that class. Any input data selected from the same class is considered as a waste of time and resources. Because SBS has other aims, pure test data is not sufficient as a basis for high quality descriptions. The consumer of the descriptions finally is a human being who must be supported in understanding the representation. Hence, the description base must be enhanced by *striking samples* which are tuples immediately recognizable by domain experts to be produced by a certain functionality.

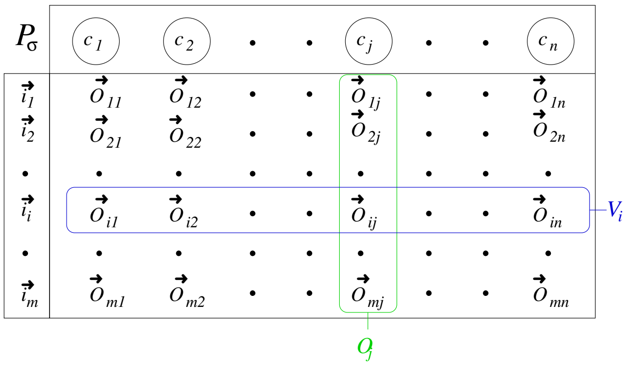

Figure 1: A partition P_σ in a SBS-repository

The core of SBS is a component repository containing components as well as their data points. The coarse grain organization of the repository is determined by the signatures of components which allow to put components with the same signature σ to one partition P_σ. To obtain large partitions, signatures are abstracted to gain generalized signatures which hide implementation dependent details without loosing the conceptual idea behind them [Pozewaunig, 2001]. One single partition contains all components and their data points. Data points are selected (or generated) such that each component is executed with all inputs. Hence, an input vector v_i represents an index entry referring to a set of output vectors V_i. Data points are chosen intelligently such that we can establish a function l to localize a component c_j, when input and output are provided: $l(\vec{i_i}, \vec{o_{ij}}) \to c_j$. The set of all outputs of a certain component then is refered to as O_j. This structure is shown in detail in figure 1. As one can see, the partition is *complete*, which means that there is no component from which we do not know its output given an existing input. Although completeness is highly recommended, it is not obligatory as long as the descriptive power (measured in terms of the function l) of the given data points do not suffer from missing values.

A SbS partition groups components on the basis of their technical structure, their signatures. These components share at least two important properties:

1. They demand (nearly) the same data types as input and deliver similar typed output structures.
2. They all can be distinguished on the available input data provided with the data points.

To locate a component in the repository it is sufficient to distinguish it from all other available ones. Clearly, localization is nearly impossible with the flat representation of a σ–partition. Hence, we must analyze the available information and learn yet unknown classifiers from that. This task is called *supervised learning* and in the research field of machine learning many techniques are proposed. We chose the approach of classifying by inducing decision trees, because the resulting structure is rather intuitive, the algorithms do not need any extra domain knowledge, the result's accuracy is at least equal to all other comparable algorithms, and they are fast [Ganti *et al.*, 1999].

The analysis itself can be easily stated in terms of a machine learning problem. The goal is to construct a hierarchical structure which finally leads to one single component. In each hierarchical step the number of candidate components must be reduced. Since we know each component (in terms of machine learning a component is equivalent to a class) in advance, the term *supervised learning* is evident. In that way, each component is described by a set of attributes and their values. Attributes in our domain are given by input vectors; their output vectors represent the attribute's values. With this view onto the problem, it is simple to derive correct classifiers for components.

Consider the simple example of table 1 where 7 string predicates taken from the ANSI C standard library are classified with the decision tree algorithm C5.0, developed by [Quinlan, 1993].

Due to the simple domain it is sufficient to select the data points wholly from the available test data. The

Samples	isSubstring	isPrefix	isEqual	isGreater	isSmaller	isLonger	isShorter
<'a';'b'>	F	F	F	F	T	F	F
<'';'a'>	T	T	F	F	T	F	T
<'a';'ba'>	T	F	F	F	T	F	T
<'abc';'xy'>	F	F	F	F	T	T	F
<'';''>	T	T	T	F	F	F	F
<'a';'a'>	T	T	T	F	F	F	F

Table 1: A string predicate repository partition

C5.0 algorithm constructs the decision tree given in figure 2 from the available behavior base.

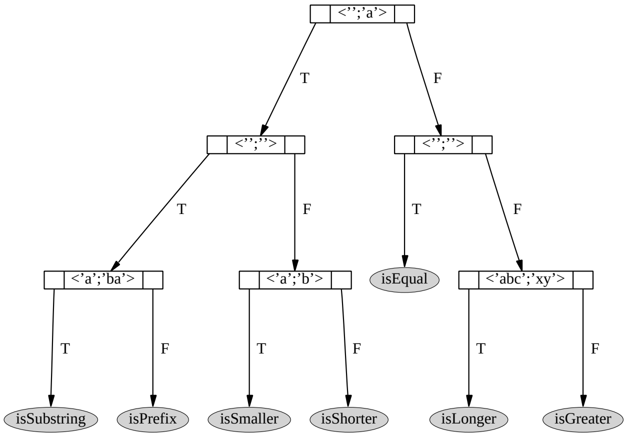

Figure 2: The characterizing structure for the string predicates of table 1

The decision tree contains a subset of the available data points which is sufficient to characterize a component completely with respect to all other components of the same repository. E.g. the description for the predicate `isSubstring` is given by the following sequence of data points: (<'';'a'>, <T>), (<'';''>, <T>), (<'a';'ba'>, <T>). No other component of that partition demonstrates the same behavior. More details about software description based on data points can be found in [Pozewaunig and Mittermeir, 2000; Pozewaunig, 2001].

4 Behavior-based descriptions to support Software Engineering Tasks

4.1 Maintenance and Configuration

The idea of SbS was initially developed to enrich descriptions for reusable software components. SbS-description can be generated automatically which renders this method practical for many other application areas within the field of software development. Especially for two fields an additional benefit for automatically provided descriptors can be recognized:

- Software Maintenance and
- Configuration Management.

If a maintainance operation results in a structural change of a system, this adaption of static elements is considered to be far reaching and normally it is reflected well in the documentation. However, if only minor changes in a system's functional dimension are affected, often the effort to adjust the documention is considered as too high [Pozewaunig and Rauner-Reithmayer, 2002]. In that way, a system's functionality drifts continually away from its documentation [Pirker, 2001]. With each minor adaptive maintenance task actualizing the documentation becomes more costly; costs which must be debited against a maintenance project's budget. Inevitably, this leads to a vicious circle where the tasks to adapt a system becomes more and more expensive, which leads to the effect to drop tasks with low priority, especially the task to update the documentation.

For configuration management the question is slightly different. The new versions emerging from a well understood base system normally are well documented concerning their novel functionality. But since here it must be ensured to keep the documentation consistent with the system, a tool for verifying this relation should be available.

This is the point where SbS enters the scene. In the previous chapter we demonstrated the base technique on classifying the components of a software repository. When maintaining software or producing new versions of a system, the partition does contain all previous versions of one (sub)system. Since all historical artifacts have been tested carefully, enough data for establishing a highly discriminating behavior base should be available. As we speak about versions of systems a commitment to quality can be taken for granted. An important task then is regression testing [Harrold, 1999] which aims at the verification that by adding new features or correcting faults nothing was destroyed. This task is very costly, but it produces a vast amount of new test data for the new version. This effect ensures SbS-completeness which is necessary for a high quality behavior base.

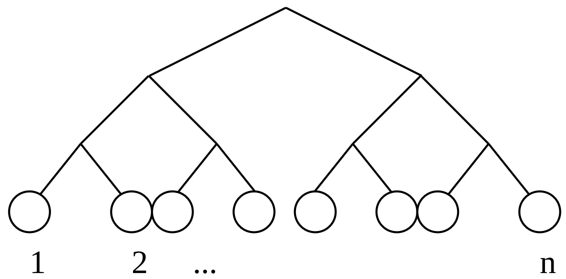

Figure 3: The SbS version tree for administering a components' versions

This situation is depicted in figure 3 where n different versions of one system (or one component) are administered. Each version is a distinct leaf in the decision tree. As a result of the SbS analyzing phase a version is identified by a characteristic difference to all other version maintained in the repository.

If a new version is released, then the most important fact is to state in what specific aspects it differs from the previous versions. To select new test tuples as data points for description purposes, it is necessary to get exactly those which are pointing at the changed functionality only. Obviously, these data points test the adapted parts of the system. But furthermore, they should be available for all previous versions, too. This requirement can be easily fulfilled by executing the systems with the current test cases and feeding the such generated data points into the SbS behavior base. The question now arises: How can data points be generated which focus mainly on the system's part which was affected by the adaption?

4.2 Gathering unerring data points

At a first sight this problem seems to be academic. Isn't it the task of software verification to test each release especially on new features? Why can't we take those test tuples for classification? Obviously, this is a feasible approach. However, one cannot be sure to get really test tuples touching the parts of the source code which were adapted. This is due to the fact that in structural testing (white box testing) without specialized tools a tester cannot systematically generate test data focusing on a certain spot in source code. Much worse is the situation in functional testing (white box testing) where domain boundaries are mostly defined by the requirements. Obviously functional testing can not guarantee to reveal characteristics which are valid for the current version only.

What helps us in this situation is the fact that here the direct access to source code is possible. This is not the case in a reuse situation where only binaries of components (COTS) are represented in a repository – eventually with their test data. Then the librarian responsible for correctly indexing components is not in the position to dig into the source code. In contrast to this reuse-situation, for maintenance or version development source code is accessible and it is a rich source of information.

The problem of getting unerring data points for characterizing software which was changed at a statement s can be restated as follows: "What input value v stimulates the execution of statement s?" This question is the same as stated for *program slicing.* Program slicing is a analysis and reverse engineering technique which reduces a program to those statements relevant for a particular computation. A particular computation is a reference to a variable v which is defined at a statement s. Since the invention of slicing techniques by Weiser [Weiser, 1984] many variations of the idea were published. Weiser's method was named *static slicing* because he analyzes source code and collects all statements which possibly affect a value v at statement s.. Furthermore, since Weiser's method computes all statements which affect a variable at a given statement it is called *backward slicing* (in contrast to *forward slicing* which computes all statements which might be influenced by a variable).

With backward slicing a tool is available which simplifies the search for unerring input data. It allows for generating exactly those values influencing statement s by concentrating solely on the slice.

4.3 Highlighting functional differences

A newly released version must be integrated into the version tree. Due to the assumption of conducted regression tests, the completeness of the SBS partition in question is ensured for the new version. Additionally completeness must be forced for the old version as well which was destroyed by introducing new tests necessary to verify the adapted functionality. If the old versions of the system are still operational in other contexts, the assumption to re-test them on the newly generated inputs is only fair. On the other hand, if old versions were really withdrawn, further keeping of such a version in the SBS repository must be questioned. All these assumptions allow to subjoin the new version without frictions.

The difference to the previous versions cannot be calculated absolutely, since only the most characterizing data tuple divergences are shown in the version tree. If there are many more differences, they are not shown in the tree directly. However, on a relative scale one can easily determine the distance between version v_i and v_j in calculating their distance in the version tree. In that way, one can state that, if the distance between v_i and v_j is e.g. 2, they are closer than a component v_k 3 units away. But one cannot conclude the amount of difference of two components which are equally far away from v_i because of the usage of the most discriminating data point for classification (neglecting other differences of data points).

5 Conclusion

We discussed an approach to describe components automatically on the basis of demonstrated behavior. Although at first sight an urge to shift from conventional natural language based description approaches seems to be overanxious, this is justified by the benefits of our approach. Due to the automatic generation of characterizing structures much more trust can be established for software which is classified impartially. The concordance between a component and its data manipulations is much more reliable than any relationship between a component and its keyword descriptions in the long term.

We do not claim that behavior based descriptions should substitute completely the task of characterising a component conventionally. However, when trust needs to be established in the real nature of a piece of software, the SBS approach of behavior based descriptions plays an important role.

References

[Bouchachia, 2001] A. Bouchachia. *Information Retrieval Techniques for Software Retrieval*. PhD thesis, Universität Klagenfurt, Sept. 2001.

[Ganti *et al.*, 1999] V. Ganti, J. Gehrke, and R. Ramakrishnan. Mining Very Large Databases. *IEEE Computer*, 32(8):38–45, August 1999.

[Harrold, 1999] M. J. Harrold. Testing evolving software. *The Journal of Systems and Software*, 47(2–3):173–181, 1999.

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

[Mili *et al.*, 1998] A. Mili, R. Mili, and R. T. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5:349 – 414, 1998.

[Mittermeir and Oppitz, 1987] R.T. Mittermeir and M. Oppitz. Software bases for the flexible composition of application systems. *IEEE Trans. on Software Engineering*, SE-13(4):440–460, April 1987.

[Mittermeir *et al.*, 1998] R. T. Mittermeir, H. Pozewaunig, A. Mili, and R. Mili. Uncertainty aspects in component retrieval. In *Proc. IPMU'98*, pages 564 – 571, Paris, July 1998. EDK.

[Mittermeir, 2001] R. T. Mittermeir. Software evolution - let's sharpen the terminology before sharpening (out-of-scope) tools. In *Proc. IWPSE 2001*, Vienna, Sept. 2001. ACM SIGSOFT.

[Pirker, 2001] H. Pirker. *Specification Based Software Maintenance (a Motivation for Service Channels)*. PhD thesis, Klagenfurt University, Austria, Sept. 2001.

[Podgurski and Pierce, 1993] A. Podgurski and L. Pierce. Retrieving Reusable Software by Sampling Behavior. *ACM Trans. on Software Engineering and Methodology*, 2(3), July 1993.

[Pozewaunig and Mittermeir, 2000] H. Pozewaunig and R. T. Mittermeir. Self Classifying Components - Generating Decision Trees from Test Cases. In *Proc. of the SEKE2000*, pages 352–360, Chicago, Ill, USA, July 2000.

[Pozewaunig and Rauner-Reithmayer, 2002] H. Pozewaunig and D. Rauner-Reithmayer. Behavior analysis based program understanding to support software maintenance. In *Proc. of the 20th IASTED* , Innsbruck, Austria, Feb. 2002.

[Pozewaunig, 2001] H. Pozewaunig. *Mining Component Behavior to Support Reuse*. PhD thesis, University Klagenfurt, Austria, Oct. 2001.

[Quinlan, 1993] J. R. Quinlan. *C4.5 – Programs for Machine Learning*. The Morgan Kaufmann series in machine learning. Morgan Kaufman Publishers, San Mateo, CA, USA, 1993.

[Roussopoulos, 1982] N. Roussopoulos, editor. *Proc. Workshop on Self-Describing Data Structures*. University of Maryland, Oct. 1982.

[Tracz, 1988] W. Tracz. Software reuse: Motivators and inhibitors. In W. Tracz, editor, *Software Reuse: Emerging Technology*, IEEE Tutorial, pages 62 – 67. IEEE-CS Press, 1988.

[Weiser, 1984] M. D. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, July 1984.