# Goal-Driven Combination of Software Comprehension Approaches for Component Based Development

Roland T. Mittermeir, Andreas Bollin, Heinz Pozewaunig, Dominik Rauner-Reithmayer
Institute of Informatics-Systems
Klagenfurt University, Austria
Universitätstrasse 65-67, A-9020 Klagenfurt
{mittermeir, andi, hepo, dominik}@isys.uni-klu.ac.at

## ABSTRACT

This paper reports on our approaches to combine various software comprehension techniques (and technologies) in order to establish confidence whether a given reusable component satisfies the needs of the intended reuse situation.

Some parts of the problem we are addressing result from differences in knowledge representation about a component depending on whether this component is a well documented in-house development, some externally built componentry, or a COTS-component.

## Keywords

Program comprehension, software visualization, cognitive models, specification animation, trace analysis

## 1. MOTIVATION

While the issue of building software from building blocks [12, 15] shifts from using classical reusable building blocks to using off-the-shelf components, modern software technology supports software development on the basis of non-trivial componentry. However, one of the key issues causing the Not-Invented-Here syndrome [29] remains: How can developers be sure that the component they plan to use in their new construction venture meets the expectations placed into it.

As long as reuse is confined to domain-specific in-house reuse, the question on whether one can trust into a component provided by colleagues is relatively benign. It can be resolved informally and final confidence might be established by means of test data. As soon as reuse transcends organizational boundaries – and with COTS integration, this is the normal situation –, this informal trust (whether it was ever justified or not is here not the issue) is lost. Just testing is insufficient to re-establish it. Specific mechanisms have to be devised in order to compensate for the loss of in-depth informal information.


*SSR'01,* May 18-20, 2001, Toronto, Ontario, Canada.


This paper reports on a project where we try to compensate for this loss of informal information by means of software comprehension technology. The approach we follow rests on a broad spectrum of approaches intelligently combined to solve a particular software comprehension task.

The breadth of the spectrum is needed, since we have to foresee that the software to be integrated will come at different levels of representation and it will be supported by different degrees of documentation. Thus, if only binaries are available, the armory for checking whether the respective software actually is what it is supposed to be will be rather limited (although the situation is not hopeless). But if other forms of documentation are available, the set of comprehension aids will correspondingly become larger, thus allowing for more efficient analysis.

In the sequel, we will highlight our approach, discuss the various strategies one might combine and finally describe, how we plan to benefit from the interaction of these strategies. The paper concludes with a sketch of a framework for a conceptual comprehension-architecture a software engineer might use for investigating reusable components.

## 2. BACKGROUND AND OVERVIEW

### 2.1 Reuse and Software Comprehension

The issue we are dealing with is essentially a software comprehension problem. This problem is discussed from many vantage points in the software maintenance literature as well as at special conferences and workshops dedicated to this question, e.g. the IWPC or ICSM. The key problem discussed at these rather maintenance oriented conferences is, that software engineers want to have some aid for rapidly building themselves a conceptual model about the piece of software they are facing.

Usually they need this model to safely perform some maintenance operation. Therefore, it suffices to obtain only some partial (in most cases local) understanding. An understanding that suffices to build a mental model one can trust upon when performing a particular change or a model for locating the spots where such a change needs to be applied. Thus, the conceptual model to be built needs to be rather detailed at critical spots but it might not address at all those portions of the system that are definitely not inflicted by any far reaching side effects of the intended modification.

In the context of software reuse or COTS integration, the fundamental issue is similar: A software engineer does not need to know every detail about a component to be inte-

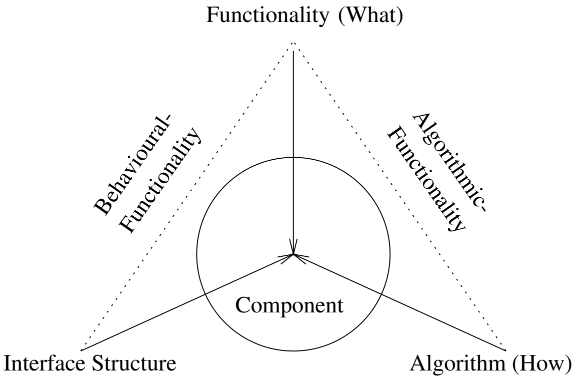


**Figure 1: Different views onto a component**

grated. S/he does need to know though, whether the component at hand renders the required functionality and whether in doing so it would not occasionally spoil parts of the system to be built by performing unwanted functionality. (One might also be interested in performance issues or other extra-functional issues. But these questions are not addressed in this paper). Hence, at a first glance, the reusers needs and the maintainers needs look alike. They are different though from two perspectives:

1. The reuser even more than the maintainer might be interested in perusing the component at hand at various levels of granularity [17], and
2. the reuser, specifically the COTS-integrator, might not have all the information available, a maintainer might have at hand.

Figure 1 attempts to conceptually show this situation. The semantics of the component at hand is always in the center of the consideration. However, while the maintainer is rather interested on the components internal structure (missing bottom-side of the triangle), the reuser is rather interested in the components functionality. In analogy to testing terminology, we refer to it as "black-box functionality" and "white-box functionality". The former rests on externally observable information. Thus, it is based on the components interface structure (signature) and can be observed by conventional testing, or, with state bearing components, from trace data. With white-box functionality, we refer to functional descriptions derivable from the very construction of the component. We will address this issue in more detail in section 5.2.

Likewise, we could (presuming the respective information is available) take the most conventional approach to assess functionality by just analyzing the components specification. Thus, Figure 1 can also be seen as three dimensional entity, with the dimension running from the bottom triangle to the top of the pyramid consisting of the levels "data", "code", and "specification". The reusers problem is then to identify in an efficient manner, whether this pyramid contains a sub-pyramid whose specification (the problem to be solved) is contained in the specification of the component. Hence, whether, without detailed examination of the code, those portions of code that might ever be executed in the prospective environment will yield exactly the behavior (data-level) corresponding to the problem specification.

In sections 3 to 5 we discuss the first issue about different granularities in more detail. In the remainder of this section we discuss the scenarios resulting from differences in the available information.

## 2.2 Information about Components

### 2.2.1 *Just Binaries*

The situation where only binaries (and some moderately telling natural language documentation) are available seems to be the poorest among the options to be considered. However, this is quite common for people working with COTS. But even in this case, we are not left in despair. Binaries can be executed and it seems fair to assume that the rudimentary documentation that comes with them tells at least something about the environment needed and the functionality claimed for this component. The latter has to describe at least how input is to be presented to the component. If even this simple assumption is not warranted, we have to accept that the piece at hand is useless and we better don't waste time on it.

If we are able to (systematically) execute a component though, we have already at least two forms of complementary representation: the *binaries*, which we hardly understand in their materialized form, and *test data* in the form of tuples of input-output value combinations. We might use data tuples as alternate representation from which we try to infer the components semantic in a more compact form. Thus, the input-output tuples will constitute the basis on which the comprehender forms hypotheses about the component.

These hypotheses about software need to be verified, irrespective on which grounds they have been established. In conventional forward engineering, we write code and verify the hypothesis that the code we've written is the code we wanted by means of testing. Here, we can reverse the process.

### 2.2.2 *Source Code's modest Role*

We might use the binary in its executable form and/or use source code directly to verify the hypotheses established previously. Such hypotheses result from the execution of the component or from investigating some specification. Of course, it is fair to assume that the source code is too voluminous (too ill-structured, or otherwise too problematic) to be inspected by hand and by eyesight. Hence, we might use partitioning (e.g. slicing) or (partial) visualization to support our comprehension task. The approaches for source-code visualization are multifarious and diverse. We will discuss techniques to support source-code comprehension in section 3.

Thus, and this might seem astonishing at first glance, the role of source code in its original textual form is quite limited, if the component to be analyzed is beyond a critical size. Of course, if we want to focus on some specific details of an operation, source code is the most telling representation.

### 2.2.3 *Specifications*

In a more luxurious situation we might even have some formal specification coming along with source code. Before being too glad, we have to verify, whether this is not a Danaean gift, i.e. whether the specification is still a valid high level representation of the code at hand. This needs verification too. But in order to check for this as well as in order to comprehend a huge formal specification, we are as much at loss as with voluminous code. We even reach the limits earlier, since the semantic contents of specifications is much tighter packed than the semantic contents of code written in an executable programming language. Hence, visualization is going to play its role again. We might even be lucky by being faced with an executable specification. In this case, we can again reach down into the level of exemplary data. Relating data for specification execution with data from program execution will be an interesting alternative to overcome the limitations of program comprehension. We will discuss aids for specification comprehension in section 4.

#### 2.2.4 Partial Understanding

Finally, we have to see that in general, we are not interested in (and not capable for) comprehending a larger piece of software in its entirety. We are just interested to know, whether it does what we want it to do and whether it does not what it should not do. In between, there might be a quite substantial space, we do not care about, since we know from the specification of the problem currently to be solved, that the current usage environment will never delve into those areas. The parts of the system that take care of these "unused portions of the domain offered" will be executionally dead in the foreseen usage environment. Dead code, specifically pseudo-dead code of this kind will not excite any software engineer. But we are so used to it in huge systems we use all day long that we should accept it - at least for a while - in systems we are really rushed to build. These issues of sufficient and partial understanding is addressed in section 6.

## 3. CONVENTIONAL APPROACHES FOR SOFTWARE COMPREHENSION

We consider the comprehension problem as the problem of building oneself a conceptual model about a piece of software. As argued in [17] there are good reasons to assume that such a reversely established conceptual model might be as much leveled as software models we consider necessary during forward development are represented at various levels. Likewise, the representational form of the models will change depending on the level of abstraction as well as depending on the level of comprehensiveness of such a model. The point of departure in conventional software comprehension approaches is usually source code. Therefore, we will first discuss approaches to better grasp the semantic content of source code. The prominent approaches to be followed are partitioning and visualization. Of course these conceptually orthogonal approaches can be combined to yield more focussed understanding.

### 3.1 Comprehension Support by Source Code Partitioning

If one wants to remain on the textual representation of the source-code level, comprehension support usually amounts to some kind of partitioning. If this partitioning is to be comprehensive but strictly without changing the level of abstraction, some kind of slicing will be the result. A slice, according to Weiser's original definition is an independently executable piece of code taken out of a more comprehensive piece of executable software [32]. Whether something belongs to the slice or not depends on the given slicing criterion, i.e. on a specific point in the program (a variable at a concrete point in the control structure). Everything contributing to the value of this variable on this spot will belong to the respective backward slice. Everything depending from the value of this variable on this spot will belong to the forward slice. This simple but powerful idea can be (and has been) generalized in various directions. We will fall back on this concept later.

Other partitioning strategies on the source-code level, be they declaration analysis, signature analysis, chunking etc, leads to loss of a key property of software: its executability. Therefore, these approaches cannot serve to model "reduced functionality" in the sense of "just the functionality necessary for the environment the component under consideration is to be integrated in". Indeed, most of these approaches, while maintaining the low-level textuality aim at higher levels of abstraction. Hence, they are valuable for focussed partial comprehension. We therefore put these approaches in a separate but distinct bag.

While this distinction holds in general, we might still briefly come back to the technique of chunking [5]. Aiming at black-box reuse, this technique will be of little use for comprehension, other that the boundaries of a chunk define an obvious boundary of focus. Considering white-box reuse though, a chunk is an executable subsection one might scavenge out of a more comprehensive system. Thus, figuratively, a chunk can be seen as a "horizontal" portion of code while a slice is a "vertical" portion cut along the data- and control-flow. Given proper "excitation" by properly defined parameters, we might rest assured that nothing else but a specific slice of the code will be executed. Thus, comprehending a slice amounts to comprehending the component in a domain-restricted way while comprehending a chunk would amount to comprehend some sub-functionality embedded in the component.

### 3.2 Program Visualization

With visualizations, we are changing the level of representation. Thus, we leave the fine granularity of textuality and replace it by some semantically rich notation (usually supported by text, whenever precision in terms of detailed semantics is needed). An overview of program visualization approaches can be found in the work of Stasko et al. [25]. They propose the following classification:

- The range of programs in terms of generality and scalability that program visualization systems are able to take as input for visualization are qualified in the "Scope" category.

- The "Content" category focuses on the subset of (program and/or algorithm related) information that software systems are able to visualize. It also applies to questions of fidelity, completeness and the time of data gathering.

- The "Form" category includes the medium (primary target), the presentation style (graphical vocabulary, animation, sound), granularity, multiple views and synchronization capabilities.

- The "Method" category focuses on the fundamental features of the program visualization system, including the visualization specification style (hand-coded, library-based) and connection techniques (connecting visualization and code).

- The "Interaction" area describes the control and interaction capabilities of the user (style, navigation, scripting facilities).

- "Effectiveness", finally, aims at measuring the quality and quantity of information which is communicated to the user.

Most approaches discussed by Stasko, only focus on a specific problem domain. Thus, they do not support different cognition models and they can only be applied in a very specific field. There are visualizing tools for concurrent programs (*PVaniM* [25, p. 237f]), tools for supporting education (beginning with *BALSA*, or *Piper* [25, p. 383f] or tools in the field of software engineering e.g. maintaining large systems using *SeeSoft* [25, p. 315f]. As a more comprehensive approach, one might consider the RIGI tool [33]. It is specifically geared towards the support of reverse engineering [26].

The RIGI Reverse Engineering tool uses *views* to direct the user's focus by means of visual data and to guide the exploration of spatial data to support program documentation and understanding [28, 19]. The system consists of a parsing subsystem, a repository and a graph editor. The reverse-engineering methodology is based on subsystem comprehension, views, hypertext layers. An extension using scripting languages is possible, too [20, 21]. The reverse engineering process involves the parsing of the program resulting in a graph where nodes represent functions and data-types and arcs represent dependencies among them. Thus, RIGI specifically aims at raising levels of abstraction and at providing focuses on various system dimensions. Specifically the latter property motivates us to mention it here, while not mentioning a host of other visualization approaches developed to support software maintenance and/or software evaluation.

## 4. SPECIFICATION VISUALIZATION

In considering the material at hand for component comprehension, we discussed the case, when specifications are available as an almost ideal situation. However, whether it is a myth and/or a fact, formal specifications [30, p 2303] are often criticized to be hard to understand [10] or not containing all important facets a software developer is interested in [14].

Some of the properties, formal specifications are criticized for (e.g. the lacking connection to other representational forms, either upstream or downstream in the software development process) might not pertain in our case. But still, even for single components, specifications might get large and if so, readers tend to be overwhelmed not least due to the semantical compactness of specifications. Therefore, specification visualization and animation, specification slicing and requirements tracing, have also been explored to improve specification comprehension. Following the arguments raised above, specification visualization techniques can be classified as follows :

1. The first class comprises tools to support *writing* and *reading* of specifications e.g. by providing some kind of pretty-printing. These tools are related to tools for syntax highlighting in source-code or to other aids helping to provide some specific focus on textual representations. One must not deny that also specification browsers are important devices for raising the understandability. We can hardly classify these aids as visualization instruments. But it is well known from source-code comprehension, that the geometrical arrangement of statements substantially aids comprehensibility. The same argument applies for specification styles and the respective arrangement of properties and terms [11].

2. Proper visualization, in the sense of transforming textual information into some *graphical representation*, is usually not the theme when considering formal specifications expressed in Z or VDM. We should not forget though that with certain specification languages (e.g. Petri Nets or finite state machines) the graphical representation is so prominent that we tend to forget the textual linear or tabular representation behind the graphs. While tools would obviously depend on the latter, people reason about the formal model in terms of the graphical representation. A justification of this phenomenon might be the explicit connectedness of related (e.g. neighboring) states. This observation might as well be transferred to those specification languages, we usually see in their textual form. Specifically in connection with context related highlighting, using graphics (or color) to highlight relatedness should prove to be helpful.

3. We might consider *animation* as the next step to render formal specifications "digestible". Here, we need to have executable specifications or at least a way to generate test data from specifications [7, 22]. Of course, animation per se is not a proper visualization. In an animated model, we rather do not see the model in its entirety. We do see a sequence of snapshots instead. Hence, model animation has to be compared with conventional code testing supported by additional mechanisms to interpret or interrelate the results of such executions).

4. A more involved class of specification comprehension tools are to be targeted at *rewriting* a specification and/or *reasoning* about it. Here, the objective is certainly to raise comprehensibility by generating a more compact representation. Thus, usually the level of abstraction is raised.

   We might, however, also consider this as a specification level analogous to program slicers. This will be true specifically, if the tool helps to prove certain properties of input-/output relations, or if it tells, which terms of the specifications have been used to establish a certain proof. In this context, we point to Daniel Jackson's work [11]. He introduces partial specifications as views (consisting of a state space and a set of operations) and composes them to a full specification. Using multiple representations (of different views), it is possible to improve the clarity and modularization of specifications. Another idea is to use advanced slicing techniques [6], either static or dynamic ones, to improve the readability and comprehensibility of specifications.

5. We finally have to think about cross-level tools allowing to *trace* from a specification to its implementation. Such links (e.g. service channels), can be built into the system in a rather simple-minded but inflexible way or can be dynamically established with neatly built systems. To define them for industrial-strength code is very complex though. A cross-section of such approaches can be found in the literature on software evolution [4].

The above discussion shows, that specifications are not just what we see when opening a textbook on formal methods. They are software that can be scrutinized at various levels of precision and they can be presented in various representational forms. Thus, people not “in love” with textbook-style formal methods can still benefit from the preciseness formal methods offer, by inspecting them in a highly focussed manner.

# 5. USING DYNAMIC INFORMATION FOR SOFTWARE COMPREHENSION

With specification animation we have seen already the link between static and dynamic information. In this section, we want to discuss this issue even a bit further. Our focus is on state-bearing software (objects, classes). Here, the crucial question is to identify, whether the hidden state of such an object (class) satisfies the properties a reuser is expecting from the piece of code at hand. Our approach to this question departs from two corners:

1. One is the assessment of object-oriented code, specifically of object-oriented code scavenged out of non-object-oriented legacy software [27, 24].

2. The other stems from generalizing previous attempts for describing procedural, state-less software by means of its executional properties [23, 18] to state bearing software.

This second approach is based on an important requirement: We do not want a human to produce the description for the analyzed components. This would entail potential for latent ambiguity. Our goal is to derive an understandable and interpretable behavioral model automatically by analyzing the effect (the test data) of software directly.

In both cases, we restricted ourselves to descriptions based on the (relatively simple) model of finite state machines, well knowing, that this covers only a modest portion of potential software. However, it covers the portion which can currently help to easily gain some understanding. Attacking software with more involved input/state-interactions is left for times when these seemingly simple issue is adequately understood. In the sequel, we will discuss these approaches, highlighting their merits and their differences.

## 5.1 Analysis of Trace-Information

In order to keep assumptions minimal, we first look at situations, where only trace-data is available as reliable resource. With trace-data we refer to sequences of function calls invoking a complex component. Such sequences are available by analyzing test logs. Textual documentation is of course also needed, but natural language documentation is always burdened with doubtful trustworthiness due to its ambiguity.

The approach we take stems from formal language theory. Thus, we can bank on the fact that any finite sequence of finite length words can always be described by a finite automaton. Various trace sequences, which in our case are represented by different test cases, are such finite length words. We obtain traces by analyzing the component's test logs. However, while the result quoted from formal language theory is reassuring, it is of little help in this context, since the automaton accepting exactly a set of given finite traces provides no reduction in complexity. Hence, what is referred to in literature [2] as grammar inference problem is looking for a more compact canonical automaton that accepts not just the trace-strings but all possible valid trace-strings obtainable from the software component being analyzed.

```
A D
A B B D
     // an empty test case
A B B B D C C
B A D
B A A D C
A B D
```

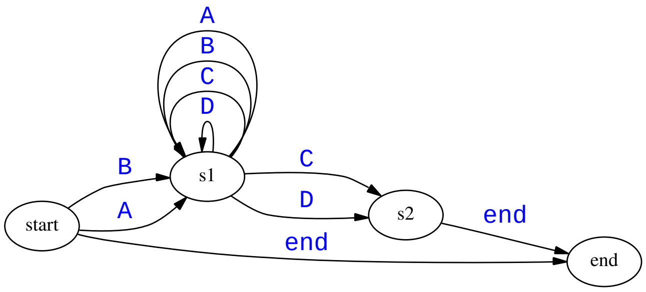


**Figure 2: The grammar inference problem**

In figure 2 an example for such an analysis is presented. Given is a set of test data in the form of method calls (method `A`, `B`, ... ) to a component. Each line represent the sequence of method calls during a “life cycle” of a component. The question is, which regularities may be hidden in the huge amount of unstructured test logs. Such regularities are expressed by a DFA to represent the knowledge about the behavior of components in an understandable form.

We are well aware about the fact that we loose the information about the value transformations performed by the method calls. This information covers a behavioral dimension of the description space which must be handled by other methods (e.g. [23, 18]). With automatic trace analysis, no human aid is necessary to obtain the resulting automaton. Thus the understanding of the component depends solely on the capability of the “reader” of the description (automaton), not on the capability of its producer.

What is left open is the question about the quality of the resulting automaton. Is the description such obtained really the optimal one? Therefore, it is important that the analysis results in a canonical form. With such a *canonical* deterministic finite automaton (DFA), one aims for the following properties, which taken together determine the quality of the resulting automaton [1]:

**Completeness:** The DFA should preserve all transitions which can be found in the trace data. All sequences in the log must be generated by the DFA.

**Irredundancy:** No transition of the DFA should be spurious. This property should prohibit incorrect transitions in the resulting automaton.

**Minimality:** The DFA shall consist of a minimal number of state transitions.

In 1978 Mark Gold [8] proved that in general the identification of a canonical automaton on the basis of given positive and negative examples is NP-hard. However, if only positive examples are available, such identification in the general case is not feasible. Therefore, research concentrated in the mid 80ies on heuristics and domain input for building automata in a specific context.

Dana Angluin [2, 3] focussed on methods, which allow to add collateral information to the knowledge base. In fact, her algorithms depend on an oracle, which answers questions like "Is this string accepted by the automaton to be constructed?". Knowing about traces which can not be generated by an component restricts the search space and helps to resolve ambiguities which occur during the inference process. In most cases, such an oracle can be easily obtained in our context, since components are executable and the query can be reformulated in the form of method calls to the component. Thus, the question, if a certain word is accepted by an automaton (which is indeed the sequence of method calls to a component) can be answered immediately. (On the other hand, this may be infeasible, if the component is not executable and can not be transformed to such an form within reasonable time.)

Building on these results, forming of merge-hypotheses and of loop-hypotheses which can be individually tested, can be seen as a promising approach towards automatically generating descriptions of components. The step of reducing an automaton, accepting test-sequences only, to an automaton (presumably) accepting the full "language" a component is ready to deal with is only a small one. The word "presumably" is used in this context, since as in our previous work on this topic, the quality of the result dependents heavily on the quality of the test-data available.

In contrast to this previous work, however, we need not rely on trustworthy test-data only in those special cases, where access to original development information is available either by means of having a formal specification or by means of the test data directly used by the developers as well as during quality assurance of maintenance operations. This is, because we can form hypothesis and test them by directly executing the component we analyze.

## 5.2 Dynamic Models derived from static information

The second approach to be mentioned here is not plagued with the problem whether the trace contains all crucial cases. However, it requires the availability of source code. This approach [24] infers dynamic models (state charts) from (object-oriented) source code. In essence, those variables, resp. object attributes maintaining state information are identified. From that attributes we infer the potential states by analyzing conditions in the control flow graph. Such identified states are the basis for revealing state transitions by looking at the variable's value changes. By considering all identifiable objects of one class we derive many potential modification curriculi. The combination of all potential modification curriculi represents the final dynamic model of the component. Deviations of the such re-engineered dynamic model from a dynamic model an UML-designer would develop, can be considered as hints for problems in the code and/or problems in the understanding of the component. This applies for the analysis of reuse-candidates as much as for the analysis of classes re-engineered from procedurally structured legacy code. To check whether the result of such transformations leads to semantically sound classes was the original aim under which this approach was developed.

# 6. GOAL DRIVEN PARTIAL COMPREHENSION

If we see the comprehension problem as the problem of building oneself a conceptual model about a piece of software at hand, one can consider our approach as a generate-and-test approach.

The software integrator first establishes her-/himself a conceptual and/or mental model about the piece of code at hand and then tries to verify whether this model holds. This verification will usually be done by analyzing a variation of the representational form for this software. Here, we can only give some preliminary sketches of such variations of representational form or goal directed probing into models about software.

The basic premise of this approach is that humans do not comprehend (and mentally manipulate) conceptual models when facing them as long strings of text. Without resorting to psychological literature, we take the host of semi-graphical notations as pieces of evidence for this claim. If so, and if further resorting to the over-used result, that humans can hold up to about seven elementary items concurrently in their short-term memory, we have to accept that "full comprehension" of some sufficiently sizable piece of code is impossible anyway. The reuser does need an adequate proxy for this full comprehension though. Hence, a suitable way (perhaps the only way) is to correlate partial evidence to form a hypothesis about what the piece at hand actually is all about and then use further clues to either stepwise support this hypothesis until a level of satisfactory trust is reached or to disprove it.

In the case of disproof, one has either to reject the hypothesis (and in most cases this decision implies also rejection of the candidate component too) or to continue with analyzing a freshly updated hypothesis if a (thus weakened) hypothesis is still of interest relative to the problem specification at hand. If an "experiment" supports the initial hypothesis, we are in a pragmatically better, but intellectually less satisfying situation. This is because a reusable component at hand is in general to immense to fully grasp it by either of the methods discussed above [17]. Thus, again resorting to general principles of scientific discovery, we aim not only for a variation of focal length but also for a variation of representational form.

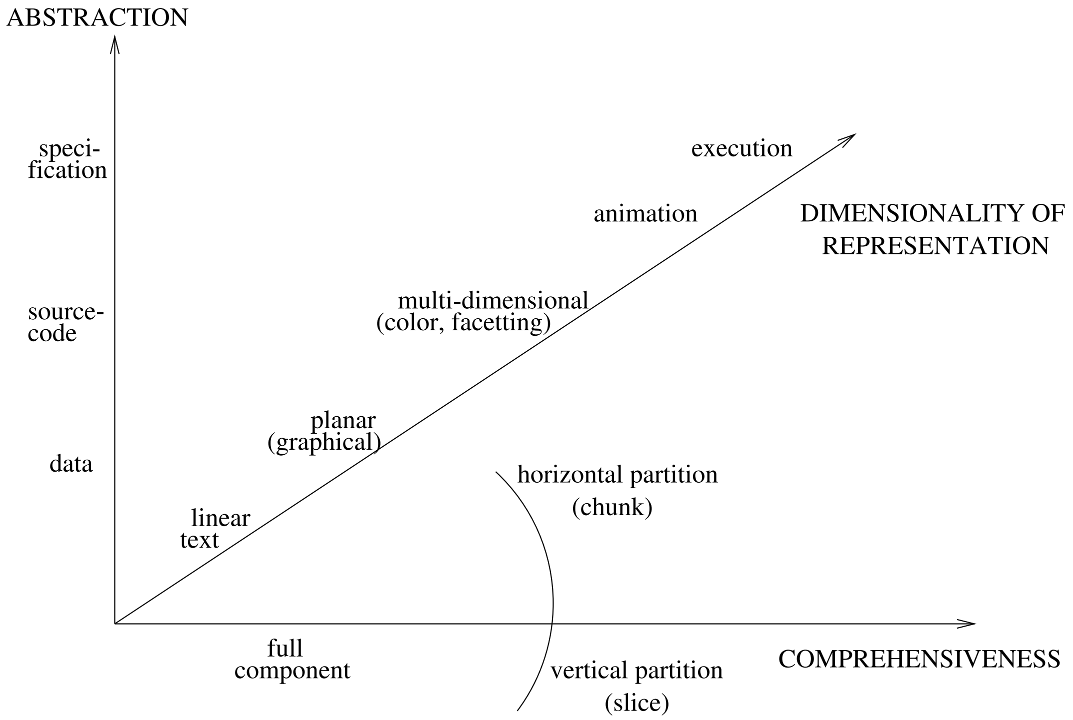


**Figure 3: Conceptual dimensions for analyzing Software**

Debating focal length, we think about the levels of specification, source-code, and data-points as three distinct representations. Specification and source-code share the property of completeness (and overwhelmingness), data-points are comprehensible but episodic. For each of these levels, we suggest a textual representation as representational norm. This choice seems adequate, since textuality is needed if one aims at adequate machine support in the analysis process[1].

In addition to the textual representation we consider partitioning (on the text level) and visualization as key options to support comprehension. Partitioning for supporting comprehensibility by reduction in volume, visualization (transformation into whatever form of two- or even higher dimensional representations) as a means to spread out information in various dimensions [16] (e.g. color can be used to model more than two dimensions on the plane of a sheet of paper or a video-screen [9]). Thus by combining various tools and techniques, we obtain a torus of various representational forms around of a common semantic core of functionality. Using animation-tools, this torus might even become "active". One must not underestimate the suggestive power of "making things behave". However, one must also not overestimate animation, because it is always burdened by its episodical character.

Figure 3 shows a grid into which such a torus might be placed. Obvious questions at this point are: "What is an optimal combination of grid-points to establish such a torus?" or "What is the optimal path through this torus?" – At this point, we have to deny an answer. We attempt at the more modest answer of sketching various more detailed road maps through it in our future research though. We can mention at this time though, that this representational torus is a conceptual framework. The concrete positioning of the various representation nodes in this framework leads to a "comprehension architecture" that needs to be domain specific.

[1]Binaries might also be considered as textual representations. But even while they are the final result a reuser is aiming for, they play no role in our considerations other than that they are needed to make source code executable (as much as executable specifications need some representation transformation for being executable).

## 7. CONCLUSION

Integrating ready-made components into a new software system under construction has always to do with trust. Trust, that the component integrated really behaves as specified resp. as desired. This paper presented a "tour de raison" through various approaches to establish this trust.

The key-claim of the paper is, that the proper combination of partial representations of software, be it representations by means of test-/trace-data, by means of source-code or by means for formal specifications as well as variations in the level of abstraction and presentation will help software engineers to more effectively and more efficiently establish the trust needed to integrate a component which is not-invented-here.

## 8. REFERENCES

[1] R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, editors, *6th International Conference on Extending Database Technology - EDBT'98, Valencia, Spain*, volume 1377 of *Lecture Notes in Computer Science*, pages 469 – 483. Springer, March 1998.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.

[3] D. Angluin. Identifying Languages from Stochastic Examples. Technical report, Yale University, 1988.

[4] B. Balzer, T. Katayama, and D. Perry. *Proc. International Workshop on Principles of Software Evolution (IW-PSE98)*. 1998.

[5] I. Burnstine, A. Mirza, K. Roberson, F. Saner, and A. Roberson. Knowledge engineering for automated program recognition and fault localization. In *Proceedings 8th International Conference on Software Engineering and Knowledge Engineering SEKE '96*, pages 85–91, june 1996.

[6] J. Chang and D. J. Richardson. Static and Dynamic Specification Slicing. Technical report, Department of Information and Computer Science, University of California, 1994.

[7] U. E. Fojan. Spezifikationsbasiertes Testen objektorientierter Software. Master's thesis, Universität Klagenfurt, Österreich, 1998.

[8] E. M. Gold. Complexity of Automatic Identification from given Data. *Information and Control*, 10:302–320, 1978.

[9] C. R. H. Gall, M. Jazayeri. Visualizing software release histories: The use of color and third dimension. In *International Conference on Software maintenance (ICSM '99)*, pages 99–108, 1999.

[10] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[11] D. Jackson. Structuring Z Specifications with Views. *ACM Transaction on Software Engineering and Methodology*, 4(4):365–389, October 1995.

[12] G. Kaiser and D. Garlan. Melding software systems from reusable building blocks. *IEEE Software*, 4(4):17–24, july 1987.

[13] L. Latour, editor. *8th Workshop on Institutionalizing Software Reuse (WISR8)*, Columbus, OH, USA, March 23-26 1997.

[14] G. T. Leavens and C. Ruby. Specification Facets for More Precise, Focused Documentation. In Latour [13].

[15] M. Lenz, H. Schmidt, and P. Wolf. Software reuse through building blocks. *IEEE Software*, 4(4):34–42, july 1987.

[16] R. Mittermeir. Die dimensionen von information. In B. A., editor, *Kultur - Information - Informationskultur*, pages 91–101, 1996.

[17] R. T. Mittermeir. Comprehending by varying focal distance. Proc. 8th International Workshop on Program Comprehension, IWPC 2000, Limerick, Ireland, June 2000.

[18] R. T. Mittermeir and H. Pozewaunig. Classifying Components by Behavioral Abstraction. In Wang [31], pages 547–550.

[19] H. Müller, S. Tilley, M. Orgun, B. Corrie, and N. Madhavji. A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models. In *SIGSOFT'92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, volume 17, pages 88–98. ACM Software Engineering Notes, December 1992.

[20] H. A. Müller, S. R. Tilley, and K. Wong. Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project. In *CASCON'93*, pages 217–226, October 1993. Toronto, Ontario.

[21] H. A. Müller, K. Wong, and S. R. Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Colloquium on Object Orientation in Databases and Software Engineering; The 62nd Congress of L'Association Canadienne Francaise pour L'Avancement des Sciences*. ACFAS, 1994.

[22] A. J. Offutt, Z. Jin, and J. Pan. The dynamic Domain Reduction Procedure for Test Data Generation. *Software – Practice and Experience*, 29(2):167–193, February 1999.

[23] H. Pozewaunig and R. T. Mittermeir. Self Classifying Components - Generating Decision Trees from Test Cases. In S.-K. Chang, editor, *Proceedings of the 12th International Conference on software engineering & Knowledge Engineering – SEKE2000*, pages 352–360, Chicago, Ill, USA, July 2000.

[24] D. Rauner-Reithmayer and R. T. Mittermeir. Behavior Abstraction to support Reverse Engineering. In *International Conference on Software Engineering and Knowledge Engineering SEKE'98*, San Francisco, USA, June 1998.

[25] J. Stasko, J. Domingue, M. H. Brown, B. A. Price, et al. *Software Visualization Programming as a Multimedia Experience*. MIT Press, 1998.

[26] M.-A. Storey, K. Wong, and H.A.Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? Technical report, School of Computing Science, Simon Fraser University, 1998.

[27] M. Taschwer, D. Rauner-Reithmayer, and R. T. Mittermeir. Generating Objects from C code – Features of the CORET Tool-Set. In *3rd European Conference on Software Maintenance and Reeingineering – CSMR'99*, Amsterdam, Netherlands, March 1999. IEEE, IEEE CS Press.

[28] S. R. Tilley, H. A. Müller, and M. A. Orgun. Documenting software systems with views. In *SIGDOC'92: Proceedings of the 10th International Conference on Systems Documentation*, pages 211–219. ACM, October 1992.

[29] W. Tracz. Software reuse: Motivators and inhibitors. In *Proceedings COMPCON S'87*, pages 358–363, 1987.

[30] A. B. Tucker. *The Computer Science and Engineering Handbook*. CRC Press, 1996.

[31] P. P. Wang, editor. *JCIS'98 – 4th Joint Conference on Information Sciences*, RTP, North Carolina, USA, October, 23–28 1998. Association for Intelligent Machinery.

[32] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[33] K. Wong. *Rigi User's Manual*. Department of Computer Science University of Victoria, June 1998.