Dissertation

Mining Component Behavior to Support Software Retrieval

zur Erlangung des akademischen Grades eines
Doktor der technischen Wissenschaften

Studium der Angewandten Informatik

eingereicht am Institut für Informatik-Systeme
der Universität Klagenfurt
Fakultät für Wirtschaftswissenschaften und Informatik

von
Dipl-Ing. Heinz Pozewaunig

Neckheimgasse 26, 9020 Klagenfurt
geboren am 17. Oktober 1967 in Klagenfurt

Begutachter:
o. Univ.-Prof. Dipl.-Ing. Mag. Dr. Roland T. Mittermeir
o. Univ.-Prof. Dipl.-Ing. Dr. Johann Eder

Klagenfurt, im Oktober 2001

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, daß ich die vorliegende Arbeit selbst verfaßt und alle ihr vorausgehenden oder sie begleitenden Arbeiten durchgeführt habe. Die in der Arbeit verwendete Literatur sowie das Ausmaß der mir im gesamten Arbeitsvorgang gewährten Unterstützung sind ausnahmslos angegeben. Die Arbeit ist noch keiner anderen Prüfungsbehörde vorgelegt worden.

Klagenfurt, am 19. Oktober 2001

Kurzfassung

Software-Wiederverwendung beschäftigt sich mit der Umsetzung der einfachen Idee, dass es kostensparender und qualitätssteigender ist, bereits existierende, hochwertige Software-Komponenten bei der Entwicklung oder Wartung eines Systems zu benutzen, statt diese neu zu entwickeln. Eine wichtige Voraussetzung für erfolgreiche Software-Wiederverwendung ist die richtige Auswahl von passenden Komponenten. Dabei müssen aus der Vielzahl von existierenden Komponenten diejenigen bestimmt werden, die die benötigte Funktionalität zur Verfügung stellen. Dies muss einfach und schnell möglich sein, denn wenn die Suche nach Komponenten mehr kostet, als deren Neuentwicklung, wird Software-Wiederverwendung ad absurdum geführt. Die meisten Ansätze für Software-Retrieval, dem Wiederfinden von Software in Komponenten-Bibliotheken, basieren auf einer textuellen Beschreibung von Komponenteneigenschaften. Texte in natürlicher Sprache sind dem ersten Anschein nach einfach zu verstehen, ihre präzise Interpretation hängt jedoch sehr stark vom Kontext des Beschreibers, bzw. des Lesers ab. Weiters kann die Beschreibung, die eine Abstraktion der Komponente ist, nicht alle Aspekte beinhalten, die einen zukünftigen Leser interessieren könnten. Dadurch wird das Auffinden erschwert. Nebenbei ist die Erzeugung von treffenden Beschreibungen aufwändig und zeitintensiv.

Diese Dissertation beschäftigt sich (1) mit der automatischen Erzeugung von Software-Beschreibungen, die nicht auf natürlich-sprachlichen Texten aufbauen und (2) mit Software-Retrievals auf der Basis der in (1) entwickelten Beschreibungen. Die generierten Beschreibungen sind wiederum keine natürlich-sprachlichen Texte, sondern einfache formale Konstrukte: Entscheidungsbäume, endliche Automaten und Grammatiken. Da diese eine exakt definierte Semantik haben, entfällt die Interpretation durch einen menschlichen Leser. Von zentraler Bedeutung für den in der Arbeit vorgestellten Ansatz sind Testdaten, weil diese das tatsächliche Verhalten einer Komponente, wenn auch nur ausschnittsweise, unverfälscht widerspiegeln.

Die Verwendung von Testdaten als Ausgangspunkt für die Beschreibung des Verhaltens von Komponenten macht deren Einteilung in die Gruppen der zustandslosen und zustandstragenden Komponenten notwendig, was auch zwei unterschiedliche Gruppen von Analyseverfahren bedingt. Für das erste Verfahren werden Signaturen von zustandslosen Komponenten generalisiert und Komponenten mit gleichartigen Signaturen zu Partitionen einer Software-Bibliothek zusammengefasst. Mittels Testdaten wird danach für jede Partition ein Entscheidungsbaum generiert, der als Browsing-Struktur zum Auffinden von Komponenten dient. Im zweiten Verfahren beschreiben endliche Automaten bzw. Grammatiken, die aus Testdaten abgeleitet wurden, zustandstragende Komponenten. Wird eine Komponente in einer Bibliothek aufgefunden, kann man durch endliche Automaten und Grammatiken feststellen, ob das geforderte Verhalten tatsächlich gezeigt wird.

Beide in dieser Arbeit vorgestellten Verfahrensgruppen unterstützen die Verwaltung einer Software-Bibliothek und erleichtern das Auffinden von Komponenten. Da sie von Testdaten ausgehen, sind sie ideale Erweiterungen von Software-Retrieval-Verfahren und das nicht nur dann, wenn textbasierende Verfahren bereits an ihre Grenzen stoßen.

Abstract

Software reuse is based on the simple idea, that it is more economic and even quality improving to make use of proven components instead of reinventing them when building or maintaining a system. An important prerequisite for successful software reuse is the choice of the most suitable components. Those which provide the required functionality have to be filtered out of a multitude of existing components. This process must be simple and quick, because if the search for components were more expensive than the development of new ones, software reuse would not make any sense at all. Most approaches for software retrieval are based on a textual description of the components´ properties. At first sight, texts written in natural language are easy to understand, but when it comes to precise interpretations one becomes aware of the gaps between the writer´s and the reader´s contexts. Furthermore, a description, which is an abstraction of a component, cannot express all the information a future reader might be interested in, which renders the search difficult. In addition, the production of detailed descriptions is costly and time consuming.

This thesis deals (1) with the automatic generation von software descriptions which are not based on texts written in natural language and (2) with software retrieval which depends on the descriptions developed in (1). In contrast to natural language texts, these descriptions are straight, formal constructs which are decision trees, finite state automata and grammars. Due to their well defined formal semantics an interpretation by a human reader is not needed any longer. In our approach test data deals with the problem in a very efficient way. Although test data do generally not reveal a complete spectrum of a component´s behavior, they specify a significant and genuine part of it.

As test data are employed for generating descriptions of the behavior of components, the two groups of stateless and state-bearing components need to be introduced. This entails the necessity to develop two different analysis techniques. Within the first technique signatures of stateless components are generalized and components with similar signatures are gathered in the same partition of a software repository. On the basis of test data decision trees are then generated for each partition, which serve as browsing structures to locate components. The second technique deals with state-bearing components. Here, finite state automata and grammars, which are inferred from test data, are being used as descriptions. When a component is located in a repository with the help of finite state automata and grammars it can be verified whether a component behaves as specified.

Both techniques developed in this thesis support the administration of a software repository and help to locate components. Starting from test data, they represent substantial improvements to software retrieval approaches – even when text based techniques have already arrived at their limits.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Software reuse is based on the simple idea that it would be better to make use of existing large building blocks in the process of building a new system instead of reinventing all the small and tiny solutions which have been reinvented a thousand times before. What is undoubtedly best practice in all other fields of engineering now, astonishingly enough, has not become state of the art in software engineering yet. Nevertheless, the level of software reuse has been increased tremendously: Programmers use compilers, whose knowledge about formal language design is reused; they use mathematical libraries and widget repositories, as well as data bases where all the functionality needed to ensure persistence, security, multi-user accessibility to data is being reused. On a high level, product lines are established to reuse all the assets which solve similar problems for a domain. Already these examples demonstrate, that it is not that easy to distinguish exactly between reuse and use. What was explicitly reuse in former days became part of the every-day life, a process which indicates a successful assimilation of reuse.

But despite this progress, the situation is not as satisfying as it could be. So the question arises, why it is so hard to access all the valuable assets which are built with so much effort, which are tested for high quality and which are annealed by their employment in a running system?

There are many reasons for that which will be discussed in detail in chapter 3. However, the main reason for this slow evolution of software reuse lies in the inherent nature of software, which cannot be captured in terms of a *materialized* entity. Although software originates in the physical world, as its bits and bytes, located on a hard disk, can be identified like characteristic fingerprints under a microscope, the physical dimension is the most unimportant one for software engineers. What cannot be materialized, however, is the concept and the knowledge embedded in software, as well as the effects it causes in the real world. Only when executed, some parts of this concepts demonstrate effects; be it as a change in the balance of a bank account after depositing an amount, or as a determination of a point's location in a three-dimensional space. But such behavioral snapshots provide only with a very small insight into the spectrum of a software's behavior. The way in which this software actually reacts under different circumstances cannot be inferred completely from examples as it is impossible to grasp the entirety of the behavior of software on the basis of some few instances.

Due to the immaterial nature of software it is not easy to describe its essentials, which has many consequences for reuse. A software engineer who produced a reusable component usually is not a specialist in describing it; and a programmer, who wants to build a new system, is not a specialist in interpreting a description. To illustrate this by a metaphor: Imagine a person writing a newsletter about the danger of drinking alcohol who needs a clip art to stress this point. The writer envisages a picture of a threatening beast lurking for its prey which should represent a hidden menace. There is a data base with thousands of clip arts available. Now, how should the writer specify his/her needs, since the artist most likely described the pictures in general terms with little regard to associative meanings? May be a query containing keywords like "animal, dangerous, hidden" leads to a result, maybe the writer has to scan through a large set of candidate pictures retrieved by the query. It could also be that the writer misses pictures which would fit her/his needs much better, e.g. a sword of damocles.

This thesis deals with the challenge of producing descriptions for reusable components and of supporting a reuser to locate components in a repository on the basis of these descriptions. If these descriptions

- were obtained automatically,
- could be interpreted non-ambiguously,
- would guide a searcher in the task of locating a component in a library, and
- would support organizing a component library for effective retrieval,

a heavy burden would be taken off the software engineers producing components, the programmers reusing them, and the librarians administering them. The results presented in this work are a step into this direction.

The main idea of this thesis is represented by the following sequence of activities: (1) Observing components, (2) reorganizing them with respect to the observed behavior and (3) abstracting from the concrete behavior to a more compact and thus more understandable form. But how can invisible activities possibly be observed? The problem can be solved by applying methods similar to those used by physicists, who have to monitor the rapid movements of particles smaller than atoms. These movements in space are not observable directly. Hence, cloud chambers are being used in which not the particles themselves, but their movements become visible through their "contrails". In this way, physicists are able to reason about properties of the particles by analyzing their traces in the fog. In the realm of software the equivalent of such visualized traces is test data. Test data indirectly reveals components by including information about sequences of input and the resulting outputs of components. In that way, test data can be regards as an observable aspect of the execution of software components.

Test data partially represents the behavioral spectrum of a component. In this work the knowledge hidden in test data is exploited for producing descriptions by using techniques from the field of data mining. The class of this descriptions is called *extensional descriptions*. Due to the different properties of their test data we have to distinguish between two types of reusable components,

- state-less components, and
- state-bearing and/or complex components.

State-less components demonstrate behavior which is deterministic for a single test case, in so far as a component always computes one and the same output for a particular input. Therefore, a meaningful set of test cases (which must have a certain level of quality) is has been analyzed; its input-output transformations are applied as the basis to produce characteristic descriptions. In this way a browsing structure for components is being established. The method developed for analyzing state-less components is called SBS, short for *static behavior sampling*.

The repository in which SBS is used for establishing access structures is organized according to the interfaces of components. Interfaces, which we generalize as signatures, independent of implementation, hold important information as well. This information is used for (1) dividing the repository of reusable components into meaningful partitions and (2) for establishing relations of different types between components. For this purpose, a method called generalized signature matching (GSM) is introduced, which extends signature matching techniques known from the literature. GSM is applicable for both state-less and state-bearing components. In this work it is used for organizing a SBS based repository.

State-bearing and/or complex components do not compute the same output on the same input, which is due to internal states maintained. In order to establish an approach similar to SBS, the input-output transformations are abstracted by analyzing the order, in which different kinds of transformations are performed. These differences are condensated by developing indexing methods. Hence, sequences of method-names provide with the information needed to generate descriptions. Two different methods for producing succinct descriptions, which are called *abstracted behavior sampling* (ABS), are introduced. Due to the particularity of ABS the abstraction may not contain enough information, which would not pose a problem since the searcher may step back to a more concrete representation as provided with SBS.

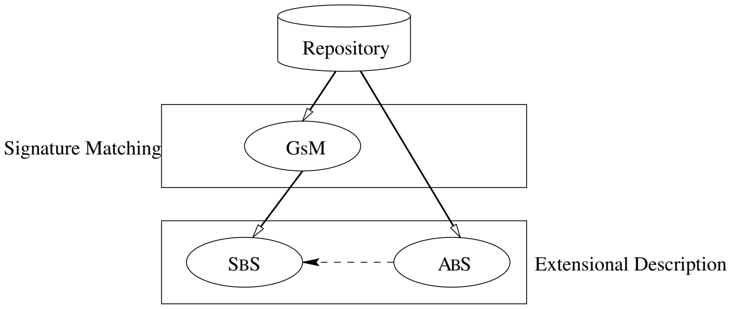


Figure 1.1: The retrieval process

In figure 1.1 the cooperation of the techniques in the task of locating components in a repository is depicted on a very high level. The process is organized into two layers: a signature matching layer and an extensional description layer. Each activity, shown as ellipse in figure 1.1, helps to restrict the number of the retrieved candidate components until the searcher gets the most promising reusable components.

A similar approach is pursuit in the thesis of Bouchachia [32]. Here, software manuals (man-pages) are analyzed and representations are generated which are drastically shortened by applying various neural-net techniques. The affinity to our work is due to the aim to (1) generate component representations automatically and to (2) establish a stratified approach. The results of both works complement one another and can be seen as tools starting from different ground, but aiming at the same target.

1.2 Organization of this work

Chapter 2 provides with a detailed reflection about the dimensions of software reuse as well as some definitions. We discuss a specific software development process model which advances the possibilities of software reuse. Furthermore, the notion of "reusable component" is examined in this chapter.

The consequences of ambiguous component descriptions are revealed in chapter 3. We identify important facets of this problem and discuss challenges which will be addressed in detail in this thesis.

Chapter 4 presents the relevant approaches for software retrieval published in the literature which largely agrees that software can be considered either a specific form of document, a mathematical model, or a formal structure. Accordingly, the methods used for solving the software description and retrieval problem (which are two sides of the coin) differ substantially. Two methods are identified as being promising, extensional descriptions and signature matching.

The idea of signature matching is elaborated in chapter 5. After discussing the basic concepts of types and signatures, a specific representation, the signature graph, is presented. Based on this graph, various signature relations are worked out; some of which are not known in the literature so far. We will discuss a graph transformation which is called *flattening*. A graph transformed in this way allows a more pragmatic and concept-based search for signatures.

The static *behavior sampling method* SBS, presented in chapter 6, is founded on extensional descriptions and uses the techniques developed in chapter 5 for pre-ordering state-less components of a repository. We will discuss the structure of a SBS-based repository and the preconditions for selecting test data to be used as components' descriptions. These descriptions are analyzed by a decision tree algorithm and thus a browsing structure is established.

Chapter 7 illustrates how the ideas developed in the previous chapter can be applied to state-bearing components as well. There we adapted two different techniques according to our needs to produce descriptions for reusable components. We call these methods *abstracted behavior sampling* methods, ABS.

In chapter 8 the work concludes with a discussion of the benefits and drawbacks of our approach. Furthermore, future work is identified and some final remarks are given.

2

Reuse Context

2.1 Reuse economics

In the mid 60ies, software began to develop from an art (or craftsmanship) to a profession. It was pushed ahead by the quest for better control of software costs, quality and time to develope a system. The birthplace of the term *software engineering*, which subsumes this thinking, is seen at the 1968 NATO Conference [160, 112] in Garmisch, Germany. There, the main focus was on finding ways to fight the *software crisis*. Along with this crisis, which evolved due to the rise in capability and the availability of considerably cheaper computers [170], the following issues became evident:

- The range of computable problems was broadened which led to more complex and larger programs.
- More and more enterprizes were able to afford computers and demanded new programs to solve their problems.

These effects led to projects which could not be finished in time (if ever), as well as to software programs with functional deficiencies which were hard to maintain. Up to that time, the best practices to develop software rendered more and more infeasible. As a solution to the software crisis McIlroy [138] proposed the idea of highly standardized software building blocks as a key to industrial like software mass production. Given a large repository of such components, together with automated techniques for customizing these components to the developer's needs, the developer rather asked the question "Which component shall I *use*?"

than “Which component shall I *build*?”[1]. This idea of reusing software assets was persuasive enough for the research community to adopted it at once. Surprisingly, the software crisis is still an ongoing challenge [85] and reuse is still seen as the technique with the highest *potential* to improve software development considerably. At his keynote talk at the STARS conference held at Tysons Corner, Virginia, USA in 1991, Barry Boehm stated three major approaches to increase the effectiveness of software development [93]:

1 You may work faster by using better tools,
2 you may work smarter by developing better processes for software development, or
3 you may avoid work by increasing reuse.

According to Boehm the largest contribution to cost saving among these three topics is expected through reuse. In figure 2.1 one can see, that in the near future the potential improvement in cost savings obtained by better tools or improved processes are not significant. Only in reusing existing artifacts it is expected that costs can be saved dramatically. This view is shared by many other authors as well [91, 72].

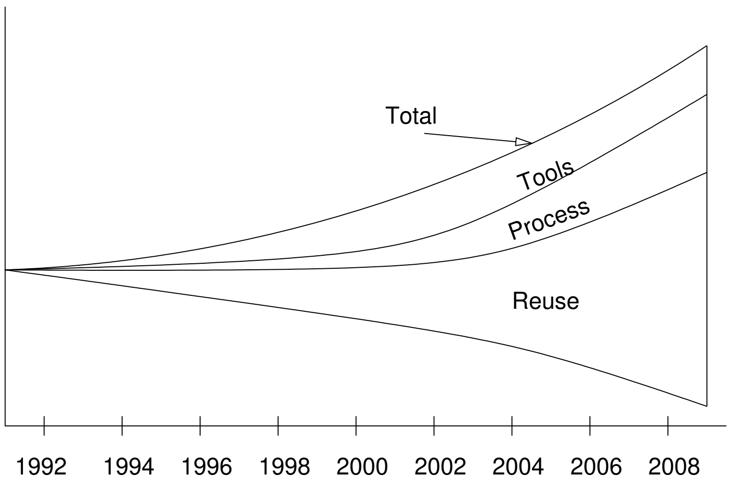


Figure 2.1: Expected cost savings by applying different software engineering techniques ([93, p 17])

[1]Some authors see the dawn of reuse already in 1958 when the FORTRAN II compiler allowed to compile subroutines separately and to generate domain specific packages, e.g. for scientific computing [12, 60].

What is reuse after all? The idea is a very simple one and in the literature of software engineering one can find a variety of definitions. In his reuse survey, Krueger states, that

> " ... *software reuse is using existing software artifacts during the construction of a new software system*" [122, p 133]

whereas Reifer defines software reuse as

> " ... *the process of implementing or updating software systems using existing software assets.*" [195]

The definition of Reifer, which is very similar to the one published by the US Department of Defense Software Reuse Initiative [62], stresses the point that reuse does not happen solely when new systems are built, but also when existing systems are updated and maintained. In this work, this is our view onto software reuse, too.

Both definitions do not limit the software artifacts to source code only. Indeed, the higher the level of abstraction of artifacts is (designs, test cases, specifications, requirement documents, ...), the larger are the expected benefits from reusing it [162, 88]. Considering the whole effort to develop a system, Boehm [31] estimated that the effort to design comes to 29 %, and the effort to conduct tests amounts to 23 %. As a consequence, reusing artifacts from the early phases of the software life cycle leads to significant economic improvements, since all dependent work products of reused artifacts do already exist.

In the remainder of this work, we refer to the notion of software artifacts as *assets*, when an entity of the whole spectrum of reusable software products from any of the phases of the software life cycle is dealt with. Only when referring to the level of implementation (source code, executable binaries) we use the term *components*.

2.2 Reuse Expectations

What are the expectations associated with reuse? There are three main reasons, why the effort to build and maintain a reuse process should pay off [153, 17, 131] and help to deliver software which is less costly:

Quality.

Since the assets for reuse are built for many different applications, their quality must be outstanding and therefore they must be most carefully examined. Additionally, reusable asset may incorporate domain knowledge which is not known by the user of the asset in its entirety. The user does not need to be informed about details of the domain. In this way, major pitfalls can be avoided which could come to existence by re-inventing solutions. As an example, the reader may think of a reusable financial analysis component, where the knowledge about summarizing details is built in. The programmer reusing this analysis tool only needs to plug it into a new system without having to know about the algorithms built into it.

Reuse therefore leads to a significant reduced defect density for the whole product. The increase in quality is due to an increase in reliability, consistency, manageability, and standardization.

Productivity.

Due to the fact that an asset does not need not be built from scratch but has to be incorporated into the product only, the productivity is higher by reusing than by reinventing. Productivity is often considered a function of software size, such as the number of lines of code delivered or the number of function-points delivered. The increase in productivity is due to the higher capacity of programmers, since less effort goes into documentation and testing, and the maintainability is being simplified.

Time to market.

By saving time through incorporating existing assets into a software system, time to market can be reduced drastically, too.

2.3 Reuse process

Reuse may occur in two different styles, systematic and/or opportunistic reuse:

Systematic reuse: This is also called institutionalized reuse and it is the style of software development, in which the inclusion of reusable products during all stages of software development (requirements, analysis, design, implementation, test) is an integral part of the development process.

Opportunistic reuse: This style happens in an ad hoc-manner. During the software development, individuals recognize patterns within their work which occurred already in former projects. Such patterns can range from products like analysis documents, algorithms, to subroutines or whole subsystems. According to their individual knowledge, these persons then incorporate assets into the current product.

Both styles of reuse adoption are important. Opportunistic reuse gives the developers a certain degree of freedom and has the advantage of being easy to implement into a repository of reusable assets. Studies have shown, that the existence of a repository is a key factor for promoting reuse [130]. However, these findings are not shared by other studies. Frakes and Fox have found that the mere existence of a software repository does not influence the level of reuse in an enterprise [75]. Providing technical infrastructure alone is insufficient for a long term success, because then the developers have to carry the main responsibility for initiating reuse. If a developer does not know about previous work, only in slowly gathering experience will his level of reuse adoption rise. This is not effective. On the other hand, the developers should not be responsible for providing reusable assets and for pushing on a development project at the same time. From cognitive psychology we know that the pressure to provide a general solution for a specific problem is too high for developers [125, 126] and that they cannot solve the problem by themselves. In addition, the tension between scarce development resources and long term benefits often leads to a neglect of the actual long term goal [228].

Effective systematic reuse needs a well defined way to generate assets and to use them. Therefore, defined processes for

1. development *for* reuse and
2. development *with* reuse [100]

must be established. Such processes, which are different from that of a "conventional" software development process, should be adapted to the special needs of a reuse centric paradigm. The following assumptions are specific for reuse centered development [18]:

- Software development, especially when having reuse in mind, needs to be viewed as an "experimental" discipline. In order to enable organizations to learn from each development project and incrementally improve their ability to engineer quality products, an evolutionary model is needed.

- To anticipate the specificities of different development projects, a single software development approach cannot be assumed for all of them.
- Software development approaches need to be customizable to reuse centered project requirements and characteristics.

These reasons prevent the application of a standard process and in the literature, different reuse process models are proposed. The model presented here is oriented towards the suggestions of Reifer [195]. It is similar to the models proposed by Hochmüller and Mittermeir [102] or Mambella [135] in the way that it reflects the intention to make reuse an integral part of software development without radically changing the organization very well. This evolutionary idea raises the chance for the success of a reuse initiative [101].

The model of Reifer consists of three processes. *Domain Engineering* aims at generating reusable assets, *application engineering* serves for developing software on the basis of reusable assets, whereas *asset management* supports both processes by administering the reuse library.

Development for Reuse – Domain Engineering.

The process of development for reuse has the aim to organize the way of building high quality assets which are easy to incorporate into a new software system. In the literature this process is called *domain engineering* as well [91, 205]. A domain is a distinct application area that can be supported by a class of systems with similar requirements and capabilities [114]. Examples for domains are avionic control centers or human resource management systems. Components developed by domain engineering has to be useful for a wide spectrum of system within a domain. In general, domain engineering has a broader meaning than "development for reuse", since the focus is not restricted to components only (on the implementation level) but is also on design and analysis documents. An important task is the development of a valid ontology for that domain. Furthermore, within a firm, more than one such processes could be established, depending on the different domains this firm develops software for.

Domain engineering subsumes all activities which are necessary to understand the major concepts of an area. Reifer suggests three main activities, *Domain preparation*, *Domain Analysis*, and *Asset Generation*. Similar process models for domain engineering can be found in [137, 102].

Domain preparation refers to the process of determining the scope and boundaries of a domain in order to define the major functions and capabilities within, to determine what functions and capabilities are excluded from it, and to identify existing interactions with external domains. Synonyms for the term domain preparation are *domain boundary analysis* and *domain definition*.

The notion of domain analysis was first used by Neighbors who defines it as "*the activity of identifying the objects and operations of a class of similar systems in a particular problem domain*" [161]. According to the US National Institute for Standards and Technology (NIST), domain analysis is defined as [114] "*the analysis of software systems within a domain to discover commonalities and differences among them.*" Neighbors´s definition differs from NIST's in the way that NIST emphasizes the analysis of *software systems* within a domain. We think that a domain exists independently from software systems and therefore we prefer the former one.

Due to its inherent abstract nature, the area of domain analysis is very complex and different approaches for conducting domain analysis are suggested in the literature . They range from top-down commonality analysis to bottom-up synthesis, where project information is gathered and abstracted which reveals failures and successes [94, 132]. If the reader is interested in further details, we suggest the work of Arango and Diaz [6, 5] as starting points.

The result of domain analysis is a *domain model*, which comprises requirements gathered in this way. A domain model represents a domain and depicts objects and relationships, functions and behaviors. The model identifies generic requirements or commonalities, as well as differences in the problems in a domain [133, page 170].

Asset generation is the process of building reusable high quality artifacts with respect to given standards. This does not mean, that they must be built in house. Any acquisition of assets is possible, ranging from buying commercial-of-the-shelf components (COTS) [30] to scavenging parts of legacy systems which have been proven to be of high quality by long standing operational use [182, 216, 41].

Development with Reuse – Application Engineering.

Development with reuse is the process of building software with an explicit emphasis on incorporating valuable assets. The more general notion of *application engineering* indicates that this development process guides the production of application software in a disciplined way. This is the standard process which software developers use in order to develop or maintain software systems. Here three main activities can be identified, *Requirements Analysis*, *Software Development*, and *Operations and Maintenance*.

Asset Management.

Application and domain engineering cannot operate independently of each other. The demand for analyzing a domain is normally driven by the need for an application within a domain. On the other hand, software developers use the results of the domain engineering effort by accessing the domain model and the reuse library. The asset management is responsible for various tasks in that environment. Important aspects are

library management: effort planing, request handling, demand analyzing, access control, financial control,

library population: negotiation with suppliers, classification, qualification, cataloging assets, and updating library holdings (which is a consequence of purchasing an asset),

library operations: configuration management, identifying troublesome assets, measurements, user support, information, planing of training,

library maintenance: maintenance planing, asset maintenance, and library reindexing (which can be performed periodically based on query profiles or after a population step), library interface adaption.

The cooperation between these three software development processes is depicted in figure 2.2. The high level activities are represented as ellipses. Rectangles represent knowledge bases, which are constructed and utilized by activities. The diverse relationships among activities and between activities and knowledge bases are shown as arrows[2]. Due to the two orthogonal (but not independent) development processes, which are supported by asset management, Reifer calls this

[2] Jacobson, et.al [108] defined a process model with three processes:

Application Family Engineering. All activities for designing and implementing a functional architecture according to a family of applications or a product line. This architecture is the basis for further software product development.

Component System Engineering. The set of activities for developing reusable assets. A prerequisite for doing so is an analysis of the requirements and their variability through the different applications of a family or product line.

Application System Engineering. The set of activities which build applications based on the common architecture by using existing assets

This view is very close to the proposal of Reifer, in so far as the first two activities of Jacobson and his co-authors coincide with Reifer´s definition of domain engineering activities. Application Sys-

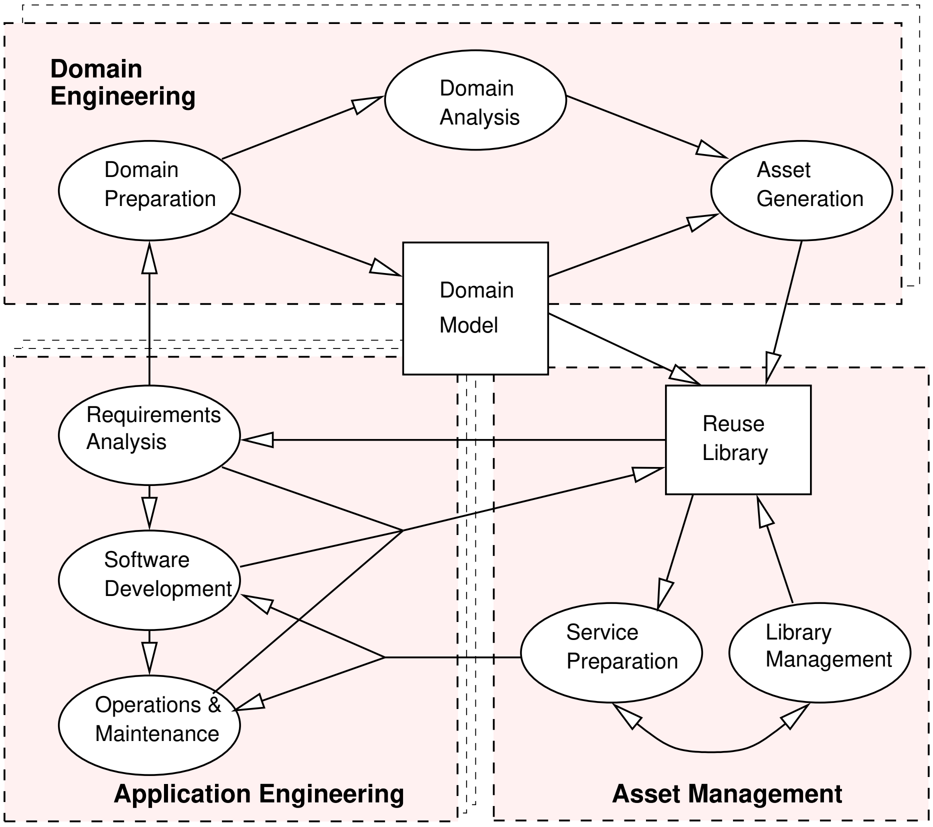


Figure 2.2: Reuse oriented software development (Source: Reifer [195]).

process model *two-life-cycle software process*. Domain engineering and application engineering are conducted in parallel.

2.4 Reuse Challenges

2.4.1 Organizational Issues

Although the software industry knows about the potential benefits from practicing reuse, the community is not quite sure about the promised gains yet [29, 74]. This is due to the fact that software reuse is not only a technical problem, but has to be

tems Engineering in Jacobson sense can be seen as similar to the application engineering defined by Reifer.

considered on the organizational level, too. Only when considering both aspects, a long term success can be achieved [159]. If a reuse program does not clarify the following issues, it will most likely fail:

Adequate investment.

Reuse does not come for free. The management must invest in acquiring, building, maintaining and upgrading reuse technology. Additionally, the continuous training effort for all developers involved must be part of the calculation.

Reuse group.

In order to gather expertise a specialized reuse group or domain engineering group must be established. In the hierarchy of the organization, this group is on the same level with any other development team. It is responsible for all reuse activities and forms a platform across all other project groups. It should create a foundation of shared software before any other development should begin [179].

Reuse incentive program.

Developers must be rewarded for successfully applying reuse technologies and adequate reporting and measurement schemes must be enacted. If the main productivity measure for the developers is a count of source lines, this measure will certainly deter developer from reuse, since an increase in the number of reused assets decreases the source line per developer productivity. Therefore, it is essential to establish a reward system which is comprehensible and fair [23, 22].

Standards incentive program.

Not only the incorporation of existing assets must be rewarded, but also the extra effort, which is taken to understand and apply standards, such as architectures. All this was normally accounted to the project's budget and has therefore usually been dropped under pressure.

Long term reuse strategy.

Within a narrowly viewed project, systematic reuse is unlikely to withstand the everyday pressure for long. Consideration of reuse must be (1) across related systems (*vertical* domains) and (2) additionally should allow for generalization across unrelated systems (*horizontal* domains). Here the product line approach tries to achieve the maximum benefit by taking both aspects into account [20].

Retarded return of investment

Reuse initiatives need a lot of resources which pay off only later in future projects. The cost of it must be shared by all ongoing projects. The benefit of a reuse program may not contribute to the economical success of the early-adopter projects. This may lead to resistance against additional expenditure.

2.4.2 Technical Issues

These organizational issues are very important, although there are still technical challenges being unsolved. The rapid changes in software development causes the belief that the main technical problems have been solved [179]; some of the "solved" problems, however, have been revitalized.

As a software development platform, the internet takes a center stage in all these changes [80, 136]. More and more projects are globally distributed and coordinated via the internet. This leads to the effect that people with different backgrounds and different mother tongues work on the same project, demanding the same understanding of what is going on.

On the other hand, components themselves may be distributed all over the world. This leads to the need for a new way of representing and deploying them [24], because certain assumptions necessary for understanding the components (language, organizational and cultural context) cannot be guaranteed to face up to every situation.

Due to the bidirectional nature of reuse, technical challenges arise in both aspects, domain engineering and application engineering.

Domain Engineering and Asset Management

The main objective of domain engineering is to build a domain model and to generate reusable assets. Whereas the domain model serves as a repository of core requirements, which holds for many applications in a given domain, the reuse library holds reusable parts for a domain, specified through informations from the domain model. A part of the task of domain engineering is therefore to identify and provide reusable items. Asset Management is responsible for considering the following technical questions based on the results of domain engineering.

Verification

is part of the quality assurance process. It is the task of determining whether or not an asset fulfills established requirements [183]. The requirements are given by the domain model. Furthermore, they must have a certain potential for being reused in the application domain.

Description

is the task of assigning a cognitive characterization to an asset which allows for representing the meaning of it without displaying all its details. The problem here is to keep the balance between including all important aspects of the asset without overloading the description with unimportant facts.

Asset organization

is the task of generating and maintaining a meta-structure which describes the relations between the assets. The purpose of this meta-structure is to support effective searching in a software library.

Application Engineering and Asset Management

Whereas domain engineering is concerned with gathering information about a certain field of operation, application engineering is the discipline of developing software products with respect to the domain. This view of development tries to emphasize the systematic approach to reuse, because the starting point of development is based on the foundation of the domain model. The interface to asset management then pertains the following aspects:

Searching

The process of asset management is in duty for providing the information about reusable artifacts during the whole life cycle of software development. During requirements analysis the analyst uses information from the domain model as well as from the software library. This indicates that a pure top-down approach to software development by emphasizing systematic reuse is not possible, since existing knowledge about the domain may restrict the search space for the analyst. The search for an asset providing the functionality wanted may be performed by browsing the content of the library as well. Here, a clear guidance for the searcher is important.

Furthermore, a situation might arise where the analyst recognizes a missing or wrongly defined element in the domain model. Then, he/she is responsible for triggering a new domain engineering process for reflecting the new situation (a bottom up situation).

The main purpose of asset management is to provide reusable assets in tight interaction with application engineering. Searching for an asset which provides a solution to her/his requirements, the software developer, expresses his/her need in such a way that the retrieval system of the reuse library is able to understand. This query can either be expressed in the same framework of description in which the assets are indexed, or it can be conceptually closer to the developer's way of thinking. In addition, the candidates obtained by a retrieval process have to be presented to the software developer in an understandable form.

Verification

After selecting assets, these artifacts must be verified to make sure that the retrieved assets comply with the query's specification. If this cannot be achieved at full certainty, at least the degree of trustworthiness in this compliance must be high. A significant part of the verification process is already being performed by the search process itself. Anyhow, the obtained assets have to be further analyzed very carefully with regard to the functionality being searched for.

Incorporation

After identifying and verifying an asset, the question arises: how can this asset be incorporated into the product? If the provided functionality of the asset meets the requirements completely, no functional modification is necessary and the asset can be used as a *black box* (verbatim, "as is")[3]. If the requirements are met only partially, modifications have to be performed. Such assets have to undergo a rigid quality assurance activity. This way of reuse is known as leveraged, adaptive, porting or *white box* reuse in the literature [72].

2.5 What is a component?

In the context of the two-cycled software development process described in section 2.3, this thesis is located within the technical part of asset management. The

[3] If a component should be integrated into a system as black box, the retrieval mechanism should not depend on interiors of them. This demands *plug-in compatibility* between query and component [69, 11]. See the glossary for details.

focus is on the process of reusing a certain type of asset, namely the components.

The general purpose of a component is its integration into a software system. It is a specialized form of an asset (the general term) which is produced during the implementation phase of either the domain engineering process (asset generation) or the application engineering process (software development)[4]. Although the term component is often directly associated with the object orientated paradigm (which supports the building of modular system well), components need not be objects [220, p 30ff].

There is no totally accepted definition of the term component [36]. The NIST reuse glossary [114] defines a component in a very general way as one of the parts, either hardware or software, that make up a system. This definition is too broad to virtually include all of the artifacts of a development process, such as test cases, source code or parameter files. This leads to the conclusion that the NIST's notion of a component is equivalent to the term *asset* as being used in this work. Additionally, the term component has not only been developed with regard to reuse, but also for supporting the evolution of a system. A highly componentized system is easier to maintain, since the localization of its parts has been simplified and the impact of changes are easier to analyze, which is due to the well defined interfaces.

2.5.1 Components as functional entities

In [104], Hopkins discusses several definitions of components. His useful conclusion is that

> "a software component is a physical packaging of executable software with a well defined and published interface."

Hopkins focuses on the physical manifestation of an executable object (not in the strict sense of the object oriented paradigm). A component must comply with a component model. Component models provide the infrastructure for the correct implementation of components, communication between them and distribution of them. Examples for component models are the Common Object Request Broker (CORBA), defined by the Object Management Group, the Distributed Component

[4] A component generated this way must comply with a component validation process, which is typically conducted in domain engineering. Therefore, a application-engineering-generated component is most often fed into the domain preparation process step of domain engineering.

Object Model (DCOM) from Microsoft, and the Enterprise JavaBeans (EJB) from Sun Microsystems [121, 209]. A very similar definition can be found in the book of Szyperski [220], who says that "*software components are binary units of independent production, acquisition, and deployment that interact to form a (part of a) functioning system.*" In addition, this definition stresses the executability, the closeness (at least via the use as binary packages) and the accessibility via interfaces of components. Additionally, it has been stressed that components do not have a per-se existence but form a building block of larger systems.

A set of definitions for components is provided by Brown et. al. in [37] as well. The authors collected definitions common in research and practice. Although focusing on different aspects, they stress the importance of an *interface specification*. The knowledge about the structure and functionality of a component may be accessible or not, but the interface must be defined explicitly, which includes a complete list of the services provided and how to access them, generic dependencies or possible error conditions. Therefore, an important finding of the authors is that components are inseparable from architecture. In accordance with the initial statement of the current section, Brown and his co-authors claim that object orientation is neither a necessary nor a sufficient precondition for realizing components. However, object technology makes the task to implement components easier due to packages, interface definitions and genericity. It is therefore often directly associated with component technology. Due to their rigid interface specification, components have clearly defined access points, which is an property enabling its reusability. However, components as such do not automatically make a reusable building block (a fact which is true for object orientated classes as well).

2.5.2 Components as abstraction vehicles

The term component is important in the field of *software architectures*, too. In general, software architectures are a means to capture the overall structural layout of a system on a high level, accompanied with principles and guidelines governing the system's design and evolution [49]. Certain styles for laying out a system are identified, such as pipe-and-filter structures, layers, event-based communication structures, client-server, or object-orientated interaction [82]. The basic building elements for establishing architectures are *components* and *connectors*.

A component (depending on the architectural style, synonyms for components are filter, object, process) is understood as a functional (domain) entity linked to other components by connectors. Connectors are interpreted as "uses" or "passes-data-to" [19] relations. Architectural components, depending on the architecture's style, mainly serve as a means to separate concerns and to abstract from the subsystems which were represented by them. One of the main purposes of architectures is to provide the designer with guidelines to reduce the complexity of the task of assembling components. Components in that sense are large grain subsystems and the architecture describes their interaction in that *specific* context. Hence, no complete description of the component's behavior is provided and presented in the interface description. But complete behavioral descriptions are essential for making components reusable! An architectural component is thus not reusable per-se.

This conclusion was drawn in [81] as well, where the authors describe obstacles to be overcome by naively putting architectural components together. The main observation turned out to be that their components (COTS e.g. an object oriented database system, a framework for building graphical editors, etc.),which were generally reusable, assume too much context in the form of infrastructure, integration, and invocation, so that the consequently introduced mutual dependencies could not be resolved.

Up to now, none of the views of components primarily stresses the aspects of reusability. This view is discussed in the following section.

2.5.3 Components as reusable building blocks

Bertrand Meyer [143] focuses on the (re)use of software components and defines them as programming elements with the following properties:

- Elements may be used by other program elements (clients), and
- Clients and their authors do not need to be known to the element's authors.

The first property excludes software, which builds a complete and self-contained system. As an example, a word processor, when used by a person does not have component property, but if it is built into a human resource management system for entering reports, the term component is used correctly. The same is true for embedded software, which cannot be considered a component. A piece of software directly controlling hardware is too specific as to be used in an other context without having been substantially modified.

The second property excludes a simple subroutine call, which can be seen as reusing functionality; but due to its ad-hoc property, such a routine is not considered a component. A component should have the potential to be used by other programmers as well which can only be the case if its interface and environment-dependencies are very well designed and documented.

In Meyer's opinion, components may be functions, modules, objects, or clusters of objects; they may either appear in the form of a source code or in an executable state. Assets from the requirements phase or analysis phase, test documents and test cases are not refered to as components, since these products cannot be used by clients directly. Components are building blocks providing important functionalities for a system. A "true" component is usable by software developers who build new systems which are not foreseen by the component's author. This purpose shifts the term component to the field of reuse again.

A four dimensional classification framework

Meyer classifies components according to the four facets of *software process*, *abstraction*, *integration*, and *accessibility*. Although he speaks of object oriented components, his classification is not based on the object oriented properties and it is also suitable for the imperative paradigm. Here, the characteristics of the facets are provided:

- The *software process* facet determines in which phase of a software process the component can be used. This ranges from the analysis phase to the design and the implementation phase. Please note, that Meyer does not consider test cases as components, which excludes subsequent phases of the software development process from his framework. Furthermore, this facet explicitly stresses components which are *not* programming elements!
- The *level of abstraction* describes the component according to its degree of generality. Different views are considered, which are *functional abstractions*, *groups of related elements*, *data abstractions*, *frameworks*, or *system abstractions*. A functional abstractions is the representation of a self-contained functionality represented by subroutines or functions which are known from traditional software libraries. A grouping is a set of gathered and arbitrarily related elements. Data abstractions may be data capsules in imperative languages or classes in object oriented languages, covering a data entity. A higher level of abstraction is achieved through frameworks, which have to be used according to a set of rules. The highest level of abstraction is achieved by the use of coarse-grained binary components. Due to the maturity of object orientated technology they are available on the basis of COM, CORBA, or JavaBeans architectures.

- The *level of integration* refers to the point in time when a component is integrated into a software system[5]. Meyer identifies three different points in time: *static execution* refers to the integration during compile time or link time (they are not changeable without a recompile), *replaceable components* are also integrated during compile or link time, but with a certain dynamic variability, and *dynamic components*, which are integrated during the execution time of a system.
- The *level of accessibility* derives from the form in which the components are available to the developers. Here, three different modes can be identified: Access through interface descriptions, where no source is available. Source code access only, where the developer has to understand the code in detail in order to determine its functionality. Information hiding is given, when access is provided through the interface and additionally the source code is available for inspection, discussion and correction. A compressed view of the classification schema is shown in table 2.1.

Facets			
Process	**Abstraction**	**Integration**	**Accessibility**
analysis design implementation	functional grouping data framework system	static replaceable dynamic	interface only source only information hiding

Table 2.1: A four-dimensional component classification framework [143]

A three dimensional classification framework

A much simpler framework is presented by Thomason [221]. He places components into a three dimensional classification space with the dimensions defined as *distribution*, *modularity*, and *independence of platform or language*. A software component can be located due to the absence of a certain feature (0) or its presence (1). In that sense, a monolithic system is rated [0,0,0], since it is non-distributed,

[5]Meyer refers to this strata as *level of execution*, which in our opinion is a misleading term, since not the execution itself but the point in time when a component is added to a system's functionality is of interest.

non-modular and dependent on a certain platform. A CORBA component is distributed. However, the implementation is often platform dependent, leading to a rating of [1,1,0]. The binary scale can be extended to a discrete or continuous one. However, then the semantics for a rating have to be defined carefully, which makes this extension rather difficult.

To summarize the discussion about the nature of components it can be said, that there are a variety of opinions and definitions in literature, depending on the starting point and goal of the authors. Hence, not all assets declared as components are reusable. In this work, a component is considered a product of the implementation phase of the software lifecycle which can be directly used as a building block to be integrated in more than one software systems.

2.6 Summary

In this chapter the main motivation behind investigating software reuse has been discussed. We identified *organizational* and *managerial* challenges as well as *technical* ones. Although the demand for reusable building blocks was already realized a long time ago, many issues remain open. An important finding is, that reuse does not happen per se: It must be embedded into the whole software development organization. We stressed the importance of differentiating between development with reuse (*application engineering*) and development for reuse (*domain engineering*) and highlighted aspects to make it clear, that these processes are not independent of each other. The chapter closes with a discussion about the nature of a *reusable component*.

The next section focuses on problems and challenges which arise from the two-cycled software process model presented in this chapter. These challenges lead to the main questions this work deals with.

3 Problem Description

3.1 Cognitive differences in asset descriptions

In our work we focus on the support of component reuse. In general, the existence of a reuse library is a key factor to success for component reuse. This library must be populated by assets, which are useful for the application engineer. Without a library of valuable assets, component reuse cannot take place.

From the domain engineer's point of view, a reuse library is the place, where the repository manager puts in a generated asset. From the application engineer's perspective, the reuse library is the place, from which he/she gets building blocks for the current development in progress. Both groups of engineers can work independently without affecting each other. There are no formal rules, which guide developers in producing asset descriptions and the abstraction hierarchy an application engineer is building (in his mind) does not necessarily reflect the domain engineer's perspective. Furthermore, as it is clearly observable in many libraries, an identical concept is often represented in substantially different ways [8]. As an example, consider the hierarchical classification of collection components in the repositories of the object oriented development systems of SMALLTALK and EIFFEL: They are completely different. Figure 3.1 on the following page shows the VisualWorks Smalltalk's collection hierarchy in contrast to the ISE EiffelBase's collection ([8, page 9]). Although both hierarchies classify the same abstract data type concept within the same domain, their structure is designed very differently. Not even the abstract data type's names are identically and only basic structures can be mapped to each other easily (although it is rather unlikely, that they really behave identically).

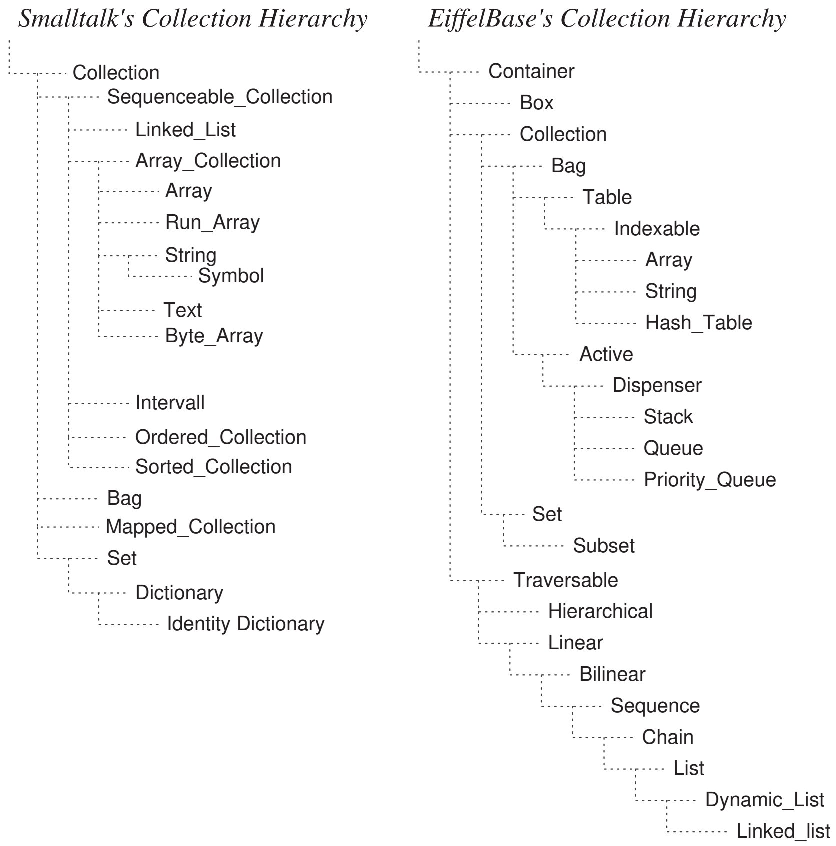


Figure 3.1: Abstraction variants in component libraries

This leads to the situation, that the components in the library are described differently. This difference might pertain to the structure, the interface, the constraints on input and output, performance needs, or the behavior of the components. Therefore, the domain engineer may not anticipate the expectations of the application engineer, leading to a conceptual mismatch in judging the component's place in the conceptual space of the application engineer. The following quotation of Reifer perfectly describes this problem of an user (application engineer) searching for an asset. He claims that

> *" ... users must have some way to judge the worth of the assets. You must make sure that the assets available provide the capabilities that*

> *your users both need and want. Finally, you must make it easy for potential users to find and access things of value.*" [195, page 9]

Due to the virtual nature of software, understanding it is in any case a complex task [66]. This work is concerned with closing the gap between the behavior of a component and the description a searcher (user, application engineer) produces to epitomize this behavior for a successful search. The next section discusses in detail the aspects causing these differences between an asset's description, its behavior and a query for it.

3.2 Challenges due to conceptual mismatches

This section is concerned with the identification of specific aspects of the divergence in the view of domain and application engineers. The discussion leads into a number of challenges a designer of an software retrieval system must overcome to establish successful asset management. The order in which the challenges are presented does not imply any weighting, since the priority depends on the specific context of the software development.

3.2.1 Seamless support for systematic reuse

In the reuse centric process model for systematic reuse, presented in section 2.3 on page 10, both processes can be architectured independently. For example, the domain engineering process may be iterative which emphasizes rapid prototyping [165], whereas the application engineering process may adhere to a spiral model [28] to reduce risks. As a consequence, the view on the purpose of a component is different as well. To generate reusable assets a domain engineer must anticipate future needs: Hence, she/he is interested in generating or acquiring reusable assets which are [109, page 118]

- general (to be usable in a variety of applications),
- canonical (to enhance understandability by standardized formats),
- variable and customizable (to ease integration and adaption).

Application engineers on the other hand are interested in specific solutions within the application domain. The assets they need must

- provide the solution for the required need,

- behave correctly, and
- run efficiently on the current platform.

Describing a component means to produce an epitome of its characteristics. What a "characteristic" feature really is, depends on the focus of the describer. Hence, it is very likely the description of the characteristics of a certain component differs substantially, if produced by the domain engineer or by the application engineer. As a consequence, although a component is available and thoroughly described, the application engineer performing a search might miss a suitable piece of software.

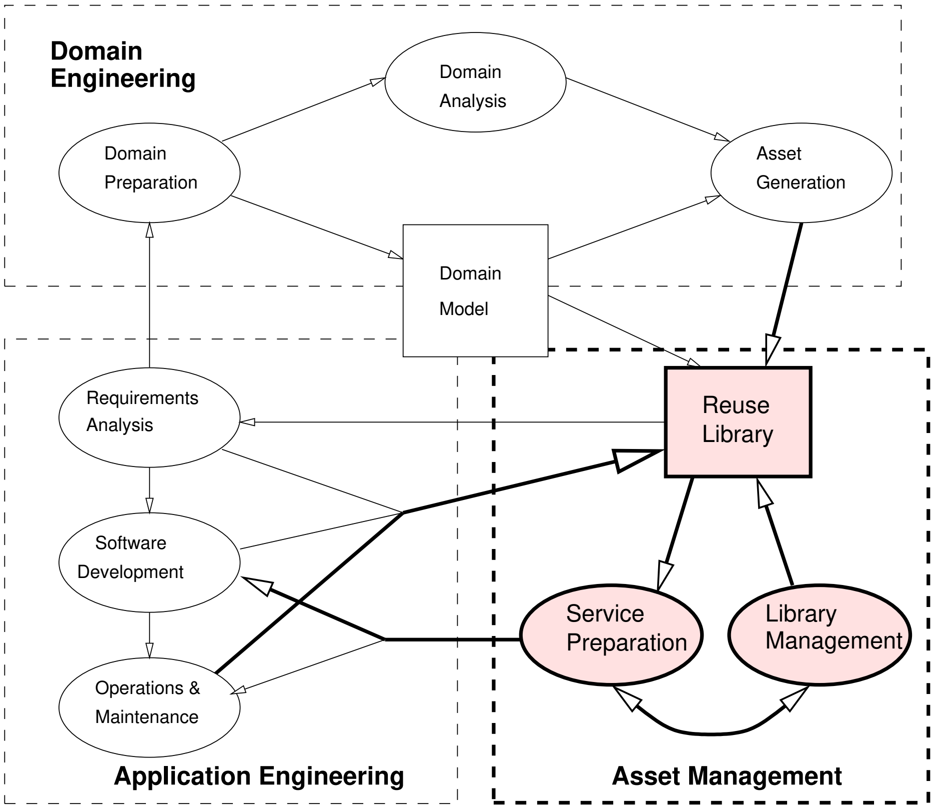


Figure 3.2: The problem context of this work

In figure 3.2 the problem space of this work is placed within the two-cycled software development process of Reifer (presented in section 2.3). It is located within its technical part, the asset management, and concentrates on aspects of organizing the reuse library, service preparation and library management. Although domain engineering and application engineering are logically separated, both rest upon the same premises: the meaning of the component's descriptions must be interpreted in *exactly* the same way. This does not imply that both groups must understand the entirety of a description, since their focal points are divergent.

Furthermore, this understanding has to be independent from technical and cultural circumstances. The description of the assets such obtained have to fulfill the following requirements:

- The domain engineer should be able to obtain or generate the description easily.
- The application engineer must be able to understand the description without the need for extensive training.
- The integration of the search method based on the description into the working environment of the application engineer must be seamless.
- The librarian should be able to produce descriptions in isolation whose quality is equivalent to descriptions produced by domain engineers.
- The administration of the description must be effective and maintenance of the reuse library should not be hampered by the description technique.

These requirements originate from a common ground which can be subsumed by the following challenge:

Challenge 1: How can people working in the context of decoupled software development processes generate component descriptions which are interpreted in completely identical ways?

3.2.2 Support for opportunistic reuse

Opportunistic reuse (or ad hoc reuse) is the style of integrating assets into new products without an explicit emphasis on a reuse process. Here, the responsibility to reuse is solely in the hands of the application engineer. In opportunistic reuse, the main tasks of the reuser after determining the need for a component is to locate it within the library and to verify, whether the component really satisfies the requirements. At a panel on the state of the practice in software reuse at the 1991 International Conference on Software Engineering, Matsumoto reported that although most often the number of assets in a domain specific library does not exceed a few hundred components, application engineers only use a small fraction of them without trying to explore the rest of the library [74] for further benefits.

In [229], the authors recognize that based on the experience with a certain software library, application engineers access its content in three modes:

1. *Reuse-by-memory*, where the developer knows by previous experience from the existence of a component. This component is reused very often, even without the support of a software library.
2. *Reuse-by-recall*, where application engineers vaguely know about the existence of a component without remembering details. Here, success is determined by an effective retrieval mechanism.
3. *Reuse-by-anticipation*, where application engineers only anticipate the existence of a component. In this case, the availability of a software library motivates the search. Consequently, if a component providing the functionality searched for cannot be found within short time, the application engineer gives up.

Whereas the first two access modes are inherent to opportunistic reuse, the third mode is the key to bridge the gap between opportunistic and systematic reuse. Here, the key to success is to convince the application engineers, that indeed the search for a component in the library is effective, and that the reuse process does not interrupt the development. A seamless integration of reuse processes, especially the search and browse activities, into the development process is important. James Neighbors states, that one of the main hindrances in using reuse technologies (libraries) is to understand the asset's abstraction [162, page 8]. He says that

> *" ... if we could provide a scheme where some of this burden (of using the library) were shifted back to the author/abstractor then it would have a significant impact."*

To do so, the developer must be supported by reuse-centered software development tools [229, 230, 231, 39, 106, 38, 100].

An other important finding is, that very often the developer is not able to specify the requirements for a component exactly. This uncertainty can be dealt with by applying two approaches:

1. The matching mechanism allows fuzzy query matching [111, 57, 198], where the distance between query representation and asset representation is computed.
2. The software library system itself is used as a interrogation tool. This can be accomplished by browsing the library and in doing so, offering decision support [68, 180, 87].

The issues raised here deal with the question of making reuse "natural" to the application engineer in such a way that she/he need not think about reuse, because it is part of the every day life. This leads us to the next challenge:

Challenge 2: How can a search technique support an application engineer in such a way that she/he doesn't need extensive training to access the repository effectively?

3.2.3 Maintainance and Search Support

When a domain engineer provides the library with an asset, her/his main task is already accomplished at this point in time. The domain engineer is not a specialist in producing precise and non-ambiguous description of the asset's behavior. This is the task of a specialized agent, the librarian, which might be a trained human, or a software agent. Based on the understanding of the librarian and the administration needs (indexing and maintaining the library) an abstraction is generated, which should represent the most important and most significant details about the asset in understandable form. Supporting administrative needs and providing correct abstractions are goals, which are potential trade-offs!

On the other hand, the application engineer (the developer) is a specialist in understanding application domain problems and to solve them in providing a technical solution to it. When the application engineer searches for an asset, although not being specialized in it, she/he has to generate a representative abstraction of the solution to formulate a query as well.

The difficulty of providing the correct abstraction stems from the different views onto components and their purpose. These views are different, since the process for domain engineering and application engineerings are orthogonal to each other, which is expressively depicted in the process overview of section 2.3. In figure 3.3 on the next page the problem of different abstractions, applied from different viewpoints is shown, a view which was already discussed in [155][1].

A query leads to the task of matching two kinds of representations. The first one is the abstraction of the asset residing in the software library, the second one is the abstraction of a solution of a specific application engineer's development problem. The question, if the solution itself is conceptual correct is important

[1] A very similar view onto the problem of query formalization and classifying reusable components can be found in the work of Albrechtsen [3] as well.

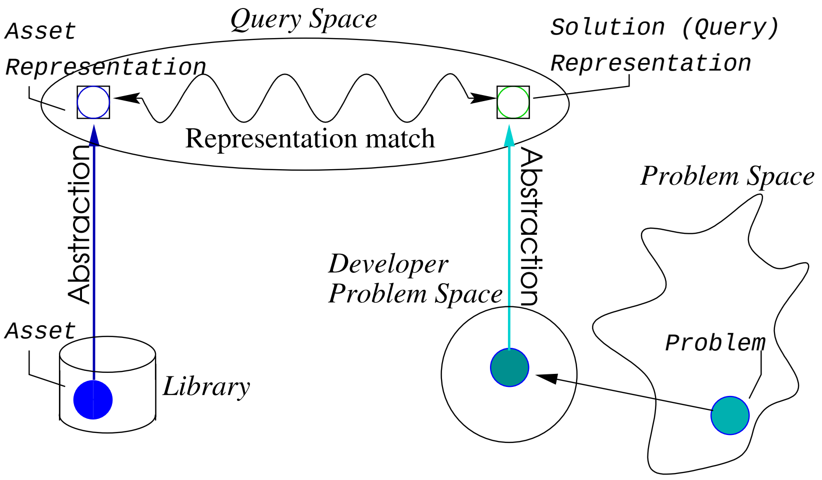


Figure 3.3: Different concept abstractions in component retrieval [155]

(the arrow from the problem to the solution in the developer's problem space in figure 3.3), but it is not discussed it this work. However, it is worth to mention, that the design of a solution and to develop a representation of it brings more uncertainty to the overall abstraction process.

A matching within the query space is performed on the basis of the respective abstraction representations. What is really intended is not a representation match, but a *conceptual* match, introducing the need for a flexible querying mechanism [70]. Therefore, the next challenges which must be solved, is the following

Challenge 3: How can a describer be supported in producing a stable description which allows

- to index and maintain a library without losing the description's expressiveness (concept) and
- to perform effective searches in the query space?

3.2.4 Obtaining descriptions automatically

Domain engineers know very exactly what functionality the implemented component performs, hence, other members of the same development team may describe the same functionality differently. This is due to different language capabilities, different concerns, different vantage points, or the effort spent to produce the description.

E.g., consider the following statements describing a procedure performing a bubble sort.

- "*This is a bubble sort algorithm that repeatedly scans adjacent pairs of items in an array segment from one location (front) to another (end). It interchanges those items that are found to be out of order. The array* `capacity[1 .. num_of_rooms]` *is sorted in ascending order by repeatedly placing the maximum towards the end of the scanned segment.*" [15]
- The following quotation is taken from a textbook for standard algorithms [208, p 100]: *"An elementary sorting method . . . is* bubble sort*: keep passing through the file, exchanging adjacent elements, if necessary; when no exchanges are required on some pass, the file is sorted.*"
- Since the specific property of bubble sort is its bad time complexity, this could be part of a description as well: "*Time complexity $O(n^2)$ is achieved by repeatedly exchanging adjacent items until the array is sorted.*"

It is not easy to recognize that these statements are describing the same algorithm and it is impossible to verify whether the such presented functionality is identical. A sort of standardization for component descriptions should help to smoothen individual differences.

On the other hand, documentation is a labour intensive task. A developer of a piece of software is no documentation specialist. Then, writing the description is considered annoying not creative and the quality of the result might suffer from that. Therefore, an analysis automatically automatically to produce some sort of descriptions would unburden developers and documentation specialists.

Both reasons mentioned here lead to the following

Challenge 4: How can component descriptions which precisely reflect their functionality be generated automatically?

3.2.5 Considering globally distributed software development

When software is developed in one location, the diversification of specialized teams regarding domain engineering and application engineering can be compensated because of the vicinity of these groups. For a large enterprise, developing software can involve several organizations and many teams at different locations, who almost never meet physically. Nevertheless, they must share information and exchange components across different environments and networks. Software engineering over the internet becomes a common way of developing software [98, 80, 136, 176]. As a consequence, teams are not only locally dispersed, but they work in different technical and cultural contexts. This leads to different styles of a component's description, depending on education, local habits and language abilities.

The internet as software developing platform plays a further role in the context of this work. Vendors offer components worldwide in a large number as solutions for specific problems [225]. With an vast amount of (even specialized) components easily available over the internet [207], the role of a precise description of the component's behavior plays an important role. This is even more important, if the component can be judged on the basis of its documentation only without giving the potential reuser the opportunity to take a look at its source code.

The questions raised here lead us to

Challenge 5: How can it be ensured, that a component's description only depends on its functionality, regardless the describer's personal, social, and cultural context?

3.2.6 Preventing effects of natural language ambiguities

Most techniques for describing the behavior of an asset are based on natural language. Natural language is inherently ambiguous [15] due to the lack of a rigid formal foundation of the semantics. If a component is not abstracted on a well-defined basis, the description can often be as difficult to understand as the source code of the component itself. Thereby the cognitive distance is increased [122, page 148]. This is the next challenge:

Challenge 6: How can a component be described non-ambiguously in a way which is natural for software developers?

3.2.7 Dominance of precision

In information retrieval two important measures for the performance of a search based on the judgment of users are common [204, 13]:

Precision is the ratio of retrieved documents which are relevant to the documents which has been retrieved. It is defined as

$$\text{Precision} = \frac{\text{Number of relevant items retrieved by the search}}{\text{Total number of items retrieved by the search}}.$$

Recall is the ratio of relevant documents retrieved documents to the relevant documents existing in the document basis. It is defined as

$$\text{Recall} = \frac{\text{Number of relevant items retrieved by the search}}{\text{Total number of relevant items in the search space}}.$$

The accuracy of these measures depends on the knowledge about the number of relevant items in the retrieved candidate set and about the number of relevant items existing in the software library. The majority of the software retrieval approaches discussed in the literature up to now are stressing the importance of recall [147]. This is justified if only a small number of candidate components is available in the library. With the situation described above, where an abundance of components is given, high recall is not important. Nowadays software is available via the internet and a very large number of components which are developed somewhere in the world [80, 207, 225] offer similar functionality. In such situations precision becomes much more important than recall [155]. If a application engineer retrieves a small number of high quality components which fit his needs, she/he does not bother much about components missed in the search. The important question then is to verify, whether the candidate set's components are worthwhile to be examined further, which leads us to the following challenge:

Challenge 7: How can a reuser verify quickly, whether components behave in an unexpected way?

3.3 Summary

The current chapter dealt with the technical aspect of component description and component retrieval. We identified various conceptual challenges to be solved by a retrieval technique. All of them can be attributed to the problem of correctly describing reusable components and interpreting these descriptions. Furthermore, requirements are identified, which has to be met, if a retrieval solution should be successful.

In the next chapter we discuss already existing software retrieval approaches and investigate to which extend they provide solutions for the identified challenges and where do they not help.

We tackle the challenges enumerated here in the sections 6 and 7, where new methods for automatically generating component description are proposed. How each challenge is solved in detail by this methods is discussed in section 8.

4

Related Work

In this chapter a short overview of component retrieval techniques is presented and their concepts and advantages resp. drawbacks are discussed. This not at all a complete survey, because we do look only on methods which allow for classification and retrieval and which are related to the problem context of this work (presented in section 3).

If one is interested in a more complete survey on component retrieval techniques, the papers of Frakes and Gandel [76], Krueger [122] and Mili et. al. [147] are recommended as good starting points.

4.1 Information Retrieval

Information retrieval (IR) is concerned with matching the needs of a searcher (formulated as query) against a base of documents [13, 214]. This match is performed by an information retrieval system. Due to the post world war II evolution of computers and the accompanied tremendous growing of the number of electronic documents, information retrieval systems are of raising importance and they use sophisticated mechanism to gain high recall and precision. The growing of the internet is a further main driver of this trend. Nevertheless, the origin of these systems are in the techniques librarians of conventional books are using. Librarians have been obliged to carry out bibliographic searches manually. They were supported by means like card catalogues or standardized universal classification schemes like the Dewey Decimal Classification [67]. Thus, information retrieval focuses on information embodied as texts.

IR is based on two main activities. The first one is the *indexing activity*, which deals with the task of computing a representation of the documents as well as the requests. The second one is the *retrieval activity* which is concerned with examining the documents and computing the relation to items expressed as a search query. But how these two activities are distributed between the time of entering documents into the repository and the time entering queries, resp. how the search activity is performed, differs substantially from approach to approach.

If one sees reusable components stored in a software repository in analogy to documents stored in a library, the glance to the field of information retrieval is quite self-evident. The next section describes information retrieval techniques which are applied to retrieve reusable components. Unfortunately, software components are not texts and as a result this analogy does not hold exactly[1]. Information retrieval techniques do not suit the needs of a component retrieval system completely [147].

4.1.1 Concepts

In this section basic technical concepts for component retrieval based on information retrieval are presented and discussed. It is important to state, that the retrieval techniques implemented in many software library systems are a combination of different aspects discussed here. Furthermore, the categories presented are not orthogonal. The performance of a retrieval system might be enhanced in combining them, if suitable. For example, boolean querying can be applied to an open description space or a set of limited keywords as well.

Boolean Querying

Boolean queries are characterized by clear and simple semantics. Due to that they are very common in retrieval systems. For component retrieval, the starting point is an abstraction of components in the form of natural language texts. This can be an abstract, the documentation, the pure source code or any thereof. The main idea is to operate on a set of abstractions of components (hereafter called index documents or for short, documents) and analyze the occurrence of words in them. The result of such operations is an unordered set of candidate components.

[1]In addition to their application in the standard field, IR methods can be successfully applied to domains which are not based on text. E.g., due to the well understood structures of chemical formulas or genetic codes, very efficient retrieval systems for them can be built. Nevertheless, software does not reveal this exact rigidity rendering the design of IRs difficult for this purpose.

In general, the component descriptions are broken into strings of words, without considering *stop words*. Stop words are terms of insignificant meaning, because they occur too often in documents (articles, prepositions, etc.). A word is contained in a document, if its string is contained in it. In the basic approach different forms or spellings of the same word are considered as different items. In a more advanced approach synonyms or flection forms (so called *terms* of a word[2]) are considered as well. The set of terms occurring in a document represent this document in the retrieval space. Every element of this set is used as *index term* in the retrieval system. Normally, index terms are organized as *inverted lists* [89], where for every term the set of documents is stored, in which this term occurs.

A simple query for set of documents containing a term is performed by locating it in the inverted list and presenting the set of associated documents. Complex queries are constructed by combining simple queries (A, B) in such a way that they are connected with the boolean operators $\wedge$, $\vee$ or $\neg$. This is realized by computing $A \wedge B$ as $A \cap B$, $A \vee B$ as $A \cup B$ and $\neg A$ as $D \setminus A$, where D is the set containing all documents. Further refinements can be achieved by introducing relational operators for numerical terms (e.g. $<$ or $>$) or positional operators e.g. for strings the operators *NEAR* or *WITHIN*.

Natural language based analysis

Stemming.

Although boolean query mechanisms are very efficient, many mismatches may occur. This is due to the fact, that the technique rests on natural language terms. Obviously, the possibly different forms of words cause problems. A major improvement is to reduce index terms to their radical words (a process named stemming), for example by eliminating plural forms.

For a highly regular language, *pattern based* stemming renders very effective. To do so, a set of rules and a set of exceptions to them, automatically reduces terms to their stems. Such a rule for English stemming may be “IES” $\rightarrow$ “Y”, which eliminates many plural forms.

If a language is subject to a high degree of irregularities (e.g. German), pattern based stemming on the basis of character substitution is not effective. In such

[2]This approach can be seen as the inversion of *stemming* which is presented in the next section. The notion *term* is normally not used in that sense in linguistics, however, as boolean querying has its background in logics, the terminology comes from there.

cases, a *lexicon based* approach is more adequate, where terms are looked up in a dictionary. Although lexicon based systems encounter difficulties with homonyms and synonyms and are more costly than pattern based stemming systems, they perform better.

Keyword restriction.

In the simple natural language based approach virtually any term of it is element of the description space (minus stop words) can be assigned to documents, If the domain of the application is stable and no future extensions are anticipated, this large and open space can be reduced to a narrow set of keywords. This set of keywords is fixed and the description and the query terms must be selected from it only.

Keywords enable a simple description of an asset. However, the reduction to a much simpler descriptor base is a domain abstraction process, where some ability to express information is lost. The person assigning keywords to an asset is limited in describing the particularities of the component. The consequence of this may be a inadequate or incomplete assignment or (to overcome the restrictions) assigned keywords which do not hit the target directly. Both result in a misleading abstraction of the component, hampering a successful search for it.

Classification.

Classification is the process of structuring items according to a formal schema. The resulting schema (also called classification) reflects one possible view onto the set of items. As this reflects the ordering of physical items very well, classification is the primary indexing method for books in a library. The main idea is to structure a set of items hierarchically into subsets, where each level of the hierarchy represents a certain granularity of generalization. Due to the nature of items, this is no easy task. Lung and Urban state, that the major problem with the hierarchical classification schemes (on high level or low level) is their inflexibility [133, 171]. Hence, if all subsets are disjunct, the classification can be easily performed and leads to a mono-hierarchical structure. But if it is possible for an item to be member of more than one class, the classification results in a poly-hierarchical structure. Henninger [96] remarked, that even in well understood and ostensible naturally looking domains like biological taxonomies, more than one classification structure is necessary to satisfy all needs for researchers in natural sciences.

On the basis of a classification the process of searching is performed mostly in browsing the hierarchy for matching components. This is effective when the viewpoint of the classification is conceptually near to the searcher. For example, if the classification of software components is based on the viewpoint of locality of production (continent, country, enterprise, department) a search for a certain functionality cannot be supported. On the other hand, if this search is performed to judge legal implications, the proposed classification schema might be the most effective one.

Thesaurus.

A thesaurus is a structured collection of natural language items and their relations. Relations are of different types:

- *broader* to state that an item is more general than another one;
- *narrower*, if an item is the specialization of another one;
- *synonym*, if two items are semantically equal;
- *related*, if two items are semantically near;
- *antonym*, if an item is has the opposite meaning of another item.

Complex collections of items based on similar relations are sometimes called *ontologies*.

In the thesaurus based approach a well defined subset of words of a thesaurus is chosen as *controlled vocabulary*. Items of this vocabulary serve as descriptors for components and only they are allowed to characterize components. Due to the underlying thesaurus further words are linked to the descriptors automatically. If a searcher enters search terms, these terms are matched against descriptors. The searcher may relax the queries by allowing broader, narrower terms, or synonyms searched for, a technique which is called *query expansion*.

Vector space model

Beside boolean search the vector space model is the most important information retrieval technique. For this approach documents are represented by descriptive vectors. On the basis of these vectors the similarity or the difference between documents can be computed easily [202]. Starting from a set of documents $D = \{d_1, \ldots, d_m\}$ and a set of index terms $T = \{t_1, \ldots, t_n\}$ every document d_i is described by its document vector $v_i = \langle w_{i_1}, \ldots, w_{i_n} \rangle$. Each entry in the vector list is called a *weight*, $w_{ij} \in \mathbb{R}$ describing the importance of an entry in the vector for a document. There exist many variants for the computation of weights, depending on local or global term frequencies [213, 201]. As an example consider an index term vector of the form $t = \langle$ACID, abstract, Ada, ..., class, ..., cyclic$\rangle$. A description for an Ada component c with object oriented features may be then expressed as $c = \langle 0.0, 0.9, 1.0, \ldots, 0.7, \ldots, 0.0\rangle$.

To query the document base, the searcher has to define a query vector $q = \langle q_1, \ldots, q_n\rangle$ from $\mathbb{R}^n$. To perform a match between the query q and a document vector v, the similarity is calculated, $s : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. There is room for much improvement in the vector space model. First, is is very likely that most of the weights of document vectors are 0, since normally the set of index terms is very numerous which leads to vectors with a large number of dimensions. To avoid a large list with most of its entries to be zero, the inverted list technique [89] can be applied here, too. Each descriptor entry is stored as an index key. In addition to the components their weights are stored as index entries, too. In that way a search result can be ranked according to the relevance of the retrieved component. Further refinements for improving the efficiency of indexing structures can be found in [13, 89].

4.1.2 Information retrieval reuse systems

Most commercial component retrieval systems are based on one or a combination of many different information retrieval techniques. Ruben Prieto-Diàz developed the faceted classification schema for reusable assets, which can be seen as a combination of the vector space model, classification and keyword description [184, 185, 187]. A faceted classification schema may consist of several facets and each facet may have several terms. Facets themselves can be grouped to obtain a better overview. In table 4.1 a simple schema with three facets for Unix components is given.

To classify a component, from each facet a term is selected which describes the component's property best. E.g., the vector $\langle$locate, character, directory$\rangle$ may be a good choice for describing the concept of the command `find`. It is not possible to describe a component with more than one vectors (single-entry library). To support the user and permit a certain flexibility the faceted retrieval might be enhanced through a thesaurus system.

In the system described by Prieto-Diáz, when a query does not result in a satisfying answer, a query expansion is performed on demand of the searcher. The aim

action	object	data structure
get	file-names	buffer
put	identifiers	tree
update	line-number	table
append	character	file
check	number	archive
detect	expression	directory
locate	entry	
search	declaration	
evaluate	line-number	
compare	pattern	
make		
build		
start		

Table 4.1: A simple faceted schema for Unix components (taken from [187])

here is not to enlarge the restricted search space, but to search for similar components if the query does not result in a direct hit. Similar components are candidates for white box reuse. For example, if a query ⟨*substitute, backspace*⟩ returns no hit, a query expansion may ensue to alternatives, such as ⟨*substitute, quotes*⟩, ⟨*substitute, blank*⟩, ⟨*substitute, digit*⟩, ⟨*substitute, tab*⟩, or ⟨*substitute, character*⟩. Each retrieved component must be analyzed further for a potential adaption to fulfill the required need. Prieto-Diàz improves his approach by introducing conceptual distance graphs providing means for measuring similarities among faceted terms. A conceptual distance graph is a hierarchical graph with vertices defining concepts and weighted edges relating such concepts. Conceptual distance graphs support query expansions through distance measures for terms as well.

A similar approach is implemented in the system developed in the ESPRIT project REBOOT (Reuse Based on Object-Oriented Techniques) [113, 210]. The project was aimed at the development of an integrated set of methods and tools to support re-engineering and reuse on the whole spectrum from organizational and managerial aspects up to technical aspects. The technical nucleus is a repository system with a faceted classification scheme with the four facets, *abstraction*, *operations*, *operates on*, and *dependencies*. The components in the REBOOT system are classified within the facets which are extended with a simple thesaurus with specialization and synonym relations. The relations are weighted (so called *term* *space*) and the weights are used for computing the proximity of terms to allow fuzzy retrieval. The retrieval process is based on the size of functionality. The main task is up to the librarian who has to build functional description from the components delivered by the application programmers. Additionally, the searcher need not use the classification based retrieval system, but may use an integrated software tool, the navigator, to browse the library instead of searching.

4.1.3 Evaluation

From the viewpoint of search space, information retrieval methods for component retrieval systems can be differentiated into two classes:

1. The search space can be completely unrestricted, which is the case with boolean search. Here, the documents are matched against arbitrary strings entered by the searcher. This approach is successful, if the searcher is aware of this open search space and there exists a common understanding on the consistency of the search space. Such a common ground is given if the domain vocabulary does not offer many variants for terms, e.g. in the case of mathematical functions. If this assumption does not hold, the risk of an unprecise description due to the ambiguous vocabulary is high. The effectiveness of indexing and searching then depends on the judgment of the person performing these tasks and not on the underlying technique.
2. The second class builds a very narrow search space by offering only a restricted vocabulary, resp. mapping terms of the search query automatically to such a vocabulary. Thus, it is meant to avoid the inherent ambiguity of natural language. However, since the intention of a searcher cannot be anticipated, various tuning option for the query mechanism must be provided, such as broadening or narrowing the query. Depending on that tuning which based on the understanding of the terms, the result of such queries may differ substantially.

   The combination of both approaches is attempted in the faceted classification. Depending on new requirements new term can be added to a facet if they do not conflict with existing descriptions. This comes with disadvantages. Due to the single entry concept of the faceted scheme, it is sometimes hard to find the correct place in it to describe a component and then often a new term is added. If this new term is not chosen very carefully, the faceted scheme becomes ambiguous. The user (indexer and searcher) needs a very clear understanding of the significance of the facet dimensions and their values. In the literature, one can find various opinions about the superiority of one approach over the other one [148].

In the context of global and organizational distributed development processes described in section 3 neither the first nor the second option offers an entirely satisfying solution. Both approaches rest on natural language and suffer therefore from the missing rigidity of descriptions. Obviously, software can be described by natural language, but this is only one (often partial) means to grasp the purpose of software (which is, after all, the utilization of knowledge [7] by executing it). In using natural language as basis, many important features of software, such as exceptions cannot be described in a clear and understandable form, although such features demonstrate characteristic behavior. If a search for components based on natural language is performed, the quality of the result in terms of recall and precision cannot be assessed:

- Recall cannot be assessed, due to the various meanings of descriptors, which hampers the exact interpretation of relevance. On the other hand, many descriptors for components can be missed, since without the evaluation of all existing items in the retrieval base, no calculation can be performed.
- Precision cannot be assessed, due to the vagueness of the abstracts. Neither an exclusion of a component from the candidate set can be claimed, nor an inclusion can be demanded.

Obviously, these arguments are valid for most of the techniques presented in the current chapter. But as information retrieval based reuse systems explicitly try to formalize the relevance criterion, we want to stress the relevance problem at this point.

4.2 Knowledge Based Methods

4.2.1 Concepts

Knowledge representation methods seem to be adequate for describing reusable software components. This can be attributed to two useful properties of these methods, *representational adequacy* and *heuristic power* [76, 77]:

Representational adequacy refers to the expressiveness of the representation. Key word lists, for example, have little representational adequacy, since there are no syntactic of semantic relationships between the keywords.

Heuristic power refers to the level of logical inference, the description representation can offer.

The benefit claimed for knowledge based methods is their ability to express different relationships between components [58]. This helps to gain a more holistic understanding of the component in contrast to information retrieval methods.

Semantic nets.

A special form of a semantic net is applied in *AIRS* [169]. The core of the *AI*-based *R*euse *S*ystem is built on top of a numerical case-based reasoner. It uses a set of heuristic estimators as distance measure between components. The description about components is stored as a set of features (similar to the vector space model described in section 4.1.1 on page 43) and it is administered in a knowledge base. Based on such distances, estimates about similarities of components can be computed. The authors suggest two comparator functions for that purpose, *subsumption* and *closeness*.

A component C *subsumes* some other components $C_{1 \cdots k}$, if it is directly realizable from them without having to modify any C_i. This can be realized by either aggregation (C is the aggregation of $C_{1 \cdots k}$) or inheritance (C is the specialization of any C_i)[3]. Subsumption relations between components are represented as directed acyclic graph. Each component renders as vertice in the graph, and an edge indicates that the source vertice subsumes the destination vertice. The weights of the edges are estimates of the effort needed to obtain the subsumer. Subsumption graphs estimate, how easy it is to apply black box reuse with given components.

A component C is *close* to an other component D, if it can be realized by modifying D slightly. In general it is not possible to judge the effort for such modifications automatically, and as a consequence the value of the closeness comparator depends on the quality of the estimation. To support this task, the authors propose a so called feature graph which is a representation of feature properties within a certain domain. As an example, for a feature *programming language*, the terms $\{$*Assembler, Pascal, Lisp*$\}$ are given. Then an expert, based on her/his knowledge and intuition, estimates e.g. the distance from *Assembler* to *Pascal* with 10, and

[3] In general, subsumption is more often associated with the notion of inheritance than composition. Although, in the definition presented in [169], the authors do not prohibit aggregations. The term *inheritance* should not confuse the reader. Here, no semantics of object oriented programming languages is meant, but the restricted form of specialization relations, common in logics.

from *Pascal* to *Lisp* with 5. The closeness relation between *Lisp* and *Assembler* was not estimated by the expert. Here, the relation *closeness* estimates the effort of transforming a program, written in *Assembler* to *Pascal*. This information is then represented in a feature graph.

Based on comparators represented as graphs, distances between components are computed and matchings with different thresholds as matching criteria are possible.

A well known method for knowledge representation is the semantic (or associative) net. Semantic nets are attractive, because they mimic the human way of associating different concepts. As a structure a semantic net is a directed cyclic graph with vertices representing concepts and edges representing relationships between concepts. It is simple to express knowledge with such graphs, because the semantics is based on the names (and therefore the interpretation) of relationships. The *CodeFinder* tool of Henninger [95, 94] is based on weighted associative nets to handle component retrieval in environments where inconsistent and incomplete indexing must be expected. The weights are calculated by the *inverse document frequency*, where less frequent terms are considered more discriminating than often occurring ones. Additionally, semantic nets are attractive for domain engineers for depicting complex ontologies [6].

Frames.

CodeFinder is only one of two tools developed by Henninger to support reuse. The second one, called *Peel* (Parse and Extract Emacs Lisp) is an knowledge acquisition program for automatically analyzing source text to extract Lisp function for the Emacs editor [96]. The information gathered is represented in KANDOR, a knowledge representation languages based on frames [58]. A frame is a data structure with slots and fillers, on which inference can easily be performed. Frames may be linked to other frames by the `ako` (a kind of) relation, which expresses conceptual neighborhood. Slots may be filled with scalar values or with other frames as well. A basic property of frames is inheritance[4]. In such a way, after specifying that e.g. `SelectionSort ISA sort` the frame

```
SelectionSort
reference       : sedgewick, robert (1993), page 96
complexity      : O(N log N)
```

[4]Because of the inheritance property frames are often considered as an object oriented knowledge representation method [64]

might contain following information in its expanded form (depending on the shape of frame `sort`):

```
SelectionSort
reference       : sedgewick, robert (1993), page 96
complexity      : O(N log N)
ako             : algorithm
operation       : ordering
operands        : data objects
```

The knowledge representation language KANDOR is the form of representation used by Devanbu et al. for their domain analysis tool *LaSSIE* (*La*rge *S*oftware *S*ystem *I*nformation *E*nvironment) [59] as well. They system was in use for helping application engineers in understanding large software systems (over a million lines of C code). LaSSIE is built upon a large knowledge base, a semantic retrieval algorithm, a natural language parser and a browsing interface.

Neural nets.

Neural nets are a flexible tool for different classification tasks and they can be adopted for reorganizing software libraries as well. Merkl et. al. [139, 140, 141] use Kohonen's self organizing feature map (SOM) [119] to detect clusters of relating components in a software library. A SOM is a two dimensional grid of neurons where all adjacent neurons are connected. The components are described by input vectors (see the section on the vector space model on page 43 for more details). During the training phase of the SOM a vast amount of vectors is presented to iteratively adapt neighborhood relations. Each vector is mapped to every neuron. The neuron with the output nearest to the input vector is the winning one, because it represents the software component (expressed as the current vector) best. The input connections of the winning neuron as well as all adjacent neuron input connections are adapted to reflect the current vector. In such a way, the library is represented as a two dimensional grid and related components appear nearby on that grid. A query to the software library is expressed as vector and the neural net determines the winning neuron for that query. This unit need not be mapped to a component in the library directly. But at least the adjacent neurons may hold components, which partly fulfill the searchers needs.

A similar approach is followed in the work of [33, 34]. Here, the inherent ambiguity of natural language is positively exploited to smoothen the differences of concepts. In contrast to other software retrieval techniques the system presented tries to get at least a rudimentary understanding of the concepts described. This process is called “shallow concept comprehension”. Based on features selected from the descriptions by applying systemic functional grammar theory, a combination of hierarchical fuzzy sets is built. These form the knowledge base on which a cascade of two neural networks (based on fuzzy adaptive reasoning technique and fuzzy associative memories) operates. The systems represents the knowledge base as fuzzy clusters and supports fuzzy retrieval.

4.2.2 Evaluation

Knowledge based software repository systems as presented above can be very powerful in their domain. However, one main disadvantage is that creating the knowledge base for a real world problem is very labor intensive making the knowledge acquisition process costly [76]. In the cases of a semi-automatic learning process, such as the unsupervised learning of neural nets, the situation seems to be better. However, the training intensity hampers an efficient application for real world problems, because any maintenance operation in the knowledge base needs a full (re)training of the net to reflect the new situation correctly. As a consequence, knowledge based approaches are restricted to stable and very narrow application domains [6].

Furthermore, semantic nets base on formal ground and are nevertheless very flexible for representing knowledge. This flexibility is due to the use of natural language for expressing basic concepts. This demands a clear guidance to build and interpret them. For example, AIRS demands the correct estimations of a domain expert for closeness and subsumption relations. Ironically, no clear semantics for such an interpretation is given, leading to the same challenges which information retrieval methods in general are facing (Which was already discussed in section 4.1.3). This makes an standard inference process for semantic nets impossible.

The SOM approach reveals the disadvantage, that depending on the initial random distribution on weights, the classification of the same set of components might differ substantially when it is repeated. This is a severe drawback, if the library has to be maintained. Furthermore, the criterion on which the classification is based is randomly chosen. Additionally, the training itself is time consuming, since many thousand training iterations must be performed to generate clusters [139]. Similar to the knowledge based retrieval mechanism discussed previously maintenance is difficult. The same arguments hold for the fuzzy set based approach.

4.3 Hypertext based Methods

4.3.1 Concept

A hypertext represents a non-linear form of documents. Basic building blocks for hypertexts are links and nodes. Nodes are associated with a unit of information and can be of different types, depending on various criteria e.g. the class of data stored (plain text, graphics, audio, video, executable program) or domain object represented (customer account, balance sheet, reservation)[5]. Links represent a non-symmetric relation between nodes leading to a directed graph. Within nodes one can define anchors, which serve as source or target of links. Users are able to move through a hypertext document by following links embedded in the document, an activity which is called *browsing* [157, 164].

Hypertexts are well suited to represent the complex relationships between and within software components. Browsing supports application developers to learn about the contents of a software library. An example is the `javadoc` tool, a program coming with the `Java` development environment[6]. It allows to generate automatically a hypertext document from the object structures and documentations found in java source code. The description is rather simple, because it does not allow to define different views to the software components or an hierarchical structure other than the class' inheritance structure or object associations. Software development environments with emphasis on reuse based on the hypertext concept are presented in [78, 106]. Here, the structuring mechanism of hypertexts is used to present to the user a guided tour leading to the assets of the repository. Biggerstaff explored the use of hypertext to aid large scale reuse [76]. Based on the experience gathered in building a multitasking window manager he claimed that hypertext representations aid primarily in understanding a system without providing the ability to support component retrieval directly.

[5]This is the reason why nowadays the term hypermedia is more adequate to describe this form of information organization

[6]`Java` and `javadoc` are trade marks of Sun Microsystems, Inc. More information about `javadoc` can be found at `http://java.sun.com/j2se/javadoc/index.html`, current August, 2001

4.3.2 Evaluation

Two aspects hamper the acceptance of hypertext guided software repository systems. First, users are not willing to spend the extra effort for learning a new system. A user working with hypertext tends to get lost in the information space without strictly defined paths. The guided tour approach is not easily integrable into the day-by-day work environment of developers [56] because it demands a knowledge about the intention of the repository developer. Second, the performance for such a system in terms of the time spent from starting the search to the point in time when a result is available, is rather poor [106]. Both facts lead to a strong resistance of developers to use hypertext based retrieval systems extensively, when this is the only way to access the repository.

4.4 Formal Specification based Methods

To overcome the vagueness of description methods based on natural language concepts the research community also looks on more formal methods to support exact content-oriented access to software repositories. The idea is the same as it is in information retrieval based approaches to component retrieval: to describe software components and retrieve them. As a means for description a formal specification is chosen. A formal specification is a language using mathematical notation to describe the behavior of a software component. Often the way *how* such functionality can be achieved is not mentioned in detail, the specification only formulates *what* has to be performed. It is mathematical, hence no speculation about the meaning of phrases in worded prose is needed [217].

4.4.1 Concept

Either a formal specifications of a component is generated by the provider of the component or the component is analyzed automatically to derive a formal structure from it's code structure. This specification of its relevant parts serves thereafter as an index for the component. Specifications may be related to each other by matching or refinement relations expressed by a logic formula. In such a way a hierarchical index can be established, e.g. in the form of a refinement lattice [149]. A hierarchical ordering of components makes a great difference to the flat indexing structures of information retrieval based system. In such a way a conceptual view to the library allows to stepwise restrict the search space by refining the index specification according to the index hierarchy and the searcher's demands.

Specification based retrieval allows formal specifications as search keys and retrieves all components from a library whose indices satisfy a given match relation with respect to the key [68]. Specification matching can be used to aid a searcher in browsing a software library systematically, because formal specifications allow to establish relationships among them.

In the literature a variety of component retrieval techniques based on formal methods is proposed. Jen and Cheng [110] present a method based on order-sorted predicate logic (OSPL). In their system components can be searched in exact, relaxed or logical modes. In [48], Chen and Cheng specify components with Larch, a model-theoretical specification language. They define a generality relation which explicitly captures the semantic obligations for an existing component to satisfy a requirement. This relation can be used to evaluate and select automatically reusable components. This technique is prototypically implemented and uses a Larch theorem prover (LP) to perform the search task.

A similar approach is presented in the work of Atkinson et al. [9, 10], where the authors define a meta-model for component retrieval using the $\mathbb{Z}$ specification language. The behavior of a component is described as an invariant of the sequence of output values. Following the ideas of [149], a lattice of behaviors together with operations on that lattice, is defined. The searcher then specifies the desired behavior as a predicate on the sequence of outputs. In such way exact and relaxed matches are performed.

Not directly aimed at the improvement of reusability, but at automatically generating documentation for code components is the approach presented in [15]. The authors focus on loop analysis, where a formal specification is generated semi-automatically by utilizing user-supplied formal loop annotations. Their tool verifies these trial specifications, thus building a complete specification in a stepwise abstraction process. The such gained knowledge is formalized as "plan" which allows the detection of stereotyped code patterns. These formal specifications could be the input to formal specification based retrieval techniques.

Representative for formal specification methods, in the next section the relational specification based approach is presented.

Relational specification

In [149] Mili et. al. specify the functional requirements of a program by defining relations between data elements. Data elements are defined in the program space S and are expressed as variables. The relation of a specification contains all input-output pairs which represents the program's behavior. This allows specifying deterministic and indeterministic behavior as well. Indeterminism is described by assigning more than one outputs to an input. Since in most cases the program space is too large to enumerate all input-output pairs explicitly, the relations can be stated implicitly by predicates defined on S.

To illustrate the principle, let the program space hold a variable of type real, $S = \mathbb{R}$. Then, for the input value s and the output value s' the relation $R_0 = \{(s, s')|s \geq 0 \wedge s'^2 = s\}$ defines a square root function. To enable a hierarchical indexing structure on these relations, an ordering between them has to be defined. It bases on a meta relation *refinement*, which is a relation on relations. A relation R is a refinement of a relation R', if it contains more specification information. If a relation $R_1 = \{(s, s')|s \geq 0 \wedge s'^2 = s \wedge s' > 0\}$ for non-negative square roots of the argument is defined, R_1 is a refinement of R_0, since it is more specific than R_0. Based on the refinement, components in a library are organized according to the refinement lattice. Retrieval is performed in specifying a query in the form of a relation. A theorem prover then locates the components by proving that the specification of a library component fulfills the requirements specified in the query.

Kotschnig [120] developed a prototype library of relational specifications. In his system implemented in PROLOG he differentiates between interface and functionality which helps to organize the software library and supports retrieval. On the basis of relational specifications, Mili [146] developed a specification model of generalized specifications. This model and the software normalization theory developed by Mittermeir [151, 152], was used by Stopper [218] to describe a method to normalize objects for getting reusable building blocks. Here, application domain objects are normalized to raise the level of generality and to transform them to a application independent reusable form. Software normalization (function normalization and process normalization) aim to provide the domain engineer with a set of rules to judge the reusability quality of components. Software normal forms follow the normal forms of normalization theory known from database design.

Such systems can be improved with respect to recall by allowing variations in the matching granularity with *exact* or *approximate* matches [111]. In [211], Snelting et.al. described the software engineering system NORA/HAMMR, which rests on the formal specifications of the components in its repository. They identified one main hindrance for successful reuse based on deductive inference which is stated as *Fourth Reuse Truism* by Kruger [122]: “You must find it faster than you can rebuild it!” Snelting’s approach uses filters applied during the deduction process to represent intermediate results if they are sufficiently precise [211]. In their subsequent work they sped up the retrieval process by parallelizing the deduction task in refining their filters and using different automatic theorem provers [11]. Furthermore, Fischer [68] suggested to shift the time consuming task of theorem proving completely from the retrieval phase to the indexing phase. In such a way, a browsing structure is built, which can be navigated by a searcher without being in danger to lose her/his patience.

4.4.2 Evaluation

Formal specifications are based on rigid mathematical theories expressed in a mathematical language. Such languages are not familiar to the majority of practicing software engineers. If the standard description for components in the software process (domain and application process) is not a formal one, engineers must be trained to let them handle formal specifications. If this is only to enable indexing and understanding of reusable components this effort is wasted. Not even a semi-automatic approach ([15]) helps, since here the base annotations must be provided as formal specification. Clearly, they are tiny, but for every loop task exceeding trivial work, such specifications become quickly cumbersome.

On the other hand, if the task of specifying the component’s behavior remains with the software librarian, the burden of producing formal specifications for querying the repository remains at the application developer. As we demanded in section 3 as challenge 2, a component retrieval method has to be intuitive and must fit reasonably well into the working process of software developers. Descriptions expressed as formal specification cannot solve this challenge.

Let us suppose, the specification language is familiar to software engineers. Then a query is expressed as partial specification and the retrieval is performed by proving for all the components’ specifications, whether they partially fulfill the requirements. This task is very time consuming and indeed a bottleneck for an interactive search process. Although some suggestions for optimizing interactive theorem proving or caching this step [206] are proposed, as far as we know, there are no promising solutions on their way.

One important argument against formal specification as software description comes from Leavens and Ruby [128]. Formal specifications capture only one facet of component behavior, its function. But there are many other facets, which might interest an application engineer, such as time or space complexity, security issues, aliasing, etc. This restriction to one dimension (that of functional behavior) may be hindering successful specification based component descriptions severely. A retrieval mechanism should allow to specify and exploit information which is orthogonal to the main focus of the specifier.

4.5 Signature Matching

Up to this point, all description techniques presented rest on additional information added to the components. This information is provided either by the developer, the librarian, or by some analysis performed on supplementary materials. The techniques presented in current and the following sections do not need additional knowledge. They exploit inherent properties of the components directly.

4.5.1 Concept

Signature matching uses the structure of the interface of a component as built-in information for indexing software libraries. This is done without taking the detour to natural language. Although most of the work described in this section is concerned with handling signatures of simple components, such as single functions with in-, out-, and in-out parameters, more complex ones can be addressed as well. Such components can be objects or modules with internal data structures and public methods. Handling complex components is performed by extending the basic principles. How this can be performed is discussed in section 5 in more detail.

Mostly the concepts of signature matching are demonstrated with functional languages [232, 233]. Functional languages ease the task of signature handling due to their relaxed typing system. Albeit, as it is shown in the work of [211, 206, 50] functional signature matching is transferable to strongly typed imperative languages as well.

A *signature* is an implementation independent representation either of the type expression of a data structure of a function, or of a module's interface. See section 5.1.1 for a detailed definition of the notion of type. A module's interface holds a set of functions and/or data types. Signatures are usually statically checkable. As an example, look at the following C-function:

```
int randomNumber (int from, int to)
```

The signature of this function is generated in determining the data types on the input side, on the output side, and the data types of variables used as input and output parameters. If a parameter of a function has the passing mode in-out, this data type appears on both sides of the signature specification. The following signature is a representation of the previously presented C-function:

$$randomNumber =\!\!\rightarrow ({}^{from}/_{int},\ {}^{to}/_{int})({}^{randomNumber}/_{int})$$

Zaremski and Wing [232] define a signature match as a function with the signature `QuerySignature` × `MatchPredicate` × `ComponentLibrary` → `ResultSet`. Given a query, expressed as (partial) target signature for the searched component, and a match predicate defining the type of match and a component library, the signature matching process returns a set of matching candidates. The authors discuss different levels of matching, established by different matching predicates.

A further mechanism for query refinement is then to set a lower or an upper bound for data types in the type lattice. For doing so, a searcher may formulate a query by specifying a subtype or supertype matching predicate [222, 156]. As an example, consider the search for a function f, demanding an integer as input. If the searcher restricts the search to $f : \mathbf{int} \rightarrow \tau$ (where τ is an arbitrary type variable), a function $f : \mathbf{real} \rightarrow \tau$ cannot be found, although it may be a good candidate (at least for white-box reuse). Therefore, as input parameter, a supertype designator is assigned, determining all supertypes of **int** in the signatures of candidate components as valid inputs. The matching algorithm is based on a recursive execution of renaming, substituting, permuting or uncurrying (disintegration of structural elements) steps to locate matching signatures in the library.

Since signatures are an implementation independent view onto types, they can be sorted according to their sub- and supertype relations. This ordering lattice is used as a primary indexing structure for a software library. In such a way, a matching process for signatures with different rigidity ranging from exact to relaxed match can be supported efficiently. Similar approaches are discussed in [47, 129].

Novak [166, 167] describes a further application of signature matching for supporting reuse in the area of functional languages. In his application development system based on the GLISP compiler, different isomorphic views on data types can be established. The integrity of views is established by adding rules to views. Thus, the reuse of generic data types for applications is eased, because the application engineer does not have to consider the details of a mapping from the generic to the concrete and specialized data type. As an example consider an generic data structure `circle` with a `radius` field. If an application engineer has to deal with "pizza" and diameter, s/he can use the `circle` data type to store the data about the diameter without considering the internal data structures and conversion rules. This is possible, because the system offers an appropriate view with read and write permissions for the application programmer. Due to an integrity rule for the established view, the conversion from diameter to radius is performed automatically. Such views can be introduced by the application programmer, too, to enable additional flexibility.

Another approach based on signature matching is presented by Luqi and Guo [134]. Here, the authors describe reusable components by computing a *profile* from the components signatures to identify them uniquely. A profile is a sequence of integers generated by counting the number of signature types, the cardinality of related type groups and unrelated type groups, and a flag (0 or 1), which indicates an "in" or "out" parameter. This sequence of integers is the basis for an efficient indexing structure. Also here, performing partial or exact (here called full) matches is possible.

4.5.2 Evaluation

It is obvious, that signature match may not lead to the component which is really searched for: many components in the library could share the same signature. For example, in the standard ANSI C math library 31 out of 47 functions have the signature *double* $\rightarrow$ *double* [233]. Thus, in searching the functionality behind the signature and with a large set of components in the candidate set, this set needs further refinement. Signatures describe only a very narrow structural aspect of a reusable component.

Although signature matching bases on a formal and rigid description, many more questions concerning the interpretation of the query semantics arise: How can optional parameters be handled? What about polymorphic parameters, where the type is defined as to be generic and therefore is not known exactly until runtime? What if the searcher does not know about the ordering of parameters, since some signature matching approaches depend on positioning information? What, if not all parameters are known? To subsume, formality does not help in such cases, where this tool works on unstable ground[7]. Most of such questions can be handled in making various design decision for the matching algorithm. Anyhow, a heavy burden is left to the searcher in determining how to optimize the querying mechanism. This brings a lot of uncertainty, counterbalancing the benefits. Nevertheless, signature matching is a first step towards a interpretation-free, automatic software classification method. We will discuss our extension towards generalized signature matching in the section 5.

4.6 Extensional Descriptors

4.6.1 Concept

Podgurski and Pierce in [174, 175] present a technique to retrieve components from a library by using directly the main property of software which distinguishes it from other knowledge sources, its executability. They named this technique *behavior sampling*. A component library using behavior sampling contains software, which can be executed within the library environment automatically. The retrieval process rests upon signature matching and is performed in two steps:

1 The searcher specifies a target interface signature with data types for all input and output parameters. This signature is seen as partial description of the searched functionality (see section 4.5 resp. section 5 for a more detailed discussion on signature matching as well). Parameters need not be named and the ordering of parameters need not be known. The retrieval mechanism then searches for components conforming to the interface signature and collects matching components for a candidate set. If generic routines are encountered, a mapping from the concrete target interface signature to the candidate routine's signature is computed, if possible.

2 The searcher specifies the required behavior of the component by providing a sample of inputs and the corresponding correct outputs. In basic behavior sampling, an operational input distribution based on the input value's frequency of a real-world use supports the search. All components of the candidate set are executed on the specified inputs and the results computed in such a way are

[7]This fact is expressed very clearly by John von Neumann: "*There is no sense in being precise when you don't know what you are talking about.*"(cited according to Jackson [107, page 290])

matched against the required outputs. Exact matches indicate (partially) fitting components, whereas a mismatch leads to the elimination of the component from the candidate set. Podgurski and Pierce conducted experiments, in which they observed that in most cases only 4 samples suffice to identify correctly a component in a library of 252 routines. In worst case, the searcher has to generate 12 sample descriptors for successfully locating a component [175].

Component retrieval by behavior sampling leads to a high precision (with respect to the query sample), because only components in conformance with the sample are kept in the candidate set. Partial or relaxed matches are not considered. The query itself is formulated directly in the context of the problem to be solved, thus it needs no interpretation. If recall is taken into account, the situation differs. Due to the fact that signature matching is an essential part of the retrieval process, the query result depends heavily on the correct formulation of the target signature.

4.6.2 Evaluation

Executional description according to behavior sampling is very helpful for component retrieval, because it rests on an inherent property of software: executability which is rendered by transformation of data. Unfortunately, some aspects hamper the application of this technique in practice:

Rigid signature matching

The first step of behavior sampling process is signature matching. As Podgurski and Pierce already noticed, components with identical functionality may operate on very different data structures. As an example, the reader might consider the data type `set`, which might be represented as bit vector, linked list, tree, or hash table. If a searcher does not know about the domain engineers implementation preferences, she/he must define the whole variety of implementation variants as part of the query in the target interface signature. Albeit, the searcher cannot be sure, if there exists an implementation using an unconventional and therefore not targeted data structures for representing a set. The authors then suggest to specify both the target signature and the component signature as abstract data type. An abstract data type hides implementation details and defines signature details as generic operations on data. In such a way, the `set` properties are expressed on the basis of its sorts (type names) *ordered set*, *element*, *key* and *boolean* by specific operations. E.g., the operation `newset` returns an empty new set and `insert(S,X)` takes an element `x` and puts it into a set `S`. For a complete specification nine operations are required. How this abstract data type is linked to concrete implementations must be specified for all implementation in order to perform component executions automatically (similar to the ideas of view integrity of [166, 167]). To ensure a correct mapping from abstract types to concrete types is at least a cumbersome task.

Abstract data types help to handle implementation variants, but to correctly specify signatures and to map them to the implementations is laborious. Additionally, to determine if a data structure is indeed a correct implementation of a concepts (e.g. set) or embodies a different concept, is also troublesome. Either the domain engineer or the librarian could perform this task. Both have to be experts in formulating abstract data types. Furthermore, if not all parameters are known to the searcher, heuristics must be embodied in the retrieval mechanism to handle such vaguenesses.

Low recall

Another critique is expressed by Hall [92]. First, he addresses the drawback of low recall, because other components with different signatures may be relevant to solve a programming problem, too. He extends behavior sampling to a *generalized behavior-based retrieval* (GBR). His technique does not consider components only, but whole programs built from reusable components. According to Hall, this should improve the degree of recall, since in addition to a focus on small routines, a higher level of problem solving is reached by retrieving programs incorporating relevant subroutines.

On the other hand, retrieving whole programs can lead to a small degree of precision, since not all parts of the offered functionality might be useful for a searcher. Furthermore, if the program itself does not offer the desired behavior exactly, it cannot be used *as-is* and only white-box-reuse is a alternative in these cases. However, whether the program really can be disassembled to get the components and if these components behave correctly out of the program's context, is another question.

Side effect treatment

Hall criticizes that with behavior sampling components generating side effects are hard to handle. For example, if a component for file deletion is sought, the execution environment must know how to restore the file after each candidate component's execution. This is hard to automate for a general environment, because all occurring side effects must be predicted correctly. GBR solves this in executing not the code itself, but a side-effect-free functional model of it. To provide such a model for each component of the repository is the task of the domain engineer or the librarian. Hall does not discuss the problem how to verify if the functional model is indeed capturing the whole range of the behavior of the code.

Operational profile focus

The operational input distribution used to select input values serves to generate characteristic tuples. The assumption here is, that frequent values for searcher represent typical behavior within the application domain and in that way is easy to handle. But Podgurski, Pierce and Hall already recognized, that a randomly chosen input sequence performs equally well compared to any other sequence. We think, that a searcher (as an application domain expert) finds boundary values more helpful and significant and the output on seldom, but remarkable input has more characteristic power than frequent input-output tuples. For example, consider a function performing a translocation computation on two-dimensional structures on a drawing sheet. Here, a frequent operation is a translocation within the boundaries of the drawing sheet. But more characteristic and important is the result of the computation, if the structure is moved over the edge of the sheet. Such rare events are more characteristic and, therefore, more important to the searcher than a number of further inner-sheet movements of the structure.

Output determination

How does a searcher specify functions with sample descriptors, if the output is not know, resp. cannot be determined with reasonable effort? Then, the search cannot be performed. In addition, consider a requirement, that the structure of the concrete output is not important, only side effects of the computations are important. This may be the case, if one searches for a parser component. How the resulting abstract syntax tree is represented is indeed of minor interest, if only the parsing itself is correct. In such cases, retrieval should be possible anyhow.

User interactivity

Formal methods were already criticized because of their insufficient performance and the resulting unsatisfying interactive search progress. The same argument holds for behavior sampling and generalized behavior-based retrieval. Not only the execution of all candidate components and the output matching is time consuming. The precedent phase of signature matching is time consuming, too. All of these factors together inhibit an interactive query-answer cycle.

Repository maintenance

Each query to the repository may activate any component. Each component therefore must be immediately executed on the presented input to keep response time low. In the original work of Podgurski and Pierce this renders not ineffective, since the granularity of the functions administered was rather uniform and the number of functions was low. In a productive environment the situations is different. For any component there an execution environment must be provided. In the worst case the execution environment's system architecture might be different from the one given for the repository system. This technical context can change (operating system or hardware migration), and any change leads to a inconvenient verification process to ensure the executability for each component under the new circumstances. This process has to be repeated for all partitions in the repository. Therefore, the effort to maintain a direct execution of all reusable components is high, since a library of reusable components is a long-term investment.

4.7 Summary

In this chapter we presented important techniques for component retrieval and we discussed them especially with respect to the requirements discussed in chapter 3. Among all of them, we see a combination of signature matching with behavior sampling as a trustworthy ground to work on. However, both techniques demonstrate some disadvantages which must be considered. Hence, in the following chapter 5 we discuss a generalized approach to signature matching and in chapter 6 we use this generalized signature matching technique for improving behavior sampling. The approach enables us to describe and retrieve components built upon the procedural paradigm. Chapter 7 deals with the extension of the ideas of interpretation-free descriptions (and retrieval based on them) to handle complex and state-bearing components as well. Thus, the developed technique is useful for generating descriptions for object-oriented components, which are the main representative of reusable components.

5 Multilevel Generalized Signature Matching

In the previous section 4.6 we discussed an extensional description approach for component retrieval. Several aspects hampering a successful application of this method were identified. One of them is the rigidity of formal signature matching. Therefore, in the current section we discuss a more general approach of signature matching as a first step towards improved behavior based function description. The main aim of generalized signature matching is to improve recall, as precision is raised in the retrieval steps later on.

The chapter is organized as follows. In section 5.1 on the next page we discuss the notion of types and signatures on a general level. Based on this we introduce a syntax for the representation of signatures, the *signature grammar*, in section 5.2 on page 73. The signature grammar serves as a common ground for representing type information obtained from "real" programs. How to obtain them is briefly discussed there, too. Signatures represented in the grammar are then transformed to an intermediate *signature graph*, which is introduced in section 5.3. Signature graphs are an intuitive representation of type structures and are used for calculating various relations between types, e.g., name equality or subtype relations. In this way, a searcher can build flexible queries. Our aim is to enable retrieval with high recall on the basis of a formal and rigid description of components. If the searcher's focus is on the structural aspects of the components reflected in the signature, then generalized signature matching achieves high precision as well. A further positive aspect of describing components on the basis of signatures is the potential to compute them automatically without the need for human intervention.

5.1 Signatures and types

Signatures describe the interface structure of components implemented as types, functions, or modules. The aim of the description is to characterize a component implementation-independently. Components described in that way can be sought in a repository which is built on descriptions of the same type. However, to successfully match signatures, the technique must provide enough flexibility to express vague requirements too. These are requirements which may not be known in detail, or might not be important for the searcher. Matching flexibility allows to

- distinguish between signatures of data types, functions and modules,
- ignore ordering of parameters in signatures,
- neglect the names of signatures or labels within their structure,
- generalize or specialize information about structures,
- match structural similarities between query signature and candidate signature with varying granularity.

If such degrees of freedom are provided, a searcher can build a query with the detail she/he is able or willing to specify. In the case of missing information the retrieval mechanism nevertheless should collect components with a signature “close” to the query. In this way, recall is improved by means of query relaxation and of query expansion.

Modules play a special role in this context. They are considered as a means to group a (semantically) coherent sets of types or sets of functions which are refered to by a name. Since modules are built upon types and function, we discuss their signature aspects first. The transformation of modules is only a small, further step.

5.1.1 Types

Types evolved in the theory of programming languages to help reasoning about different concepts of data (which all are implemented as bit strings of fixed size in memory). In programming language history, the initial universe of data was unorganized in the way that every concept was represented as bit string, be it a character, a number, a reference, a structure, or a program. But such a uniform representation is inconvenient and unsafe to handle, since operations performed on numbers and characters might differ substantially. The meaning of the data is infered by determining the corresponding external concept. If the concept of a bit string is identified wrongly, the result of an operation could be disastrous. To avoid dangerous ad-hoc inference, techniques were developed to structure the data and to help representing external concepts better. The result of this structuring process is *typed* data. Depending on its purpose resp. programming paradigm, the structuring process is performed differently. However, seen from a historical perspective the direction of each purpose is always the same: Starting from an untyped universe, more and more stronger types are introduced to organize the domain [45]. The term *strong* indicates restrictions which are imposed by constraints on the domain. As a consequence, a type t is stronger than a type s, if the cardinality of its$t$'s set of elements is smaller than the cardinality of s' set of elements.

This top-down view is one possible approach to data types. The other one sets a focus on the constructive approach, where starting from axiomatic base constructs more and more complex types are built. This bottom-up strategy is pursuit in this chapter. However, the top-down and the bottom-up view are two sides of a coin, and both aspects are considered in this chapter.

A possible realization of data typing is the utilization of set theory to subdivide a domain. In set theory everything is seen either as an element, a set of elements, and/or other sets. Mathematicians use sets to organize extremely rich and complex structures. The use of set theory for structuring domains reflects the idea of types as sets of values. In contrast, in the λ-calculus everything, beginning from numbers, data structures, or even bit strings, is considered as a function. In this work we follow the example of Zaremski and Wing [232] who organize data types as sets of functions, thus applying aspects of sets and functions in a bottom-up way. Their ideas are adapted to our needs for generalized signature matching.

Types are built by applying type operators to zero or more arguments, or by referencing to already defined types via type variables. Since we do not let users define their own type operators, we encounter a *first order type system* [45]. As an initial step, type operators are defined to be used as tools for constructing types. Depending on the arity of their operators, *base types* and *complex types* may be distinguished.

Base types are built from operators which do not demand operand lists. There does not exist a predefined base type operator in order to define base types like e.g. `int` or `char`. To define base types we take the name of the type itself for a 0-ary operator. In our work the semantics of base types is not considered further; the only one exception is the reasoning about subtype relations (see section 5.4.1

for details). There, to enable subtype checking, for each base type a-priori subtype relation must be defined in advance.

Definition 5.1. – *Base Type Operator* τ_b
A *base type operator* τ_b is a unique name.

With base types as building blocks complex types are synthesized with the aid of *complex type operators*. Complex type operators are built by applying an n-ary operator τ_c, with $n \geq 0$, where the type operator variable τ_c is instantiated with an element from the set $\{\times, \cup, \rightarrow\}$. A complex type operator demands one or two operand lists.

Definition 5.2. – *Complex Type Operator* τ_c
A *complex type operator* τ_c is element of $\{\times, \cup, \rightarrow\}$, where

- the length l of the operand list of the operators $\times$ is greater zero, $l \geq 1$,
- the length l of the operand list of the operators $\cup$ is greater one, $l \geq 2$,
- the operator $\rightarrow$ demands **two** operand lists of length $l \geq 0$.

The *cross product operator* $\times$ is defined with an operand list containing at least one parameter. In this work, in contrast to the definitions of Zaremski and Wing [232], we prohibit unlabeled elements in the operand list. Therefore, cross products are not equivalent to a Cartesian product in set theory, since the elements' labels restrict the set to a subset of the Cartesian product (leading to mathematical relations). The $\times$-operator as defined here is equivalent to the *record constructor* (or *labeled* Cartesian product) of Cardelli and Wegner [45, p 487]. Cross product operators are an integral part of most modern programming languages, e.g. the `RECORD` constructor in Modula-2, Modula-3 or Pascal, or the `struct` definition in C.

A *type union operator* $\cup$ (or *variant record* type operator [44]) is known in C and Modula-2 as well. It is an n-ary disjoint union of types which represent alternative structures at run-time. To prohibit empty alternatives, a union type operator must have at least two alternatives (two operands). The structure of a variant record depends on the evaluation of an associated condition. Since we focus on structural aspects only, this semantics in not discussed further in this work.

The *function space operator* $\rightarrow$ is meant to construct functional coherent entities, similar to the `PROCEDURE`- or `FUNCTION`-constructors in Modula. It demands two operand lists; one for the input specification, one for the output specification. Only this operator is allowed to have empty lists. These operand lists are considered internally as *unnamed cross products*. Consequently, cross products derivated from input lists are always defined implicitly. To keep the information about the position of individual elements in the parameter lists, each element of the operand list gets an index. Obviously, this transition strategy may lead to cross products with only one operand, when the input (output) list contains that number of parameters. If an empty parameter list is declared, which is not allowed for cross products, a reference to the base type `void` is introduced.

If a function returns an *anonymous value* as result, we label this value with the meta symbol `~out` and place it as parameter into the out-list. An example for an empty operand list is the ANSI C function `int rand(void)`, which generates a pseudo-random number without getting an input. For further details on the semantics of type operators, the interested reader is referred to the work of Cardelli and Wegner [45].

With base and complex type operators we now have all elements available to define formally the concept of a type.

> **Definition 5.3.** – *Type* $\mathbb{T}$
> A type $\mathbb{T}$ is either a type variable or a type operation. A type operation is the application of a type operator τ to zero, one or two operand lists. A *base* type is built by applying a base type operator τ_b. A *complex* type is built by applying a complex type operator τ_c.

In the remainder of this work, we do not distinguish between the operators τ_b and τ_c, unless it is necessary to do so. A type name is a string to which type refers to a type; formally $name := \mathbb{T}$. Such a type which has a name is called a *named type*.

Now the operands of type operators must be defined. Operands are packed into sequences. For a sequence, to each element a positive whole number is assigned which serves as the element's index. According to Diller [61, page 97] a sequence X is a finite function from the non-negative whole numbers to an element set X

whose domain consists of the numbers between 1 and the number of elements, $seqX \equiv \{f : \mathbb{N} \rightarrow X | dom f = 1..\#f\}$. In this way, the ordering of the elements in the sequence is maintained and as a consequence, elements from X may occur repeatedly. However, we do not allow more than one identical operand label on the same level in an operand list.

A operands is a pair consisting of a label and a type. It might be considered as a locally defined variable within the scope of its operand list. Labels serve as selectors for accessing the operands information. A scope defines the area of accessibility to labels which are accessible at the level of a operand list or at the level of sublists contained in its list. A label's scope is overridden, if an identical label is assigned to a declaration in a sublist. The following definition formalizes the previous paragraphs.

Definition 5.4. – *Operand list of type operator*
An operand list α of a type operator τ_c is a list $(\alpha_1, \alpha_2, \ldots \alpha_k)$. Each operand α_i $(1 \leq i \leq k)$ is an element of the form ${}^{l_i}/_{\mathbb{T}_i}$, where l_i is called the *label* of type $\mathbb{T}_i$. The list α may be empty.

There are two ways to declare types of operands. They might be declared explicitly or implicitly. Every base and named complex type is a explicit type. Implicit types occur within operand lists only; they model conceptual substructures in the operand list. We call them *unnamed types* as well.

A type is defined *recursively*, if it is referenced by name from within its operand lists (direct recursion) or from a type which is used in its operand lists (indirect recursion). Only named types might be defined recursively.

The following examples of types should clarify the points raised so far. We present the types in pseudo-code, but it is not difficult to establish a mapping from existing imperative programming languages to signature definitions.

Example 5.1. *Various Types* $\bigtriangledown$

base: $real$

The type "$real$" is built from an operator without operand list. Please note, that only syntax is defined but no semantics.

complex: $\times(\,{}^{x}\!/_{real},\,{}^{y}\!/_{real})$

This type is constructed by applying the cross product operator, obtaining a record structure. The operand list contains two operands, x and y, both of type real. This type is well defined, but it cannot be accessed directly, due to a missing name for referencing it.

named: $Point := \times(\,{}^{x}\!/_{real},\,{}^{y}\!/_{real})$

The type named "$Point$" is a record structure which is explicitly named and therefore accessible within its defined scope.

unnamed: $Person := \times(\,{}^{name}\!/_{string},\,{}^{x}\!/_{\underbrace{\times(\,{}^{street}\!/_{string},\,{}^{city}\!/_{string})}_{\text{implicit type}}})$

The second operand of the cross product operator of "$Person$" is an unnamed structure representing the concept of an address. This substructure is implicitly defined.

anonymous return value: Consider a Pascal-like function

```
FUNCTION HShift (source:Point, dist:real) : Point;
```

(horizontal shift) for computing a new horizontal position of a point. The type of this function is:

$HShift :=\rightarrow (\,{}^{source}\!/_{Point},\,{}^{distance}\!/_{real})(\,{}^{\sim out}\!/_{Point})$

The complex type "$Point$" together with a distance value for relocating this point horizontally are provided for the input to the function "$HShift$". The result of the computation returned to the caller is labeled with the the meta-designator `~out`. We do not use the function's name to label the output variable (as it is the case e.g. in Pascal), since, in contrast to the functional programming paradigm, we consider the output of the function as a different concept than the function itself.

△

The type structures presented in this chapter are an abstraction from concrete types implemented in a programming language. To distinguish implementation dependent types from their abstraction we use the notion "*signature*". In the remainder of this work, we use the term "type" to refer to the elements on the implementation level and "signature" to refer to elements on the abstraction level.

5.1.2 Modules

Base types are first class citizens of programming languages and, therefore, accessible in every compilation unit. If a programmer implements user defined types within a compilation unit, they are accessible within their scope and there need not be an access path to them. However, if types should be accessible for clients (which are other programs or components), the concept of a signature as an interface descriptions plays an important role. Such separate units are called *modules*. Modules group types (data types and function types) and provide a client with a name to access them. This view is in accordance with the approach of Zaremski and Wing who consider a module as a coherent combination of types [232]. Examples of modules are interface descriptions of objects in `Java`, interfaces of `MODULA-3` modules, or packages and abstract data types in `Ada`.

> **Definition 5.5.** – *Type of a module*
> The type $\mathbb{M}$ of a module is a named set of the types declared in the module.

A module's signature is a named set of the signatures of types declared in its interface. Through this interface the access to its elements is enabled. With the help of modules the analysis of identically named types declared in different modules is possible. If a signature declared in a module must be accessed, it is necessary to specify the access path to it. We chose a common representation for path navigation, namely the separation of path elements by a dot. In that way, it is specified how to access elements of of a module's signature (e.g. `module.type`).

In continuing the example 5.1 from page 70 the data type `Point` used for storing two-dimensional points together with the relocation function `HShift` might be separated from the main program and declared in a separate module `TDPoint`, which is specified by the following interface:

```
INTERFACE TDPoint =
  Point = RECORD
    x : real;
    y : real;
  END Point;
```

```
  FUNCTION HShift (source : Point,
                   distance : real) : real;
  ...
END TDPoint.
```

5.2 Signature definition

5.2.1 Signature grammar

To treat different programming languages we need an intermediate representation of signatures. The signature grammar shown in figure 5.1 on the next page is a type expression sublanguage for a first order type system, similar to the type construct operators of imperative programming languages[1]. The issues discussed in the previous sections are reflected in its structure.

The transformation of type declarations expressed in terms of an imperative programming language into implementation independent signature definitions conforming with the signature grammar is straight forward. However, the transformation rules must be defined for each programming language individually. In the following section we give some hints for this transformation process for those spots, where difficulties might occur.

---

[1] The grammar is expressed in *E*xtended *B*ackus *N*aur *F*orm (EBNF), which is a meta language for defining formal languages. EBNF's building elements are rules and three different types of symbols. Rules are built like an equation. On the left hand side the name of the rule is given, followed by the assignment symbol "`::=`". This name is a *non-terminal symbol*. The body of a rule is on the right hand side of the assignment. The symbols are grouped into

1 meta symbols, which are used to build the structure of the target language.
The parenthesis symbols "`(`" and "`)`" group definitions. The meta symbol "|" expresses alternatives. Definition left or right to it are equivalent alternatives. The bracket symbols "`[`", and "`]`" define a zero-or-one alternative declaration.Braces ("{", "}") denote iterations, which may never appear or which may appear with an arbitrary frequency.

2 Terminal symbols are given "as is" in the form of e.g. `'a'`.

3 Non-terminal symbols are names of rules and appear at the left hand side of an assignment. They may occur in the body of a rule as well for referencing rules to build complex structures.

EBNF in the presented form is redundant due to unnecessary meta symbols, because it is possible to express any definition recursively by using alternatives ("|") only. However, a richer set of constructing elements make a much shorter and a much more readable declaration possible.

```
Declaration     ::= Module | Types.
Module          ::= 'MOD' SigName '=' Types.
Types           ::= TypeDec {';' TypeDec}'.'.
TypDec          ::= TypeName ['=' TypeBody].
TypeBody        ::= FunOp | CrossOp | UnionOp.
FunOp           ::= '->' '(' TypeOperandList ')'
                    '(' TypeOperandList ')'.
CrossOp         ::= 'X' '(' TypeOperandList ')'.
UnionOp         ::= 'U' '(' TypeOperandList ')'.
TypeOperandList ::= TypeOperand {',' TypeOperand}.
TypeOperand     ::= Label '/' (TypeName | TypeBody).
TypeName        ::= [ModuleName'.']Label.
ModuleName      ::= Label.
SigName         ::= Label.
Label           ::= char{char|digit}.
digit           ::= '0'|'1'|...|'9'.
char            ::= 'a'|'b'|...|'z'|'A'|'B'|..|'Z'.
```

Figure 5.1: A signature expression sub-language

5.2.2 Hints for transforming types to signatures

References

References to a type within the scope of a declaration are established by providing the name of the referenced data type. Since we only want to maintain its structure, a recursively defined type such as a linked list, is transformed by replacing the pointer to the structure by the name of the refered type. We are well aware that this may lead to an infinite recursive definition. This problem will be taken care of in the matching process later on. See example 5.2 on the following page for a transformation of the linked list `PList`.

Arrays

Many programming languages provide with arrays which are intended to group elements. In that way, arrays are similar to record types, but, in contrast to them, the elements have to be homogeneous (they have to be of the same type). This may not be true for records. A further particularity of arrays is the index associated with each element. This index permits the addressing of each element directly.

The signature expression sublanguage presented in 5.1 on the preceding page does not provide with a declaration for arrays. This motivated, that arrays may be expressed by other signature structures which will be sufficient for matching purposes. With this in mind, arrays may be considered as implicitly linked lists. Then, the information about the length of the original array and, furthermore, the position of its elements gets lost. However, for matching purposes position information might be important and, therefore, we transform *array types to records*, where each element of an array is represented as a field of a record. In that way, the focus is set on the data (and not the meta structure needed to administer it). In order to maintain the index information at least partially, after the transformation, the labels of record fields contain index information. For this purpose the meta symbol `~i` is concatenated with the index. As previously mentioned, index information is maintained partially only. This is due to the fact, that the information about multi-dimensionality is lost when arrays are transformed to records. Dimensions are flattened by sequentially enumerating array elements. This effect is shown in example 5.2 on the following page, where the two-dimensionality of the array of points `PArr` cannot be inferred from its signature pendant.

Functions

For functional entities the information about different passing modes of parameters is important. In this work we differentiate only coarsely between IN, OUT and INOUT modes, without considering particularities like call-by-value, call-by-name, copy-restore, or call-by-reference passing modes [2]. They are not of primary interest when it comes to signature matching.

IN-parameters indicate a value which is passed to a functional entity and it occurs on the left hand side (in the first operand list) of a function's type. The indicator for values passed to the caller of a function are OUT-parameters, which are specified on the right hand side (in second operand list). If values are passed in INOUT mode, the names of parameters are placed on the left and the right hand side as well. Functions declared with empty parameter lists result in empty in- and/or out-lists (which are administered as records containing a void-type).

Example 5.2. *Transformation of declarations to signatures* ▽
Consider the following definitions of types for storing two-dimensional points, defined in a Pascal-like pseudo code. The record type `Point` holds the coordinates of a two dimensional point. The linked list `PList` administers a set of points. `PArr`is implemented as a two-dimensional array which elements are points. Furthermore, two functional entities are declared: the procedure `HMove` intended for shifting a given point horizontally `l` units along the x-axis and the function `Dist` computing the distance of the position of the entered point from the origin of the coordinate system.

```
INTERFACE TDPoint;

  Point = RECORD
    x : real;
    y : real;
  END Point;

  PList = RECORD
    p : Point;
    next : POINTER TO PList;
  END PList;
```

```
  PArr = ARRAY [0..1] [1.2] OF Point;

  PROC HMove  (IN l:real, INOUT p:Point);

  FUN Dist (IN p:Point) : real;

END TDPoint.
```

These type definitions are transformed according to the definitions of the signature grammar (figure 5.1 on page 74) with respect to the transformation hints discussed in this section.

```
Module TDPoint =
  real;
  Point = X (x/real, y/real);
  PList = X (p/Point, next/PList);
  PArr  = X (~i1/Point, ~i2/Point,
             ~i3/Point, ~i4/Point);
  HMove = -> (l/real, p/Point) (p/Point);
  Dist = -> (p/Point) (~out/real).
```

△

Although the base type `real` is not declared in the implementation, it must be present on the signature level. We give priority to an explicit definition of base types than having them as first-class citizens on the signature level, because this provides with the flexibility to deal with a multitude of implementation languages. A base type on the signature level is defined by its name only; no type operator is involved. This style is a shortcut definition indeed, since the name of the operator is the name of the type itself as well.

The recursive declaration of the list of points `PList` established by the pointer `next`, has to be converted into a recursive signature declaration. The information that a pointer reference was originally the declared is not maintained. This is not consider a loss of important information, because for matching purposes the concept beyond the technical implementation is much more interesting.

The `INOUT` parameter `p` in the parameter list of function `HMove` occurs in both operator lists of its signature equivalent. Finally, the parameter `~out` placed in the second parameter list of the signature of function `Dist` is the result of the transformation of an anonymous return value.

5.3 Signature graphs

5.3.1 Basics

To compare efficiently signatures and as basis for establishing signature relations we map the signature information of types into a graph. A signature graph $G = (E, V)$ is a directed graph with a set of directed edges E and a set of vertices V. We refer to these sets as $E(G)$ and $V(G)$ as well. An element $v \in V$ is either a type operator (including base types) or an unique root node. Root nodes are "entry points" of types within a certain scope, from where each named type is immediately accessible. Either the signatures are accessible via the scope of a module or via the scope of a main program. In the first case, the access vertice is labeled with the name of the module, in the second one, the label is named "`root`".

Each named signature is connected directly to the access node. All other signatures are unnamed and implicitly declared. However, to simplify the graph representation, base types, although named, are not directly connected to the access node. They are shown in the graph as double circles.

An edge $e \in E$ is a pair of vertices (v_i, v_j) where the left vertice v_i is the source and the right vertice v_j the target of e. The set of directed edges E represents the declaration relations for each signature. Edges may have properties which hold information about labels and the position information of operands of cross product operators.

Function operator transformation

As shortly mentioned in section 5.1.1, the input and output lists of function types are transformed to cross products which are represented adequately in the graph. The cross product serving as input structure is designated with the meta-symbol `~I`, the output structure with `~O`, respectively. In the case of an empty input- and/or output list a graph representation would lead to an empty cross product structure, which is not allowed according to definition 5.2 on page 68. In the graph such a list leads to a unary cross product, containing the only operand which has the type `void`.

Example 5.3. *Signature graph of example 5.2 on page 76* $\triangledown$
This example is a continuation of the transformation example 5.2 on page 76.

In figure 5.2 the signature graph of the structures of the module `TDPoint` is given. The root node is depicted as a box and named with the module's name "`TDPoint`".

The root is linked to the cross product `Point` which contains two elements, `x` and `y`. Both elements are defined as base type `real`. The numbers in square brackets indicate the position of the record components in the original operand list. The recursive definition via the pointer `next` referring back to `PList` is shown as reflexive edge `next [2]`.

The input and output lists of the function `Dist` and of the procedure `HMove` are cross products, labeled with the meta-symbols `~I` designating the input, resp. `~O` specifying the output. `Dist` takes only one input, which is a variable `p` of type `Point`. The return value of the function is a cross product (with the label `~O` as selector) having only one operand, which is labeled with the meta designator `~out`.

The array `PArr` is transformed to a record structure and potentially stores the same amount of data as the original does. However, the information about the two dimensions initially specified on the implementation level is not present in the graph, although all homogeneous and sequentially indexed elements of type `Point` are contained in the signature graph.

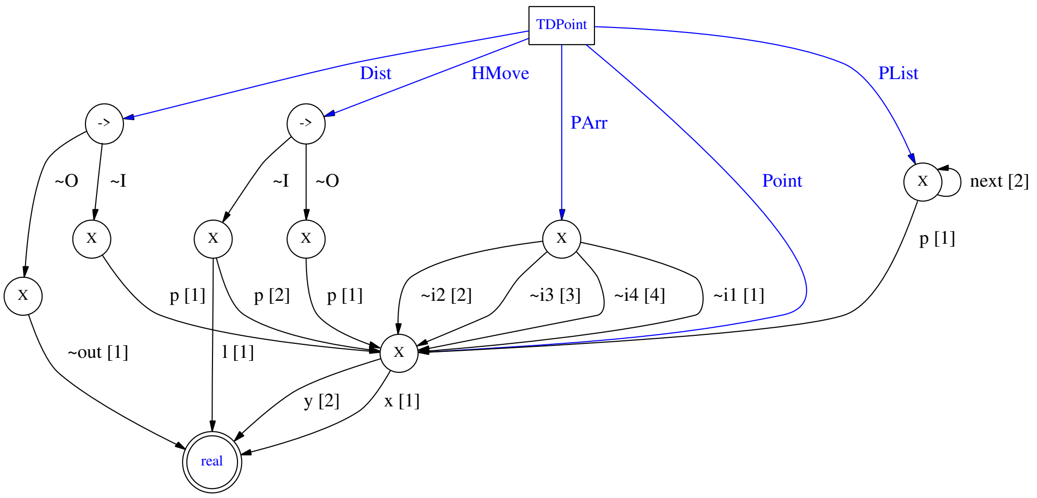


Figure 5.2: The complete signature graph of module `TDPoint`

△

Union operator transformation

A special case is a type p constructed by the union (or variant) operator ($\cup$). Each element in the operand list is a possible implementation alternative for p, but for a specific variable of type p only one alternative is valid at a certain point in time. However, for the purpose of determining type relationships (which is a static analysis) all possible variants must be considered. This makes sense for matching purposes as well, since a searcher looks for one possible instantiation of her/his demand. A union type is meant to change its shape dynamically and to fit the demand expressed in the query. After the query's signature is matched successfully with one of the alternatives, any other alternative is not interesting anymore (only the union specificity should be indicated). Hence, all variants of a $\cup$-type have to be analyzed and are transformed to the graph. In the signature graph a union is denoted by an operator node '$\cup$' with all its possible variant types as successor nodes. In contrast to the process of the transformation of cross product types, the initial ordering of the operands need not be maintained, because all alternatives are equal choices. The initial order does not provide any additional information. The specific semantics of a variant type must be considered in all analysis and matching steps. The transformation of an union signature to its counterpart represented as graph is demonstrated in example 5.4:

Example 5.4. *Representing a union type in the signature graph* ▽
Let p be a union type with the following signature $p = \cup({}^{a}/_{int},\ {}^{b}/_{real})$. In the graph the signature p is transformed to an operator node 'U' with its two alternatives as immediate successor nodes (figure 5.3 on the facing page). If e.g. a subtype relation to another signature s is calculated, both variants 'a' and 'b' have to be checked against the structure of s for validating this property. If indeed one out of these alternatives has the subtype property, the union itself then is a subtype of s. A detailed example for subtype matching is presented on page 93.

△

Similar graphs called *tgraph*s, are discussed by Katzenelson et al. [115]. In their work, the authors construct an ordered directed graph from type definitions to support type matching for different modules at link time. Vertices of a *tgraph* contain name information, type operands, and record labels. Edges are not labeled and all information about labels, type names and type operators must be derived from the vertice's designator. Additionally, the edges leaving a vertice are ordered.

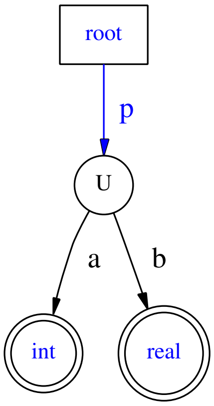


Figure 5.3: Transformation of an union type

The access point r to a *tgraph* is determined by the user, who specifies the type to be analyzed. All types induced by r then are added to the graph. The intention of the authors is to compute a polynomial representation for efficiently matching types on a hashing schema thus obtained. This schema is based on irrational numbers (which are called *magic number*s) representing each type graph uniquely.

The matching of *tgraphs* based on their magic number equivalent is claimed to be very efficient, as it is based on comparison operation in the hashing scheme representing the *tgraph*. The difference to the approach presented in this work is due to the fact that we establish different signature relations and additionally allow matching relaxations by graph transformations (discussed in the following sections). In contrast to Katzenelson's approach we may match parts of types as well (and allow different matching criteria), since we do not pack their signature information into one indivisible representation.

5.3.2 Signature graphs spanning different modules

Modules define the scope for the types implemented in it. On the signature level, each module as well as the main program serve as an entry point for accessing their constructs, a fact which is reflected in signature graphs. If an element is defined which type is imported from an other module, we have to keep the information, how to incorporate this reference into a signature graph.

We do so by referring to the signature graph of the module via the external access point by merging the two graphs in such a way that a graph has more than one entry points. The idea is depicted in the following example.

Example 5.5. *Accessing externally defined types* ▽
The declaration of the record `Location` is placed in the main program. It is intended to contain data of a two dimensional map. One data element of the record is the position of the location `position`, administered by the structure `Point` which is imported from the module `TDPoint` declared in example 5.2 on page 76.

```
PROGRAM Map
  ...
  IMPORT TDPoint;
    ...
    Location =
      RECORD
         position : TDPoint.Point;
         name     : string;
      END Location;
    ...
END Map.
```

The resulting signature graph (figure 5.4 on the facing page) contains the structures declared in the main program `Map`, which are accessible via the `root` node of the graph. The signatures infered from the types of the module `TDPoint` in example 5.2 are merged into the graph. We do not show the intermediate step of translating the implementation dependent declarations of the program `Map` to signatures. The access node to `TDPoint`'s types is shown in the graph. It enables the correct derivation of their naming scheme (access path).

△

5.4 Signature relations

5.4.1 Strata of signature relations

In this section we discuss relations which can be established between signatures. On the basis of such signature relations various "distances" can be formulated. We use signature graphs as basis for comparing structural and designative properties. In general, two different aspects are important for the comparison:

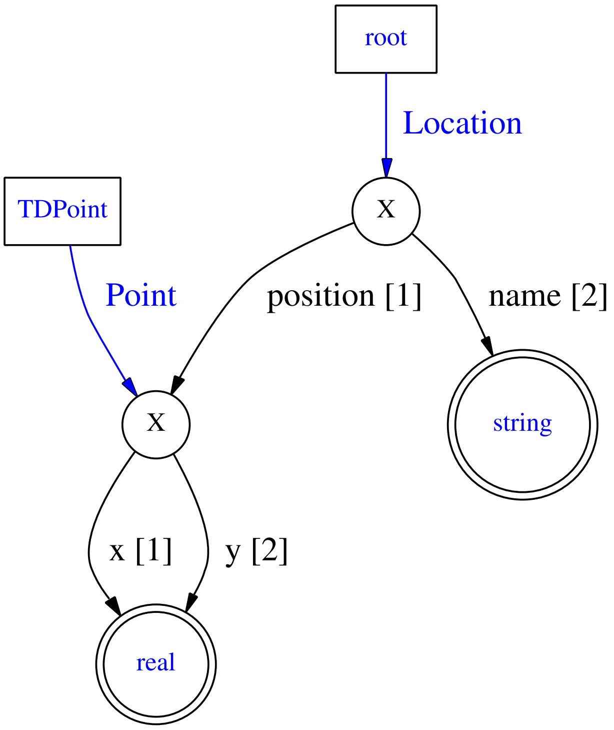


Figure 5.4: Merging signature graphs of different modules

1 The first aspect is concerned with the designation of signatures and their substructures. Many programming languages base their type system on the uniqueness of names. Although this eases the task of checking for compatibility, in many cases this approach is too restrictive.

2 The second aspect takes the internal structure of types into account.

Both aspects are important and depending on a searcher's requirements, they must be respected. Some of the signature relations here presented are based on rather pragmatic ideas, a pragmatic which stems from the aim to support component reuse best. Application engineers often search for domain related aspects, without considering the technical and theoretical background in detail at that moment. Therefore, subtype matching (based on a sound theory) often leads to unexpected results which are considered to be counterintuitive to a "naive" searcher. In this work we support a more holistic view to signature matching, respecting sound theory and heuristic structural matching as well.

All algorithms presented in the subsequent sections are implemented in SETL2 and are listed in appendix B. SETL2 is a high level general purpose programming language based on the theory of finite sets [212]. Due to this set-orientation it is very convenient to prototype the ideas developed in this thesis without having to deal with too much implementation details. SETL2 offers all primitives for developing understandable and modular programs, including existential and universal quantifiers, modules and interfaces. Its syntax is similar to the one of Ada.

Let s and p be two signatures, with $s = \tau_s \alpha_s$, where τ_s is the type operator, and α_s is the operand list $({}^{s_1}/_{\mathbb{T}_{s_1}}, {}^{s_2}/_{\mathbb{T}_{s_2}}, \dots , {}^{s_m}/_{\mathbb{T}_{s_m}})$ of s. The signature $p = \tau_p \alpha_p$ is built from a type operator τ_p and a operand list $\alpha_p = ({}^{p_1}/_{\mathbb{T}_{p_1}}, {}^{p_2}/_{\mathbb{T}_{p_2}}, \dots , {}^{p_n}/_{\mathbb{T}_{p_n}})$. We use s and p for demonstrating type relations.

Name equality relation

The first relation, name equality, is concerned with the names of types and it is the simplest one to check. E.g. name equality is used in the `MODULA-2` compiler to check for type equality. Two types are name equal, if both have the identical string as designator. In the signature graph, we can determine name equality between two types s and p, if they are directly reachable from an access node in the graph and the edges from the node to the type operator are labeled equally.

> **Definition 5.6.** – *Name Equality* $(=_n)$
> A type s is *name equal* to a type p (symbolically $s =_n p$), if s and p are identified by the same string .

We can see immediately that the name equality relation is reflexive and symmetric. It is transitive as well, which results in an equivalence relation.

Since the basis for checking name equality is the signature graph, the check for it is to compare labels of edges leaving an access node. Hence, only named signatures can be in a name equivalence relation, as unnamed ones are not directly reachable from an access node.

Obviously, edges from the same access node (defining the scope of a module or a main program) can never have the same identifier as name: It is not possible to declare different types having the same name within one module or program. On the other hand, signatures defined in different modules may have indeed the same name. Name equality plays an important role during matching of signatures, because module names (as primary designators) are not considered on the highest level in a matching process.

Example 5.6. *Name Equality* $\bigtriangledown$
In a module A a signature $node := \times({}^{k}/_{int}, {}^{v}/_{char})$ is given. This signature is name equal to $node :=\rightarrow ({}^{x}/_{real})({}^{y}/_{int})$ declared in a module B, which is symbolically expressed as $node =_n node$. $\bigtriangleup$

Subtype relation

In general, a type s is a subtype of a type p, if and only if all elements (values) which are in s are in p, too. The main motivation for subtyping is to determine, if a type s can be substituted by p, without changing invariants which hold for a type s. This process is called substitution or subsumption [83]. Due to the property of substitutability, when designing an algorithm for signature matching which should allow inexact matches as well, the sound theory of subtyping is a primary candidate to be considered for matching.

In the literature, subtypes are defined differently and, consequently, the inference of subtype relationship is discussed differently as well. As already stated in section 5.1.1 this is attributed to they way how to look at data types. When discussing subtyping, we must be well aware of the fact that two orthogonal dimensions must be considered. The first dimension is oriented towards the top-down view, where the domain is considered to be a set which must be partitionized according to constraints with the aim to share properties between types. This view is discussed in the literature as subclassing (see also the discussion on covariance on page 89). The second dimension pursues the idea of a constructive, bottom-up approach, where types are built up from simple building blocks to get more and more complex constructs. Here subtyping deals mainly with the question of substitutability, i.e. to determine, whether or not a supertype variable may be safely substituted by a subtype variable. This inference problem is discussed in this work as contravariance subtyping starting with page 89, too.

The following parts of this sections reflect this discussion. We start with the simplest form of structuring types by using set theory, where the subtype relation is defined on the subset property of its homogeneous data elements. The next section extends these ideas, but considers inhomogeneous data elements, record structures, as well. The most complex form of subtyping is given, when defining data types as functions. This is the topic of the last part of this section.

Types as set of values

The idea of subtyping evolved from set theory as basis for organizing the universe of data. An element is said to *have a type*, if it is a member of the set. Since an element may be member of more than one set, it can have more than one types either. The universe of data can be structured according the subset relation which hold within this universe. Such various relations are modeled as a lattice. The top element of the type lattice is the type $\top$ which is the set containing all data of the modeled universe. The bottom $\perp$ of the lattice contains the element of the smallest type. In most cases $\perp$ complies with the empty set [45, p 490].

If one sees types as a mean to organize otherwise homomorphic data values, set theory is an appropriate mean to structure such values. We then can refine the statement about subtypes given above: A type s is a subtype of type p iff s is a subset of p ($s \subseteq p$) in the type lattice [44, 45]. This provides us with a simple way to determine subtypes for subrange types where the internal structure of the elements does not play a role.

Definition 5.7. – *Subrange subtype* ($<:$)
If s and p are types, s is a subrange subtype of p, symbolically $s <: p$, if $s \subseteq p$.

Subrange subtypes are defined on the properties of base types. A statement about subrange subtyping is valid only, if we can compare the values of types. In our signature grammar, the extensional semantics of base types cannot be considered. Consequently, in the signature graph we cannot find subrange subtypes in the graph's structure directly. Subrange subtype information, such as $int <: real$ (every element of int is member of the set $real$), is domain knowledge and must be available a priori. For the purpose of the formalism developed here, this knowledge is maintained by the set ssb:

Definition 5.8. – *Subset specification for base types*
A subset specification for base types ssb is a set of pairs (t_i, t_j), where t_i and t_j are base types and every element of t_i is an element of t_j as well.

Example 5.7. *Subrange subtype* $\bigtriangledown$
Given are the types $s = \{1, \ldots, 10\}$ and $p = \{0, \ldots, 100\}$. Then, since every element of s is an element of the set p, the pair (s, p) is entered into the set ssb. Now $(s, p) \in ssb \;\Rightarrow\; s \subseteq p$ holds, and therefore, the subrange subtype relation $s <: p$ holds, too. $\bigtriangleup$

The following algorithm `isAprioriSubT` checks, if two types s and b (which have to be base types) are subrange-subtype-related $s <: p$, an information which is provided globally via the set ssb. It returns 'true', if a pair is found in ssb which specifies this relation.

Algorithm 5.1.

```
FUNCTION isAprioriSubT(↓ s, ↓ p) :  BOOLEAN;
BEGIN
  RETURN ∃(t1, t2) ∈ ssb | t1 = s ∧ t2 = p;
END isAprioriSubT;
```

Types as structures

Record signatures are formed by the $\times$-operator and combine labeled fields. It is easy to establish a "natural" subtyping relation between records, similar to the idea of object oriented specialization: Given a record p, a subtype s of p can be constructed by adding fields to s or weakening (subtyping) the existing component types of s [52, 44], e.g. a supertype's field is a $real$ and the subtype's field is int. Since labels are selectors, record fields are uniquely identified by their labels. Thus, subtypes contain at least the same fields and labels as their supertypes, but may have more. Refinements of subtypes are allowed, in the way that operands of the subtype record have a (sub)type of its associated supertype operand (although both must have equal labels). We do *not* assume any fixed order in the operand list (nevertheless, this information is maintained in the signature graph). The association between super- and subtype operands is established via the operands' labels.

In that way, a record structure might be subtype of many supertypes, namely of all those types which have structural fewer fields. However, this complies ideally with our view onto the subset-superset concept as basis for type theory.

Definition 5.9. – *Structural subtyping*
Let s be a record $s = \times\alpha_s$ with $\alpha_s = ({}^{n_1}/_{\mathbb{T}_{s_1}}, {}^{n_2}/_{\mathbb{T}_{s_2}}, \ldots, {}^{n_k}/_{\mathbb{T}_{s_k}}, \ldots, {}^{n_n}/_{\mathbb{T}_{s_n}})$ and $p = \times\alpha_p$. The operand list α_p has length k. Then s is a structural subtype of p, $s <: p$, if $\forall({}^{m_i}/_{\mathbb{T}_{p_i}})\exists({}^{n_j}/_{\mathbb{T}_{s_j}})|\mathbb{T}_{s_j} <: \mathbb{T}_{p_i} \wedge n_j = m_i$.

On the first sight, it seems "unnatural" that a subtype record may have more fields than its supertype(s). However, to think of a type as restrictions on the set of all elements helps. Each record with n fields *is a* (can be used as) record with $n - k$ fields, leading to k unused fields. This is not possible vice versa! That means that the cardinality of the set of $n - k$ records is larger (or equal) than the set with n records, since every record with n fields is a record with $n - k$ fields, but a $n - k$-field record is not a n-field record (for $k \geq 1$). In that sense, every set with n-field records is a *subset* of a corresponding set of $n - k$-field records. Wherever a supertype variable exists, a subtype variable may exist too [44] and the data stored in the supertype can be stored in the subtype structure as well, whereas the data stored in a subtype cannot be stored in a supertype variable in all cases.

The algorithm 5.2 determines, whether the type s is a structural subtype of p or not. If s is a subtype, `isCrossSubT` returns true and false otherwise. For each element op_p in the operand list of the (assumed) supertype p, there is exactly one appropriate equally named element op_s in the operand list of s, which is a subtype of op_s. The auxiliary function *label* returns the label of an operand.

Algorithm 5.2.

```
FUNCTION isCrossSubT(↓ ×opl_s, ↓ ×opl_p) :  BOOLEAN;
BEGIN
  RETURN
    ∀ op_p ∈ opl_p ∃_1 op_s ∈ opl_s | label(op_s) = label(op_p) ∧ op_s <: op_p;
END isCrossSubT;
```

Types as functions

In the case of the function space operator $\rightarrow$, determining subtype relations is more difficult. A subtype can exist anywhere in a program, where its supertype is. This property must hold for functions as well. Thus we pursue the goal of *type safety* in the sense of signature substitutability. In the object orientated paradigm, subtyping may serve another purpose, too. Subtyping can be used as a modeling tool to express specialized behavior of a type. This is done by adding elements to a type which narrows the subtype's domain, an activity called "specialization". If subtyping is used in that way, it may violate substitutability. This problem is well known in the type theory of object orientation, hence, there the distinction between subtyping through inheritance or specialization is pursued fiercely [52]. However, this discussion is not restricted to object orientation, although due to its paradigm of programming by specialization, it is important to make it clear, which kind of subtyping is used. Two different views on function subtyping can be identified in literature [46, 43]:

1 If specialization is the aim of subtyping, we have *covariant function subtyping*. Covariant function subtyping is an intuitive way of modeling, since for the function which is a specialized subtype, only specialized input and specialized output types are allowed. Covariant subtyping is implemented e.g. in the Eiffel programming language [142] or in the O_2 object oriented database system [14]. Formally, the covariance for two signatures s and p is defined as follows:

If $s_i <: p_i$ and $s_o <: p_o$ then $(\rightarrow (s_i)(s_o)) \ <: \ (\rightarrow (p_i)(p_o))$, for short $s <: p$.

The function s is a covariant subtype of p, if the input type of s is a subtype of p's input. The output type of s must be a subtype of p's output as well.

2 *Contravariant function subtyping* on the other hand focuses on substitutability through type safety. To allow substitution, the input of a subtype function must be broader (a supertype) than the input of the supertype function. The output side (second operator list of the function space operator) must be covariantly, the subtype's output is narrower than the supertype's one[2].

The discussion about how to define function subtyping is very vivid in the field of object oriented development systems. In object orientation, function refinement for subclasses is the common way to change functionality when developing

[2]Castagna [46] suggests the term "co-contravariance" for this form of functional subtyping, because the input is a contravariant, whereas the output is a covariant subtype. In practice, this term was never established.

or maintaining a system. In our context of procedural functions, we do not focus on specialization and function refinement for polymorphly substituted variables is not our primary goal[3]. We want to establish subtype relations similar to subrange subtyping between functions. As a consequence, we have to define function subtyping contravariantly:

> **Definition 5.10.** – *Function subtyping (Contravariance rule) ($<:$)*
> Given is a function $s =\!\!\rightarrow \alpha_s^i \alpha_s^o$ and a function $p =\!\!\rightarrow \alpha_p^i \alpha_p^o$. Then $s <: p$, if $\alpha_p^i <: \alpha_s^i$ and $\alpha_s^o <: \alpha_p^o$.

The operand lists α_p^i, α_p^o, α_s^i and α_s^o of the functions which must be compared are $\times$-signatures. Therefore, the parameter lists are checked for structural subtyping relations (defined in 5.9 on page 88). In the remainder of this work contravariant subtyping is meant, when talking about functional subtyping refinement.

Example 5.8. *Function subtyping* ▽
Let $abs =\!\!\rightarrow ({}^{x}/_{int})({}^{\sim out}/_{int})$ be a function for computing the absolute value of an integer and $sin =\!\!\rightarrow ({}^{x}/_{real})({}^{\sim out}/_{real})$ be the standard trigonometric $sine$ function. The base type integer is defined a priori as subtype of real, $int <: real$. Consequently, abs must not be a subtype of sin, as its both lists, input and output, are refined covariantly. Hence, there exists no subtype relation between abs and sin or vice versa. This, at a first glance, seems indeed to be rather unintuitiv.

In contrast, consider the function $floor$ determining the largest integer which is smaller as a given real number, $floor =\!\!\rightarrow ({}^{x}/_{real})({}^{\sim out}/_{int})$. Now the input signatures are analyzed: abs gets an integer as input, $floor$ a real. Since sin gets a real number and we know that $real <: real$, the input type of the function $floor$ is in both cases a supertype. On the output side the situation is reverted: $floor$ computes an int (the same is true for abs), sin a $real$, which establishes a subtype relation for the output type of $floor$ with the output types of both other functions. As result, we see that $floor <: sin$ and $floor <: abs$.

In figure 5.5 on the next page the signature graph of the abs, $sine$ and $floor$ functions is shown. The information about the existing subtype relations for named types is denoted with bold arcs. If s is a subtype of p a directed bold arc from p to s is drawn, named “Subtype”. △

[3] However, type substitutability which is eased by polymorphism is important in the succeeding steps of the reuse process, where the searcher wants to exploit the components with a minimum of effort.

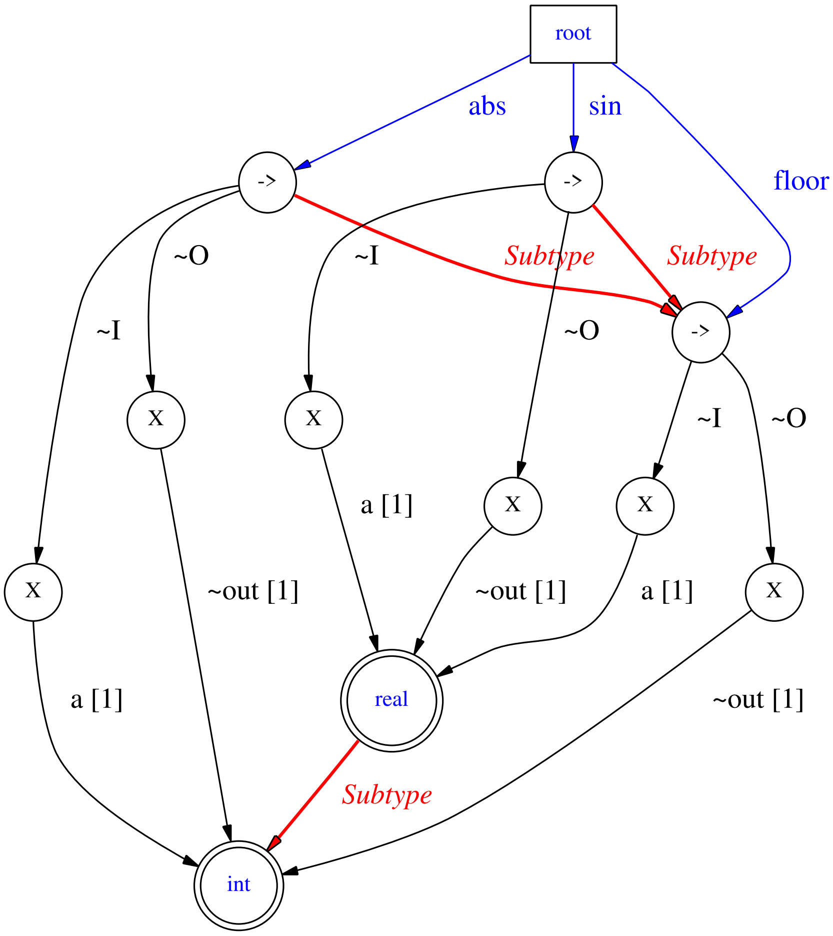


Figure 5.5: Functional subtype relations between abs, sin, $floor$

The algorithm 5.3 implements the definition of function subtyping in a straightforward manner. Input to the algorithm are two function signatures s and p. It returns 'true', if the subtype relation between its parameters s and p holds, symbolically $s <: p$ and 'false' otherwise.

Algorithm 5.3.

```
FUNCTION isFuncSubT(↓→ (input_s)(ouput_s),
                        → (input_p)(output_p))  :  BOOLEAN;
BEGIN
```

```
  RETURN input_p <: input_s ∧ output_s <: output_p
END isFuncSubT;
```

Types as unions

A union signature s is a subtype of a type p if both are constructed with a $\cup$-operator and additionally, if the subtype property holds in the following way: each operand of the supertype p has to be mapped to any operand of the subtype's s operand list such that for this mapping the subtype property is valid as well. Therefore, the idea that wherever a supertype variable exists, a subtype variable may exist as well, is valid for all runtime instances of p, because there always exists a variation of s, that is a subtype of the current runtime shape of union p[4]. The algorithm 5.4 simply checks, if such a pair of operands of s and p exists:

Algorithm 5.4.

```
FUNCTION isUnionSubT(↓∪ opl_s, ↓∪ opl_p) : BOOLEAN;
BEGIN
  RETURN
    ∀ op_p ∈ opl_p ∃ op_s ∈ opl_s | label(op_s) = label(op_p) ∧ op_s <: op_p;
END isUnionSubT;
```

The small difference to algorithm 5.2 (checking structural subtype property) is the fact, that here it is not demanded to obtain a 1:1 mapping to the extent that for each element of the supertype one and only one subtype element must exist. However, this difference is academic, because due to the uniqueness at the same level of the operands' labels (the selector property), in any case only one mapping pair can be established. A union subtyping is depicted in the following example, where two types have subtype property.

[4]The reader might note that this does not imply that at any point in time a substitution of p is possible, since the shape of s may be fixed after the first value assignments. We rather stress the point of potential substitutability, as the structure is meant to be matched in its static form and not during run-time.

Example 5.9. *Union subtyping* $\bigtriangledown$
Given are two union types, $s = \cup({}^{a}/_{int}, {}^{b}/_{real}, {}^{c}/_{int})$ and $p = \cup({}^{a}/_{int}, {}^{b}/_{real})$. Since both of p's operands a and b are supertypes of s's operands a and b, the type s is a subtype of p. In contrast, although p's operands a and b are subtypes of s's operands with the same name as well, p can not be a subtype of s. This is due to the fact, that the operand named c is no supertype of any of p's operands (there is no operand labeled c in p's operand list). $\bigtriangleup$

The complete subtype algorithm

The algorithm 5.5 on the following page computes the general subtype relation between two signatures s and p for any type operator. It is defined as infix predicate demanding two signatures as input and returns a boolean value. The first step of the algorithm is performed to prevent infinite recursion, which may happen when recursive definitions are encountered. Consider the signatures depicted in figure 5.6, where s is a subtype of p and p is a subtype of s. Any data represented in type s may be represented in p (and vice versa). In that case, the algorithm 5.2 would check recursively the operand list of both types endlessly. Therefore, the information about types which are already analyzed is kept, which is performed by the following function:

- `isAlreadyChecked(`s`, `p`)` is a predicate which determines, if its input signatures (s, p) were already analyzed so far. By each call to the subtype algorithm these parameters are added to the set (if the pair was not already member of the set) and 'true' is returned. If the parameters were member of the set before, the predicate returns 'false'.

In this way, the algorithm performs recursive subtype checking similar to the technique of an one-step fix point expansion used to unfold recursive subtypes in static pointer analysis [171].

With the following step name equality is checked. In the case that name equality holds between two signatures, it is known that s must be a subtype of p, $s <: p$. This is due to the fact that within the scope of one signature graph all names are unique[5]. Otherwise, the type operators of s and p are determined, a task performed

[5] Name equivalence is possible for types defined in different modules only. For subtype checking, identically named types declared in different modules are made unique by taking into consideration the access paths to a signature via the root node of the signature graph. This is performed during the type-to-signature transformation process.

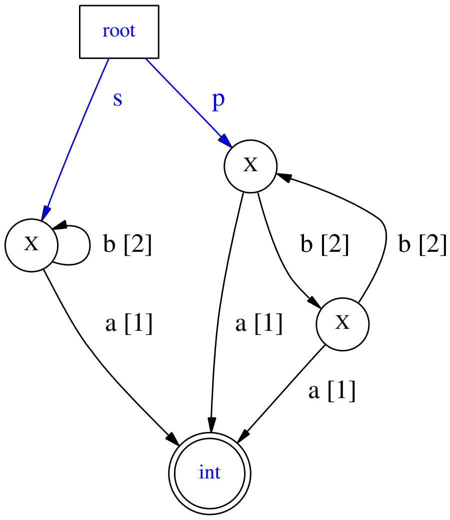


Figure 5.6: Recursive mutual subtypes (from [115])

by the auxiliary function `TypeOpOf`. In the case that s and p are constructed by different type operators a subtype relation do not exist and the algorithm returns `FALSE`. The next step deals with the selection and execution of the appropriate specialized subtype calculation, depending on the type operators of the signatures. The subtype algorithm is applied recursively to the input structures.

Algorithm 5.5.

```
INFIX FUNCTION <:(↓ s, ↓ p) :  BOOLEAN;
  BEGIN
  IF isAlreadyChecked(s, p) THEN
    RETURN TRUE;
  ELSE
    IF s = p THEN
      RETURN TRUE;
    ELSEIF TypeOpOf(s) = TypeOpOf(p) THEN
      IF TypeOpOf(s) = "BASETYPE" THEN
        RETURN isAprioriSubT(s, p);
      ELSEIF TypeOpOF(s) = × THEN
        RETURN isCrossSubT(s,p);
      ELSEIF TypeOpOF(s) = → THEN
```

```
        RETURN isFuncSubT(s,p);
      ELSEIF TypeOpOF(s) = ∪ THEN
        RETURN isUnionSubT(s,p);
      ELSE
        RETURN FALSE;
      END IF;
    ELSE
      RETURN FALSE;
    END IF;
  END IF;
END <:;
```

The subtype relation is reflexive, since any type is subtype of itself, and it is transitive. It is neither symmetric nor antisymmetric (which was demonstrated already in example 5.8 on page 90).

Constitutive equality

Subtype matching is a powerful technique to establish relationships between types. However, it is often the case that the searcher does not look for substitutable subtypes, but has some domain specific "closeness" relationship in mind, an idea which might but must not hold for subtyping. E.g., let s and p be two records, whose structure is shown as signature graph in figure 5.7. Neither s is a subtype of p, nor vice versa, although both are conceptually similar to each other, due to the fact that the number of elements is the same and they refer to the same base types. But the labels and the order of the operand list's elements are not identical. If the reuser wants to find components, which are conceptually near to a query, the names of signatures and the labels of signature operands might not be interesting. In such cases, for the searcher want a component whose structure (regardless of names) fulfill the requirements. Therefore, we introduce a similarity relation called *constitutive equality*.

When checking constitutive equality, names and internal labels of signature are ignored. The structure of the signatures to be compared must be congruent and the base signatures have to be equal as well. Additionally, the order of signature operands in a operand list is neglected. If compared signatures are functions, both input- and output lists must be checked for constitutive equality. For any other signature operator, a bijective mapping from each operand to an operand of the

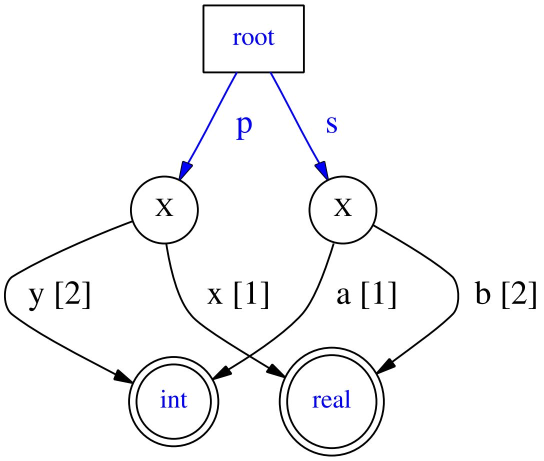


Figure 5.7: Two constitutive equal signatures

compared signature's operand list must exists and the such mapped signatures are constitutive equal, too:

Definition 5.11. – *Constitutive equality* ($=_c$)
A type $s = \tau_s \alpha_s$ is constitutively equal to a type $p = \tau_p \alpha_p$, (symbolically $s =_c p$), if the operator of s is equal to the operator of p, $\tau_s = \tau_p$, and

- τ_s is a base type operator, or
- if $\tau_s = \rightarrow$ then, since α_s is structured as $\alpha_s^i \alpha_s^o$ and α_p as $\alpha_p^i \alpha_p^o$, the property $\alpha_s^i =_c \alpha_p^i$ and $\alpha_s^o =_c \alpha_p^o$ holds, or
- if $\tau_s \in \{\times, \cup\}$ then $\alpha_p = {}^{p_1}/_{\mathbb{T}_{p_1}}, \ldots, {}^{p_j}/_{\mathbb{T}_{p_j}}, \ldots, {}^{p_n}/_{\mathbb{T}_{p_n}}$ and there exists a permutation P of α_s with $P({}^{s_1}/_{\mathbb{T}_{s_1}}, \ldots, {}^{s_j}/_{\mathbb{T}_{s_j}}, \ldots, {}^{n_s}/_{\mathbb{T}_{s_n}})$ such that $\mathbb{T}_{s_j} =_c \mathbb{T}_{p_j}$ for $j = 1, \ldots, n$.

Constitutive equality is an equivalence relation, since all necessary preconditions (reflexivity, symmetry, transitivity) hold.

The implementation (see appendix B.3 for its source) is rather straightforward. The algorithm `areConstitutiveEqual` demands two signatures, s and p as input and returns "true" if they are constitutively equal. If these signatures were already checked so far (determined by the predicate `isAlreadyChecked(s,` p)), it is known, that a cyclic definition occurred and up to that point no constitutive inequality was detected. In such a case the algorithm returns the boolean value "true". For efficiency reasons name equality is checked as well, if the signature graph does not span modules. In that case, identical names are set equal to identical signatures, since names (labels from access nodes to type operators) are unique within the scope of a signature graph.

If the operators of the input signatures are not the same, no constitutive equality can be established. Otherwise, the algorithm reacts depending on the operator type:

- If base types are encountered, since then s and p must be different, they cannot be constitutive equal; the algorithm returns "false".
- When functions are encountered, the input lists and the output list of s and p have to be validated in detail.
- For any other type operator, their operand lists are analyzed: for each element of s's operand list exactly one constitutively equal counterpart in p's operand list must exist.

We use the following auxiliary functions to simplify the design of the algorithm:

- `getOperator(`s`)` computes the operator type of s.
- `getOperandList(`s`)` returns the list of operands of the signature s. One further remark on the semantics of the construct `getOperandList(`s`)("~I")`: The result of the function `getOperandList(`s`)` is a set of pairs, whose first element, the label, is unique. Thus, this set specifies a function. When calling this function with a label as input, the type referenced by this label in the operand list is returned.
- `isBaseType(`s`)` returns "true", if its input s is a base type.

Algorithm 5.6.

```
FUNCTION areConstitutiveEqual(↓s, ↓p);
  BEGIN
    IF isAlreadyChecked(s, p) THEN
      RETURN TRUE;
    ELSE
```

```
      IF s = p THEN
        RETURN TRUE;
      ELSEIF getOperator(s)=getOperator(p) THEN
        IF isBaseType(s) THEN
          RETURN FALSE; --diff. base types are never c.e.
        ELSEIF getOperator(s) = → THEN
          RETURN
            areConstitutiveEqual(
              getOperandList(s)("∼I"),
              getOperandList(p)("∼I"))
            ∧
            areConstitutiveEqual(
              getOperandList(s)("∼O"),
              getOperandList(p)("∼O"))
        ELSE -- check X or U
          IF #getOperandList(s)= #getOperandList(p) THEN
            RETURN
              ∀ts ∈ getOperandList(s)
                ∃ tp ∈ getOperandList(p) |
                  areConstitutiveEqual(ts, tp);
          ELSE -- #getOperandList(s) ≠ #getOperandList(p)
            RETURN FALSE;
        END IF;
      END IF; -- isBaseType(s)
    ELSE
      RETURN FALSE; -- different type operators
    END IF; -- s=p
  END IF;-- (s, p) ∈ isAlreadyChecked
END areConstitutiveEqual;
```

Structural equality

A stronger conceptual relationship between signatures of types holds, when the order of operands is considered as well. However, labels of operands are not regarded yet.[6]. This type relation is called *structural equality* which is an equivalence relation, too.

Definition 5.12. – *Structural equality* ($=_s$)
A type $s = \tau_s \alpha_s$ is structural equal to $p = \tau_p \alpha_p$, if the operator of s is equal to the operator of p, $\tau_s = \tau_p$, and

- τ_s is a base type operator.
- if $\tau_s =\rightarrow$ then the property $\alpha_s^i =_s \alpha_p^i$ and $\alpha_s^o =_s \alpha_p^o$ holds.
- if $\tau_s \in \{\times, \cup\}$ then $\#\alpha_s = \#\alpha_p$ and $\forall\, {}^{s_i}/_{\mathbb{T}_{s_i}} \in \alpha_s \exists\, {}^{p_i}/_{\mathbb{T}_{p_i}} \in \alpha_p \mid \mathbb{T}_{s_i} =_s \mathbb{T}_{p_i}$, with $i = 1 \ldots n$.

We do not show an algorithm for computing structural equality here, as it is nearly identical to algorithm 5.6 on page 97. There is only one exception: the element in the operand list of t_p which is mapped to its counterpart in s is not arbitrarily chosen, since it has to be on the same position in the operand list.

Exact equality

Exact equality is the strongest relationship between two signatures and it holds, when every detail, beginning from type name to type operators, operand lists, and operand labels is identical in both signatures compared:

Definition 5.13. – *Exact equality* ($=_e$)
A type $s = \tau_s \alpha_s$ is exact equal to $p = \tau_s \alpha_s$, if $s =_s p$ holds and $\forall s_i \in \alpha_s\ \exists p_i \in \alpha_p \mid s_i = p_i$.

All signature relations presented in this section may be used as a basis for reasoning about structural or domain related properties of signatures of components. Nevertheless, although signatures and relations holding between them can be derived automatically, the structures themself may contain specific information about design decisions. Such decisions determine the shape of data structures to a great extent. Without knowing the design rationale, a person searching for a structure is not in the position to determine its shape, although the concept sought is known exactly. Hence, we present an approach to weaken structural details which are not mandatory for storing all the data in the next section. The purpose of this task is to raise the level of recall by focusing on the concept.

[6] An other relation might check the equality of labels without considering the order of operands. This need not be introduced, since it is analyzed during subtype relation checking

5.4.2 Flattening signatures to simplify matching

Signature structure vs. signature concept

In general, when searching for a component it is not possible to consider all structural and semantical variations which might be valid manifestations of a concept. For example, if a component dealing with customer records C is sought, many structural variations can be possible realizations. The customers address might be implemented as

- inseparable part of the record: $(C = \times({}^{name}/_{String}, {}^{street}/_{String}, {}^{city}/_{String}, \ldots))$,
- internal substructure: $(C = \times({}^{name}/_{String}, \times({}^{street}/_{String}, {}^{city}/_{String}, \ldots)))$, or
- predefined address signature: $C = \times({}^{name}/_{String}, {}^{adr}/_{Address})$, with $Address = \times({}^{street}/_{String}, {}^{city}/_{String}, \ldots)$.

Many other realizations are imaginable as well. When searching, the concept (and its data) is the important aspect; whereas the meta-structures reflecting the implementation are not of primary interest. Structural complexity, which is not needed to capture the concept, is due to the modularization of underlying concepts. Modularization is best practice in software development to separate concerns, but it introduces additionally implementation variations. These variations may hamper searching substantially. As a consequence, to ease the task of searching, the least common divisor for representing data is its unstructured, “flat” representation.

Therefore, the option to flatten structures to reduce variations is available. Flattening is a “vertical” reduction of the of the signature graph’s complexity by reducing some paths from the access node to the base types. The base idea is to analyze a signature’s operand list and merge compatible operators with the signature’s operator. Thus, flattening is a simplified form of *type uncurrying* [222, 223] known from the field of functional programming language analysis. Any signature relation operation presented in this work can be applied to flattened signatures as well. However, the searcher must be aware, that (structural based) relations established between two unflattened signature, might not hold for their flattened counterparts.

Flattening can be performed only for cross product and type union operators. Obviously, it is not applicable for functions, because the different semantics of its two operand lists would get lost then. If the input list itself is a function, raising it to the level of its “parent” operator would eliminate the evidence of the list’s purpose as input.

Following requirements have to be respected during flattening:

1 Direct recursions must be maintained,
2 indirect recursions must be transformed correctly, and
3 all data stored by the initial structure must be maintained by the transformed structure as well.

Many edges in a signature graph vanish after flattening the graph. To verify the result and to easy the task of explaining the origin of a path, the information of the source of an edge must be kept. We introduce the path symbol “>” which is used to reconstruct the origin of an edge by iteratively concatenating the labels and the path symbol. E.g. if a path exists containing consecutive edges labeled “a”, “b” and “c”, the resulting edge is labeled as “a>b>c”. In that way the genesis of merged edges is stored. The next example depicts the idea.

Example 5.10. *Flattening a simple structure* ▽

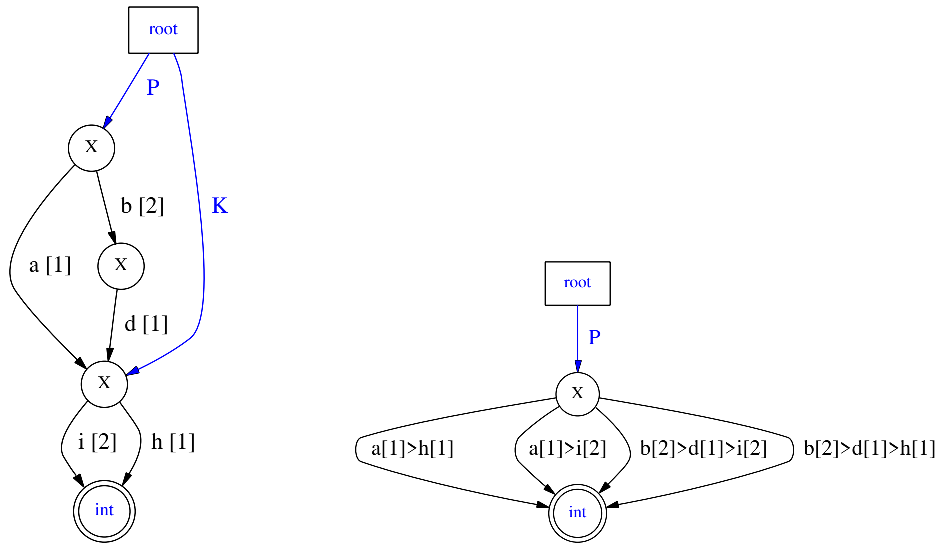
Consider the following signature definitions. The graph representation of the signatures is given in figure 5.8(a) on the next page. The task is to compute a flattened version of signature P.

$$P = \times\left({}^{a}\!/_{K},\ {}^{b}\!/_{\times({}^{d}\!/_{K})} \right)$$

$$K = \times\left({}^{h}\!/_{int},\ {}^{i}\!/_{int} \right)$$

P is built upon two operands, a and b. The operand list of P contains references to K and to an unnamed structure $\times({}^{d}\!/_{K})$. Due to the record structure of both operators, it is allowed to flatten P. In the resulting structure, K's successor vertices and the structure $\times({}^{d}\!/_{K})$'s successor vertices are now direct successors of P. The path from P to the base signature “int” via the edges “a[1]” and “h[1]” must not contain the signature operator of K anymore, which must hold for all paths from P to the base type as well “int”.

The result of the flattening process is given in figure 5.8(b). It contains all paths of the source graph from its root to its base which allows to store the same amount of data. The origin of merged paths is encoded in the labels. In that way, the label “a[1]>h[1]” is the merged representation of the original path “a[1]” to “h[1]”. △



(a) Signature graph of P and K

(b) Flattened P

Figure 5.8: Flattening a record signature

The flattening algorithm in detail

The algorithm 5.7 computes the flattened version of a signature. Input is a signature identifier *curType* and the signature graph g. As usual, the graph is represented as a set of vertices $V(g)$ and a set $E(g)$ holding pairs of vertices $(f,t) \in E(g)$ which represents all edges in g. The processing can be divided into two main steps:

1. Determining the mergeability of the operand list. Mergeability is given, if
   - the current signature is an union or a record,
   - the operator of the current signature and the operators of the operand list's signatures are the identical, and
   - a direct recursion between the current level and the operand list does not exists.
2. If these conditions hold, the current type is linked to all successors of the mergeable signatures in its operand lists. In that way, an intermediate level in the graph is eliminated. The links from the current type to the elements in its operand lists are eliminated.

Also here, auxiliary functions are defined to simplify the presentation of the algorithm. These are the functions

- `getTypeName(`t`)` providing with the string representation of the signature t's name,
- `isReferenced(`v`)` checking if a vertice v is refered to from somewhere in the graph, and
- `isChanged(`g`)` checking if the graph g has been restructured after the previously performed analysis step.

Provided with the parameter value *curType*, the algorithm analyzes every substructure and descends deeper into the graph step by step. This top-down approach is chosen because the directed graph structure can be traversed much more efficiently by starting from the root then to collect base types and search in a bottom-up direction.

The first loop iterates over all elements in the operand list of the signature. If operands are mergeable with the current vertice, all childs (operands) of an mergeable operand are re-linked directly to the current vertice. The edge from the vertice to the operand is deleted afterwards. When the operand is not referenced anymore, it can be eliminated, too. In that way, we get rid of all intermediate mergeable vertices from a path $e_i - e_j - e_k$ by eliminating e_j, which finally leads to a flattened graph.

In the next step it is determined how a change of the graph's structure, which is reported by the function `isChanged`, must be handled. If it was restructured during the previous breadth first analysis of operands, the flattening process must start again beginning with the root node, because a vertice which is closer to the root might have become mergeable. If the graph was not changed, the algorithm recursively descends deeper into the structure by retrieving the operand list of the current vertice and applies the flattening algorithm to all elements of the operand list. The algorithm stops, after having encountered a base type.

To avoid infinite iterations induced by a recursive signatures, a list of already visited vertices has to be maintained. Vertices which are contained in that list are not analyzed further.

Algorithm 5.7.

```
FUNCTION flatten(↓curType, ↓g);
  BEGIN
    visitedlist := visitedlist ∪ {curType};
    FOR (e1,e2) ∈ getOperandList(curType) LOOP
      IF getOperator(e2) ∈ {×,∪} ∧
         getOperator(e2) = getOperator(curType) ∧
         getTypeName(e2) ≠ getTypeName(curType)
      THEN
        FOR se ∈ getOperandList(e2) LOOP
          E(g) := E(g) ∪ (curType, se2);
        END LOOP;
        E(g) := E(g) - (curType, e2);

        IF ¬isReferenced(e2) THEN
          V(g) := V(g) - e2;
        END IF;
      END IF;
    END LOOP;

    IF isChanged(g) THEN
```

```
      g := flatten(t, g);
    ELSE
      FOR se ∈ getOperandList(curType) LOOP
        IF ¬(se2 ∈ visitedList) ∧
          ¬ (isBaseType(getOperator(se2))) ∧
          se2 ≠ curType
        THEN
          g := flatten(se2, g);
        END IF;
      END LOOP;
    END IF;
  RETURN g;
END flatten;
```

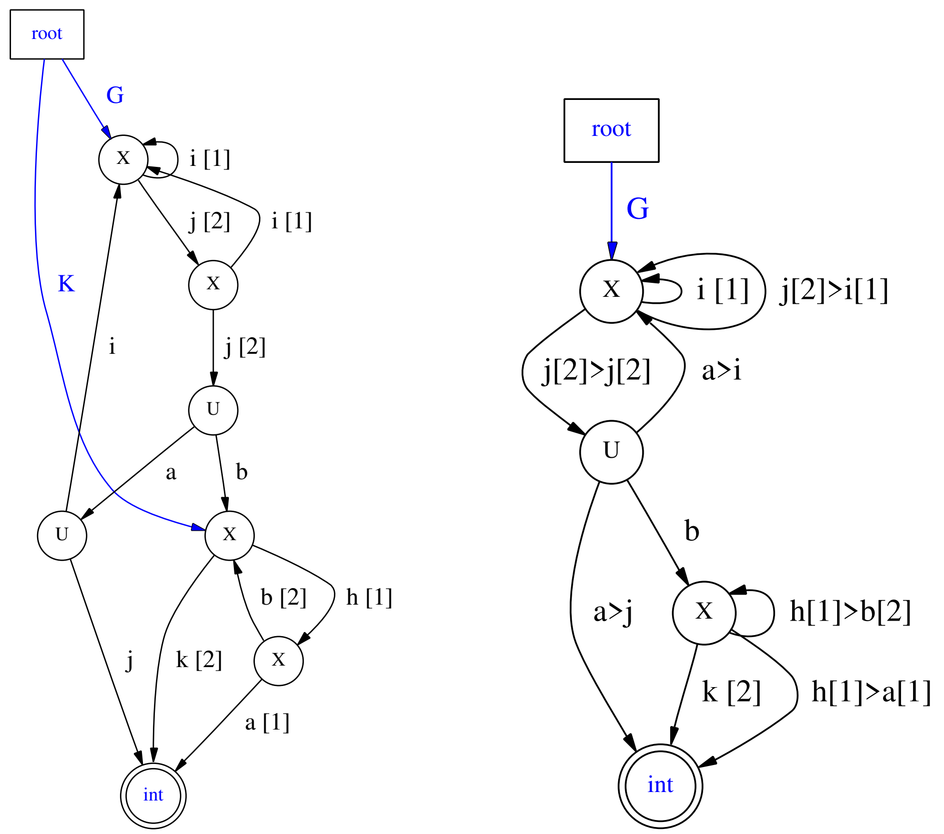
The algorithm does not show in detail how the renaming of merged paths is done. Renaming is performed to track the genesis of an edge. Due to this relabeling (thus, making edge names unique in a graph) the accidental deletion of two edges, which were labeled identically in the original signature graph, is avoided.

The order of operands of the resulting signature is determined by the breadth first approach of the algorithm. Due to this particularity the initial order is maintained. It may be reconstructed by scanning labels from left to right. E.g., the label `j[2]>j[2]` is sorted after a label `j[2]>i[1]`, which conforms with the initial order of the edges.

Example 5.11. *Signature merging* ▽
Given is the following complex signature G expressed in the type grammar previously presented:

```
G = X(i/G, j/X(i/G, j/U(a/U(i/G,j/int),b/K)))
K = X(h/X(a/int, b/K),k/int)
```

The record G is a deeply structured signature built upon cross product and type union operators. Two points are worth mentioning. First, `G` is referenced directly from its operand `i` and indirectly from the signature of `H` via the structure's element `c`. The signature graph of G is shown in figure 5.9(a).



(a) Unflattened graph

(b) Flattened graph of G

Figure 5.9: Signature graph merging of G

The result of the merging process is presented in figure 5.9(b). Every structure successing a cross product was flattened. If their type operators were compatible, the structures in G were merged with its predecessor in the graph. The signature K is not visible in the flattened version anymore. Please note, that the recursive references to G (either direct or indirect) are shown, too. $\triangle$

Flattening is justified by the observation that conceptual close signatures could be implemented differently. If intermediate structures are flattened the focus shifts to the data maintained. Each original path exists in a transformed form in the flattened graph as well. This is clearly an advantage, if the searcher focuses on the concept's aspect only. In that way, a query for a concept may be eased by specifying the matching criterion on flattened signatures. As a byproduct, the understanding of a signature's concept is eased which is due to the reduction of the graph's complexity.

Some of the signature relations presented in the previous sections base on the structure of signatures. Consequently, the relationships established between two signatures s and p may not hold for their flattened counterparts s' and p'. Such a situation is discussed in the following example 5.12

Example 5.12. *Flattening may destroy established relations* $\triangledown$

Consider two signatures $s = \times({}^{a}/_{int}, {}^{b}/_{s})$ and $p = \times({}^{a}/_{int}, {}^{b}/_{\times({}^{a}/_{int}, {}^{b}/_{p})})$. Signature s is a subtype of p and vice versa. They are constitutively equal, too; $s =_c p$. However, their flattened versions, which are $s' = s$ and $p' = \times({}^{a}/_{int}, {}^{b>a}/_{int}, {}^{b>b}/_{p'})$ are not constitutive equal (which is due to the different length of their operand lists) and no subtype relation holds between them. $\triangle$

The flattening operation helps to match signatures, since it hides structural complexity. On the other hand, the resulting signatures may demonstrate different semantics with respect to signature relations. This side effect must be kept in mind when operating on flattened signatures.

5.4.3 Generalized indexing

In the previous sections we introduced various relations between signatures of types and a merging operation to hide structural differences of conceptually equal signatures. With the aid of these relations the "closeness" of a query to a candidate component can be determined. This closeness is stratified according to the respective signature relation.

If two components are exactly equal, they are identical and the query is completely fulfilled. Two components are near to each other, if they are structurally equal, and not that close (in a structural sense) if they are constitutively equal. When two components are exactly equal, structural and constitutive equality is implied as well. Additionally, these components must have the same name, a fact which need not be true the other way round: two components sharing the same name need not be constitutively equal. Trivially, if two components are exactly equal, they are subtype related. Two components which are structurally equal are constitutively equal as well, but they need not be exactly equal. Since subtyping depends on the labeling of operands, which is not checked for structurally equal components, structural equality does not imply subtype relations. The same holds for constitutive equality.

Name equality refers to the identity of string designators independent from any structural property. Therefore, name equality is an orthogonal concept compared to the other signature relations.

Based on these observations we may establish a hierarchy of signature relations which is shown in figure 5.10. Arrows are interpreted as *is-a* properties between signatures relations. As "exact equality" is source of the arc to "structural equality", we know that exact equality between two signatures implies structural equality (and transitively, constitutive equality) as well.

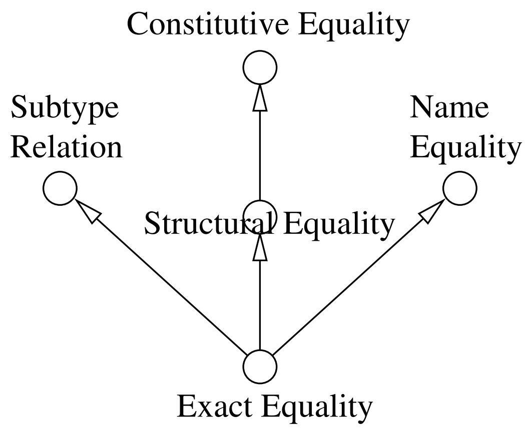


Figure 5.10: The relation of type relations

Successful signature matching must be supported by a *generalized signature index*. Whenever a new signature s is encountered, the relations to all constructs already indexed are computed and s is added to the index structures, according to its relational properties. Transitive relations can be derived dynamically from the existing information, hence, they are not considered in the generalized signature index.

Definition 5.14. – *Generalized signature index*
A generalized signature index GSI is a set of five elements, $GSI = \{NE, ST, CE, SE, EE\}$. Every element is a graph, where V is the vertice set containing all known signatures. Let t_i and t_j be two vertices from V and

- $NE = (V, E_n), \forall(t_i, t_j) \in E_n \mid t_i =_n t_j$, the name equality graph;
- $ST = (V, E_{st}), \forall(t_i, t_j) \in E_{st} \mid t_i <: t_j$, the subtype graph;
- $CS = (V, E_c), \forall(t_i, t_j) \in E_c \mid t_i =_c t_j$, the constitutive equality graph
- $SE = (V, E_s), \forall(t_i, t_j) \in E_s \mid t_i =_s t_j$, the structural equality graph;
- $EE = (V, E_e), \forall(t_i, t_j) \in E_e \mid t_i =_e t_j$, the exact equality graph.

This set contains all information about known signatures. It is exploited whenever a query is entered into the retrieval system. A query is formulated as a signature, tagged with additional information about matching requirements which may concern the whole signature or only parts of it. Requirement may specify whether candidate components should be returned which match exactly (exact equality, name equality) or match relaxedly (subtypes, constitutive equality, structural equality). In that way, queries can be formed which look like this one: "Retrieve all signatures which correspond to $\times({}^{a}/_{int}, {}^{b}/_{\times({}^{x}/_{int}, {}^{y}/_{real})})$. Additionally, the second operand b may match constitutively". Such requirements for relaxed matching stem from two different directions, which are

- the searcher's knowledge about implementation needs, or
- her/his a-priori knowledge about the realization of signatures (components) in the repository. In that way relaxed matching supports opportunistic reuse-by-recall (see section 3.2.2).

The searcher may further relax the search by specifying to apply matching to flattened variants of signatures. The generalized index structure of flattened and unflattened type relations could differ, as already explained and visualized in example 5.12 on page 107.

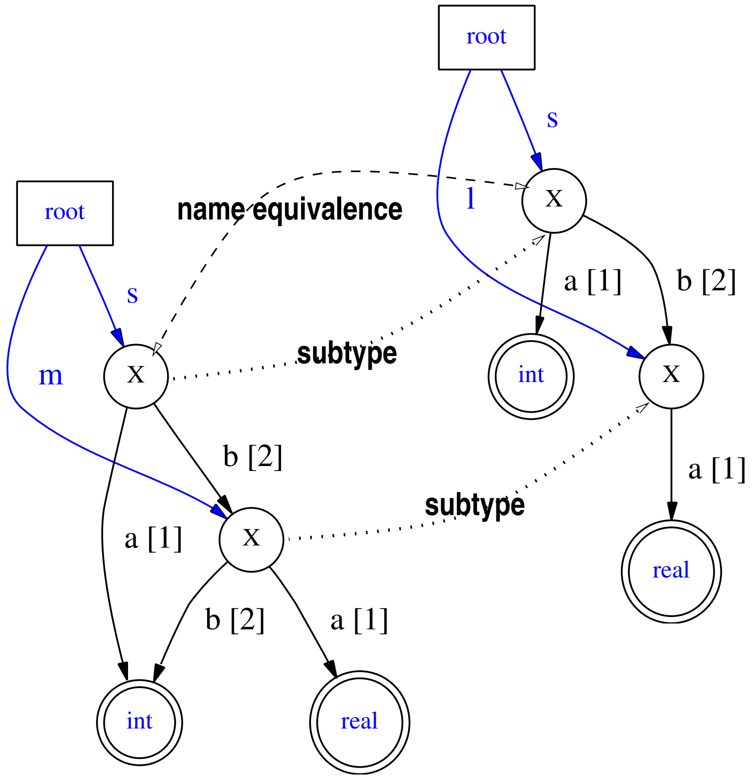


Figure 5.11: Navigation by signature relation

5.5 Final remarks to generalized signature matching

Signature based descriptions of components deal with a very special aspect of components only, their interface structure. Nevertheless they are useful, since the aim of reusable components is to integrate them into a larger system. Integration is based on a clear understanding of interfaces. Components must offer services as interfaces. Hence, signatures obtained by interfaces, are provided for free and the generalized indexing structure can be established automatically.

The information such obtained is sufficient to identify components on a pure structural level. However, if structure does not provide with sufficient information for locating a component, generalized signature matching is used as a filter for identifying structural relevant components. Therefore, generalized signature matching is not considered to serve as an stand-alone solution. It is an important concept towards a holistic approach for *development with reuse*, where both aspects, interface structures and functional requirements, are equally important for judging the reusability of an asset.

5.6 Summary

In this chapter we presented the concept of signatures which are implementation-independent representation of types. We discussed rules and hints for transforming types from programming languages to signatures. Signatures are represented intermediately in a *signature expression sublanguage*. For the purpose of detailed analysis, signature graphs are introduced and rules for transforming expressions into graphs are given. On the basis of signature graphs we established five relations. These relations are tools

1. for determining the closeness of two signatures and
2. for enabling flexible matching mechanisms for component retrieval.

Furthermore, a pragmatically motivated signature simplification, called *flattening*, is presented, with which unimportant internal structures can be neglected. This adds to the flexibility of matching, too. At last, an index structure is presented, which exploits all concepts introduced here. This index is a basis for a component repository, where signatures are used as the components' descriptions.

The contribution of this chapter is given in its novel approach to establish different levels of signature relations administered in one indexing structure. Some of these relations are pragmatically motivated and not yet mentioned in the literature. The relations a filter initial component sets with the aim to generating sets of reusable candidates. In that way the necessary flexibility is provided which renders practically the overall component retrieval process.

In the next section we go one step further building on the results obtained in the current chapter. After having narrowed the candidate set, either with structural or conceptual matching, these candidates must be evaluated in detail concerning their functionality. Whereas signature matching aims at increasing the level of recall, in the following chapters the focus is set on precision. This is realized by enhancing ideas from extensional description methods (presented in section 4.6 on page 60) towards an extended behavioral description and retrieval technique.

6

Behavioral Description of state-less Components

In the previous chapters we introduced generalized signature matching as a flexible technical filter to narrow the set of reusable candidate components. If the cardinality of the candidate set is sufficiently small, the additional step for selecting the best fitting component is to check the conventional descriptions for conformance with the searcher's expectations only. However, if the number of candidates exceeds a certain threshold then analyzing each detail of the descriptions is cumbersome. In this chapter we build on the promising approach of extensional descriptions introduced as behavior sampling [175, 174] in section 4.6. There we identified the following drawbacks of behavior sampling (or its successor, generalized behavior sampling [92]):

- Focus on the *operational profile*, which is the stress of frequent use cases as main source of behavior. Frequent use cases do not necessarily lead to better and more significant description. Exceptional cases, although not appearing frequently, describe important characteristics, too. This is especially true if a searcher is a domain expert who knows about the correct treatment of "non-mainstream" situations.
- *Output determination*, where the searcher has to determine the correct result of an input. Such a determination might be too difficult or too costly,
  - if the exact functionality is not specified sufficiently, or
  - if the searcher does not bother about the exact outcome of a component's computation, or
  - the searcher is satisfied with any transformation within a certain limit of tolerance.

- *Interactivity* is the situation where a searcher wants to be guided during the retrieval process or at least wants to get some feedback about the direction of the search.

In this chapter we introduce an approach to describe components which lacks these drawbacks which is called *static behavior sampling* (SBS) (in contrast to the (dynamic) behavior sampling technique developed by Podgurski and Pierce). The main idea of SBS is to establish an indexing structure which can be browsed interactively by a searcher.

Similar ideas of browsing content-based representations are developed for the area of retrieval for multimedia databases. As an example, the PICHUNTER [54, 55] system presents pictures to the user and she/he selects the best fitting one. The next level consists of better fitting pictures (which is evaluated by constant feedback loops) thus, leading to a step-by-step approximation of the searcher's intention. However, to our knowledge, such ideas are not yet transfered to the field of component retrieval. SBS is a first step towards this aim.

In the SQUEAK Smalltalk system a similar technique to retrieve object oriented classes from a repository is implemented [40, 105]. Here, the user may enter examples and the retrieval systems finds promising candidate classes, which are able to reproduce the examples. In contrast to SBS, here the exact signature of the yet unknown classes (or their methods) must be known in advance, which often makes a query infeasible.

6.1 Static Behavior Sampling – SBS

6.1.1 General idea of SBS

Behavior sampling is based on the *on-the-fly* generation of output by executing the components. Indeed, this is the step causing the majority of its impediments. In our work we want to avoid the direct execution without loosing the simplicity and expressiveness of input-output descriptors. To do so, we do not analyze query-time executions, but former executions, which are stored as test cases. Considering some important preconditions (which we will discuss in section 6.2.3 on page 126) some test cases render valuable as knowledge base for producing high quality descriptions. A test case is a sequence of data tuples (also called test tuples).

Definition 6.1. – *Data tuple*
A data tuple is a pair $(\vec{i}, \vec{o})$ of two vectors. The input vector $\vec{i}$ holds all elements which are input to a function, whereas the output vector $\vec{o}$ holds all elements which are the result of the function call.

For example consider the function `HShift (source : Point, dist : Real) : Point` specified in the example 5.1 on page 70). It is called with the following values: `([1.0;1.0], 1):[2.0;1.0]`. Values of record structures are depicted in squared brackets. The data tuple $dt = (\vec{i}, \vec{o})$ contains the input vector of the form $\vec{i} = \langle [1.0; 1.0], 1.0 \rangle$, resp. the output vector of form $\vec{o} = \langle [2.0; 1.0] \rangle$.

By analyzing already available test cases the time-consuming and costly execution of software is avoided or, when test data is not provided with the component, it is shifted back at least from the retrieval phase to the indexing phase. The indexing phase is concerned with the integration of one or more components into the repository.

Test cases reflect an important section of a component's functionality. Hence, they can be considered as *partial specifications*. The basic principle of SBS is to extract this valuable information from the large set of test cases to get a "good" partial specification. Since testing does not aim at producing descriptions, test cases must be augmented with additional input-output tuples in order to raise discriminative power. We refer to such a description set representing a partial functional specification of a component as *data points*.

Definition 6.2. – *Data point*
A data point is a data tuple used as SBS descriptor for a component. Any data tuple stored in a SBS repository is a data point.

A SBS-based repository is organized in the following way:

1 The repository is partitioned according to the generalized signature of its components. Each partition holds components conforming to a signature. Obviously, a searcher cannot use the information of a partition's data points directly in order to guide her/his search. This is due to the unstructured mass of input-output tuples.

2 Data points of a partition form the knowledge base from which a classification tree as indexing and browsing structure is extracted. For that purpose, data mining techniques are utilized, which extract unknown information from structured or unstructured data [79]. The field of data mining in general addresses the question of how to use data best to discover general regularities hidden in it, thus, improving the process of decisions making [150].

6.1.2 SBS's search support

The initial information provided by the searcher is the signature for selecting the appropriate partition. As it is discussed in detail in chapter 5 generalized signature matching provides with the flexibility to gather all relevant components. The search process itself is a dialog in the form of simple query-answer cycles. The search process is initiated by the repository system by asking the searcher questions in the form: "Here are some examples, $f(i) \to o_1,\ f(i) \to o_2,\ \dots,\ f(i) \to o_n$. Which of the results $o_1,\ o_2,\ \dots,\ o_n$ is the one you expect from the component your are searching for?" Thereupon the searcher selects the output which is in her/his opinion the correct result on the presented input. In the case of a continuous output domain, the question is presented as: "Is the result of $f(i)\ \theta\ o$?" The operator θ is some relational operator defined on the domain of o. The searcher then may to answer with "yes" or "no". In that way, even though the searcher does not know the exact answer, vagueness leads to an exact result.

Due to the initial selection of a signature partition and the underlying balanced classification tree, after a few dialog cycles the search leads to an end. Thus, the characteristics of the search turns out to be a browsing activity.

Finally, the browsing activity leads to a situation where either

- a component has been identified, or
- a question is not decidable.

If the answers to the queries specifies the searched functionality, the component such identified is the best candidate available in the partition. There exists no other component in the partition conforming to the partial specification built by the data points collected during browsing. The browsing process leads always to a component in the tree! The final judgment remains with the searcher, but there is only one component left to be verified.

In the case of an undecidable question, the searcher has not sufficient knowledge about the domain. Then at least, the uncertainty can be handled by taking a quick look at deeper levels in the browsing tree. The browsing activity can be backtracked at any point in time to allow a re-adjustment of intermediate answers and to choose different decision paths.

6.1.3 Partial specifications build upon data points

This section is concerned with the necessary properties of data points in order to establish a high quality knowledge base. Furthermore, we describe how are selected from a set of test cases.

From a structural view, data points do not differ from data tuples. However, due to their specific properties some data tuples perform better then others for the purpose of specifying the behavior of components.

In general a specification describes the behavior of a system exactly. It focuses on the aspect of "what" has to be done (in contrary to "how" it must be performed). This part of the work is partially inspired by the relational specification method of Mili et.al. [146, 149], who specifies the behavior of a program by defining relations on the input and output domain[1]. With SBS, not the domain itself, but values from that domain are basis for the specification. Since the a relation cannot be described by extensional values completely, data tuples represent only a subset of a relation. For that reason, we refer to SBS as a *partial relational specification* for a component's functionality. Another starting point can be found in the field of specification-based testing [178, 197] where a formal specification serves as an unambiguous standard against which the behavior of the program unit under test can be automatically compared. Additionally, formal specifications are the mechanism to develop systematically test cases. In SBS this course from formal specifications to test cases is followed into the reverse direction: Test cases form the vantage point for building a specification.

Whereas the purpose of a specification is to define the totality of a components behavior (discriminating it against the universe of all components), in SBS it is sufficient to define just as much as it is necessary to discriminate between different components in one partition. Hence, the partial description provided with a data point specification adequate for the SBS approach.

A necessary requirement for a good partial specification built from data points is the high quality of the underlying data regarding its discriminative power.

[1] As discussed in section 4.4.1 on page 55.

Test data

Test data is generated with a clear goal in mind: A test case checks for a possible fault and no other test case should aim at the same target. Consequently, well designed test suites cover a broad range of the component's functionality. Thus, test cases represent the majority of data points.

Characterizing tuples

Specifically interesting are tuples which enable a unique identification of a component with respect to all other components in a partition. We refer to such data as *characterizing tuples*. The principle is shown in figure 6.1, where two functions are shown. The function $f(x)$ is a polynomial function whereas $g(x)$ is linear. In case $g(x)$ is tested only with the test cases indicated as circles in the figure, these tuples mark its linear character very well. On the other hand, if the same input describes $f(x)$, that may be a sufficient, too (assuming that some few additional points are provided). However, when considering both functions, on the basis of these test cases the function cannot be distinguished. The test cases indicated as boxes in figure 6.1 are then a much better choice: (1) both functions are characterized and (2) the test cases enable a discriminating of them. Hence, the test cases represented as boxes may be selected for data points.

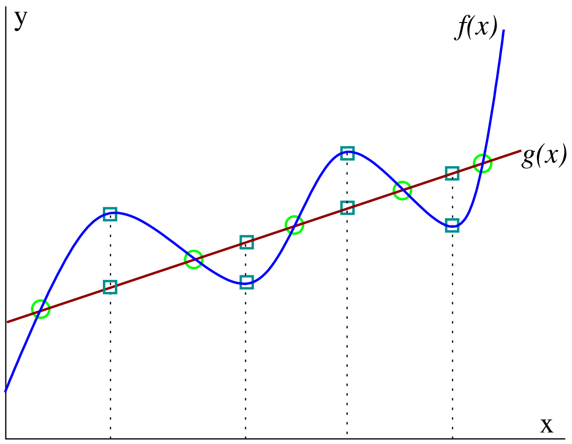


Figure 6.1: Characterizing tuples

Characterizing tuples cannot be identified when looking at the test cases of one component in isolation. Only the analysis of all data points of a partition's components lead to the selection of those which demonstrate the most discriminative power. To which extent a tuple is really characterizing can be verified only when taking the whole partition into account.

> **Definition 6.3.** – *Characterizing tuple*
> A characterizing tuple t with respect to a component c is a data tuple whose input vector leads to an output vector unique for the partition in which c is placed in.

At the first glance, it seems to be overanxious to strive for characterizing tuples. A significant overlap in the behavior of components should be unusual for a well organized repository of functionally different components (see the discussion on the data point property *necessity* on page 126, too). This may be true for a simple mathematical domain as shown in example 6.3 on page 137. But for other application domains, the situation is likely to change. Discriminative power helps then to optimize the automatic analysis process.

Striking samples

When organizing data points, an optimized indexing process is not alway the main goal. Consider the hypothetical case, that the trigonometric sine function is sought which is described by the data point $(\langle 5287.89\rangle, \langle -0.56\rangle)$. It's arguable, whether a searcher, although a domain expert, immediately detects the sine functionality of such a partial specification. A more eye-catching sample then would make the identification of the sine much easier. Data points which support the searcher in identifying components at a first glance are called *striking samples*. Striking samples may be orthogonal to characterising tuples and they may not improve the characteristic property of a partition's data points. But for a human domain expert they are immediately attributable to a functionality. For instance, in the domain of trigonometric functions the functions $\sin, \tan, \sinh, \tanh,$ and $\arctan$, when provided with the input 0, return the output 0. A domain expert immediately recognizes this behavior, thus, many other trigonometric functions can be excluded from the candidate set.

Nevertheless, for browsing they prove to be very useful, since a searcher expects such behavior from the component. If other components stored in the repository demonstrate the same behavioral pattern the browsing process is not hampered: in combination with characterizing tuples they are filtered out easily.

Definition 6.4. – *Striking samples*
A striking sample is an element of the data points of a partition, which is immediately attributable to a component by an domain expert.

Both types of data points and necessary for describing components on a behavioral basis [181]. After having identified them, the question arises, how to obtain these data points. Obviously, the main source is the set of the component's test data. But testing is primarily concerned with detecting deficiencies and not with producing descriptions; this is the reason why not all test cases make good data points. Additionally, test cases are generated with various techniques, each stressing a special aspect. As a consequence, we have to distinguish carefully for which purpose test cases are generated when selecting data points.

6.1.4 Where do data points come from?

In testing theory, a distinction between *functional* and *structural* testing [53] is popular. Synonyms for functional testing are black-box [21], data-driven, input-output driven [158], or requirements-based testing. For functional testing, test cases are designed without having the internal program structures in mind. For generating test cases it is refered to the specified functionality only. Random testing is a variant of functional testing where the input is generated automatically. Generating test cases for random testing may be refined by call on a random distribution which is chosen according to an operational profile. In such a way the tests concentrate better on (functionally) important parts of the program under test.

The second test strategy, *structural testing*, is refered to as white-box, glass-box or design based testing as well. Here, test cases are chosen according to the internal structures of the program under test. The idea is to analyze the program's control structure and select input values with respect to this structures. In the way that the execution of different regions within the program is controlled.

Characterizing tuples

Since characterizing tuples mark off components within a partition, it is important that the set of available test cases is sufficiently large to guarantee this. Furthermore, as many components as possible should be tested with *all* inputs of a partition's data points. The selection process has to consider this aspect, too.

To get characterizing tuples, the test technique used to generate data is not important, unless the number of test cases is too small. In such a case, random testing is the best choice for producing characterizing tuples quickly and cost effectively.

Striking samples

Finding striking samples automatically is difficult, since the user has to judge their intuitiveness. In testing theory, some functional testing approaches deal with the generation of data placed at domain boundaries of components. Domain boundaries are "means by which a domain is defined" [21, p 147]. Domain testing partitions the input domain into distinct regions identified by a set of inequalities over the input vector. The assumption here is, that different domain partitions require different ways of processing as well, whereas within a domain partition always the same process is applied. Thus, input elements of a domain partition are considered as equivalent to each other. Of particular interest are domain boundaries which reveal characteristics of components best [158]. For instance, consider the following example which is taken from the book of Beizer [21, p 151]:

Example 6.1. *Domain boundaries* ▽
Given are the US income tax calculation rules for singles from 1993:

Domain	**Process**
$0 <$ tax-inc $\leq 22,000$	tax $= 0.15 *$ tax-inc
$22,100 <$ tax-inc $\leq 53,000$	tax $= 3,316 + 0.28 *$ (tax-inc $- 22,100$)
$53,500 <$ tax-inc $\leq 115,000$	tax $= 12,107 + 0.31 *$ (tax-inc $- 53,500$)
$115,000 <$ tax-inc $\leq 250,000$	tax $= 31,172 + 0.36 *$ (tax-inc $- 115,000$)
$250,000 <$ tax-inc	tax $= 79,722 + 0.396 *$ (tax-inc $- 250,000$)

In that example domain partitions can be easily identified and they are specifiable as *domain inequalities*. The domain boundaries are \$0, \$22,100, \$53,500, \$115,000, and \$250,000. If an application engineer (with the knowledge of a tax expert) searches for components performing the tax calculations, she/he would certainly take a closer look at the domain boundaries from both sides. If only these boundaries are stored as input in the test base (which is rather unlikely) the sequence of striking inputs is nevertheless meaningful due to the domain-wide known particularity of tax calculation thresholds. △

Domain boundaries themselves and test data obtained from domain boundary analyis are candidates to be selected as striking samples, since this testing technique is based on the specification of functional requirements. Most often, boundaries are not identifyable automatically and domain experts have to specify striking samples. This is especially true, if good striking samples are not only found on the boundaries, but are standard reference values.

After having identified three different types of data points and discussed some hints of how to obtain them, we present the necessary infrastructure for a SBS-based repository in detail.

6.2 SBS Repository

The basis for a classificiation in a SBS-repository $\mathcal{R}$ are generalized signatures as presented in chapter 5[2]. The set $\mathcal{C}$ contains all reusable componentes $c \in \mathcal{C}$. A repository stores components together with the data points describing them.

6.2.1 The coarse grain structure

The set of all signatures of a repository is denoted as Σ. A specific signature of that set is denoted as $\sigma_j \in \Sigma$, where j runs from 1 to the number of all identified signatures. We assume that we can infer for each component c its signature. The set $\mathcal{C}$ of all components then is further partioned according to the signatures σ_j. We call such a partition P_{σ_j}. Signatures provide a structurally motivated classifier for reusable components. This does not hamper any other classification dimension concurrently established. By defining a query, the searcher may indeed formulate it in specifying a signature and additionally restricting the search further to e.g. "financial" components. Albeit, the treatment of multi-hierarchies is not discussed in this work.

Generalized signature matching provides much flexibility. How the repository is partioned depends therefore on the granularity of the signature criterion which could be the subtype relation, constitutive equality, or structureal equality. As it depends on the type of question and the domain in which the system is placed, it cannot be decided in advance which criterion is the best one. It is even an option to reorganize the partion dynamically or, alternatively, establish multiple partition

[2] We abbreviate the term "generalized signatures" in the remainder of this work as "signatures".

views onto the repository, if there is the necessity to switch from one criterion to another one.

Components with several implementations

Due to the fact that data points in SBS are partial functional specification, the description of behavior may not suffice to characterize components completely. This may leed to a situation where several algorithms demonstrate identical behavior on the functional level [154]. A typical example are implementations of sorting algorithms. To distinguish between them on the functional level solely is not possible, unless non-functional behavior, e.g. time complexity, is explicitly provided.

As a consequence, in SBS when speaking about a component c the functionality described by data points is meant. This is the reason, why SBS-components are named as SBS-*component specifications* as well and the distinction between the component specifications and their implementations is necessary. However, in this chapter, when speaking about components, in fact the component specifications defined by its SBS data points is meant.

Since we cannot obtain more information from the given data points, the component c itself might refer to multiple implementations i. Then a searcher must take a closer look to the implementations of the component and filter out those which do not fulfill the demanded non-functional requirements.

Implementations for several components

Some implementations have a polymorpic interface structure. This is especially true for reusable components. Polymorphism is useful for offering a broader range of solutions differing in technical details only (such as data types of parameters). On the other hand, this complicates the static analysis of signature information. Consider the given standard ANSI-C function quicksort which takes an `array` of `n` elements of size `size`. Its last argument is a pointer to a function used to compare the elements. As the type of the elements is not known yet, the algorithm specifying how to compare them is instantiated at runtime. This characteristics makes the determination of an exact signature for `qsort` impossible.

```
void qsort (void *array, size_t n,
            size_t size, int(*compare)());
```

As a consequence, the `qsort` function is placed in several partitions; each is a valid place for the component. Implementations can demonstrate more than one behavior and therefore can be attached to a multitude of components [154] as well.

However, an implementation may also be attached to several components, which are placed in the same partition. This would point out to an implementation which can be parametrized to demonstrate more than one functionalities (like `qsort`). Such an implementation is associated with more than one SBS-component specification. However, this does not hamper the browsing process and due to the component's different sets of data points, different behaviors of one implementation lead always to separated component specifications (regardless the partition in which the component is placed in).

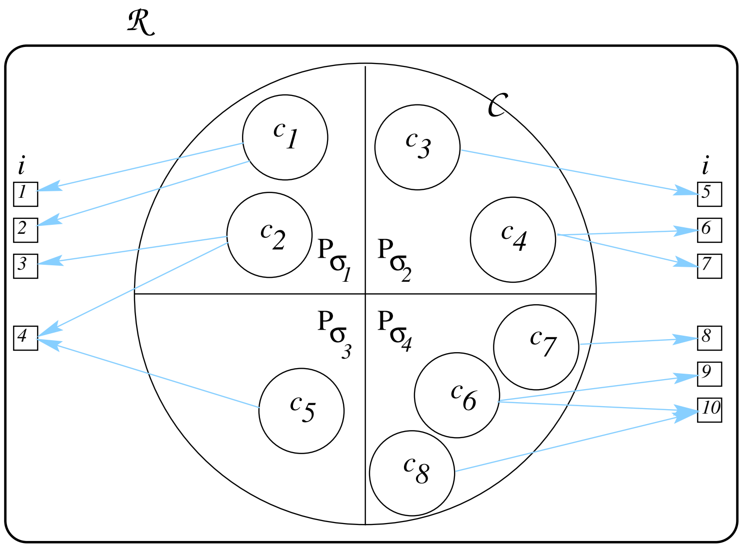


Figure 6.2: The SBS-repository structure

A schematic structure of a partitioned repository $\mathcal{R}$ is presented in figure 6.2. It is divided into four partitions according to the four identified signatures. In this example there are ten different implementations belonging to eight components (component specifications). Components are depicted as circles, their implementations as boxes outside the signature partitions structure of the repository. Due to the preciding step of generalizing signatures implementations cannot be assigned to a specific partition. E.g., component c_1 is mapped to two different implementations i_1 and i_2, which could be realisations of different sorting algorithms. The same happens with c_2, where its implementations are i_3 and i_4. Albeit, the implementation i_4 is refered to from component c_5 as well, which is placed in a different partition P_{σ_3}. The implementation i_4 could be a function with a polymorph signature, where the concrete type of its polymorphic parameters is set during run time. The implementation i_{10} demonstrates different behaviors which becomes evident in its multiple assignment to the components c_6 and c_8. The reason for that assignment is functional polymorphism.

In the next section we will describe how data points are organized to serve as knowledge base in a partition P_{σ_i} of the repository $\mathcal{R}$.

6.2.2 The SBS-partition structure

In the previous sections it was described how components (and their implementations) are classified according to their signature. The resulting coarse grained repository structure is divided into partitions. Since we discuss its structure on the level of one partition, in the remainder of the chapter we do not denote a signature with an index.

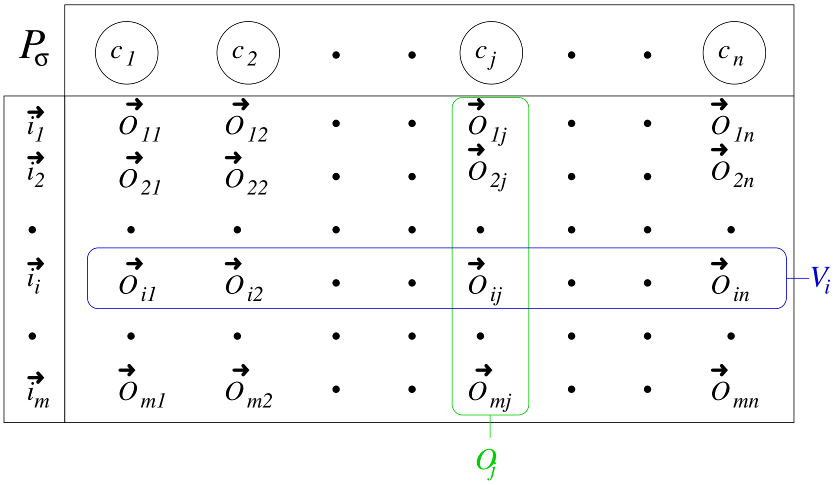


Figure 6.3: A partition P_σ in a SBS-repository

A partition P_σ holds a set of components $\mathbf{C} \subset \mathcal{C}$ whose elements conform to the signature σ. We refer to $\mathbf{C}$ as *signature congruent components*. Furthermore, a data point dp (which is a data tuple) formally consists of an input and an output vector, $dp = (\vec{i}, \vec{o})$ with the input vector $\vec{i} = \langle i_1, \ldots, i_m \rangle$ and output vector $\vec{o} = \langle o_1, \ldots, o_p \rangle$. The elements of a vector are values, which can be structured further as well. Each data point is assigned to a component in the partition which makes it unique. If the partition's signature is $\sigma = I \times O$, then its components are functions of the type $c : \sigma$, with $c(\vec{i}) = \vec{o}$ and $\vec{i} \in I$, resp. $\vec{o} \in O$. The data points are organized in a tabular form in such a way that there are m rows and n columns. Each row is named with the corresponding input $\vec{i}_i$ and each column is named with the corresponding component c_j. An entry $\vec{o}_{ij}$ in that table is the result of the call to $c_j(\vec{i}_i)$. The design of the data point organization within a partition is shown in figure 6.3 on the preceding page.

6.2.3 SBS partition properties

In the current section we describe which properties are necessary to hold for data points in a SBS-partition and how data points are organized within a partition.

Initial completeness

A partition consists of components (with the references to their implementations) and the set of data points describing them. A necessary property to enable their analysis is *initial completeness* [181] which ensures that for each component c_j is provided with all input vectors $\vec{i}_i$ and the result of the call $c_j(\vec{i}_i) = o_{i,j}$ is defined.

Due to the homogeneous structure of the partition's signatures initial completeness can be established easily, if the available set of its test data is sufficiently large. If data points are missing, then the components concerned must be executed with the inputs. In general, the testing process is not totally decoupled from the asset management and the fulfillment of the requirement of initial completeness should be demanded from the testing team, because it can be provided with all input values of the SBS repository.

Necessity

The second property which is important to provide discriminative power in a partition, is the *necessity* of data points. Necessity ensures that two different inputs really stimulate different outputs. This must hold for all inputs in a partition. There does not exist two different input vectors $\vec{i}_i$ and $\vec{i}_{i'}$ with their *value sets* $V_i = \{\vec{o}_{i_1}, \vec{o}_{i_2}, \ldots, \vec{o}_{i_n}\}$ (the set of output vectors generated by the components

$c_1, c_2, \ldots, c_n$ provided with an input vector $\vec{i}_i$) and $V_{i'}$ in such a way that $V_i = V_{i'}$. If two input vectors $\vec{i}_i$ and $\vec{i}_{i'}$ really generate identical value sets, then no discriminative power is added to the knowledge base represented by the data points and one of them must be omitted. Trivially, necessity is implied, if for a partition all its data points are characteristic tuples (section 6.1.4 on page 120).

Descriptor uniqueness

The next property adds to the discriminative power for data points in a SBS partition as well. It ensures, that given the whole set of input vectors of a partition, the sets of output vectors for two different components are distinct. Let the *component output set* O_{c_j} containing all output vectors of a component c_j. If an initial complete partition descriptor unique, there must not exists any other component output set $O_{c_{j'}}$ and $O_{c_j} = O_{c_{j'}}$. However, if two component output sets are identical and initial completeness holds, then the components' specifications are identical as well. Consequently, within a descriptor unique partition the implication $O_{c_j} = O_{c_{j'}} \Longrightarrow c_j = c_{j'}$ holds. Therefore, we call a component output set O_{c_j} the *descriptor* of c_j.

In the following section we use the descriptive capability of data points to build a browsing hierarchy. The technique used here for analyzing data points is the method of decision (or classification) tree construction. It is well known in the field of data mining, although its application comes into ones mind only, when preparing the partition as knowledge base.

6.3 Data point analysis

Due to the properties initial completeness, necessity of data points and descriptor uniqueness a partition holds a set of *partial functional specification* of signature congruent components. But the descriptions are hidden which is due to the unstructured way in which they are assigned to components. The question now is, how can an representation be generated which helps to identify components easily. This task is typical for the field of data mining or machine learning[3].

In machine learning, one can differentiate between

[3]The term "data mining" is rather overloaded in the literature. Sometimes it refers to the whole process of knowledge discovery and sometimes to the specific machine learning phase. Machine learning is mostly used in connection with the scientific study of induction algorithms [118].

- unsupervised learning where no feedback is given to guide the learning process and
- supervised learning where an agent gives direct feedback about the appropriateness of the process' performance [123, page 8].

In supervised classification learning it is assumed that each instance of evidence holds some information in form of an attribute specifying the class of that instance. This attribute is called *dependent attribute*. The goal of the learning process is to induce a concept description predicting this attribute accurately. One form of supervision during the training phase is to specify in advance the classes in which the instances will fall.

Classifying with unsupervised learning means that the classes are not known and the algorithm must distinguish desirable actions from undesirable ones on its own.

In SBS, a supervised machine learning technique is appropriate, since the "classes" (speaking in terms of maching learning) are indeed known in advance. The data points describe components and therefore the term "class" is equivalent to the term "component" in SBS.

Many classification models are proposed [79, 123] in the literature: neural networks, genetic algorithms, Bayesian methods, log-linear and other statistical methods, decision tables and lists, and tree-structured models, so called *decision trees* or *classification trees*. For the purpose of data mining decision trees are appealing for the following reasons [79]:

- The resulting classification models are given as intuitive representations and are therefore understandable.
- The algorithms for constructing decision trees do not need any domain knowledge other than the knowledge base.
- The accuracy of the prediction is equal or higher than any other classification models.
- Decision tree algorithms are fast.
- Additionally, they are scalable and, therefore, they can analyze very large knowledge bases.

Some authors [144] sometimes argue in favour of rule-based representations. For large domains with many classes and/or many attributes they judge the readability of tree structures as poor. When the domain is complex, trees tend to become “bushy” and are therefore difficult to understand. Rules on the other hand are modular and can be read in isolation. However, decision tree algorithms are usually very fast and as a compromise many algorithms build decision trees which can be transfered to rule based representations [192]. Here, this approach is followed, too.

6.3.1 Decision tree preconditions

The simplest and most widely used form of decision trees are *univariate* decision trees [123], where a set of instances is divided into mutually exclusive subsets at each level. This splitting is based on the value of a single predictive attribute (thus the term *univariate*). A decision tree is a tree graph and each inner node is labeled with the predictive attribute and each leave node is labeled with the name of a class (component). Each edge originating from an inner node is labeled with a splitting predicate involving only the node’s predictive attribute. Due to the mutual exclusive subsets of each tree level the sequence of attribute-value pairs from the root to a leave node is unique.

In figure 6.4 on the following page an example of a decision tree is shown. It explores, whether or not it is recommended to take a walk in a certain weather situation (example taken from [97]). Inner nodes are labeled with the attributes of the observed domain (view, humidity and windy) and leaf nodes represent the two classes: “take a *walk*”, and “stay at *home*”). Edges are labeled with conditions holding on that specific path, e.g. “The view is sunny”.

The following properties must hold in a domain to render it suitable for applying a decision tree algorithms [192]:

Attribute-value problem description: The knowledge must be describable by a set of attribute-value pairs. The number of attributes must be fixed. The values may be of continuous or discrete nature.

Predefined classes: Due to the supervised learning technique, all classes of the domain must be known in advance.

Non-overlapping classes: No classified instance must belong to more than one class. Therefore, the classes must be disjunct.

Sufficient data: Due to the underlying statistical analysis, the algorithm depends heavily on a significant amount of data.

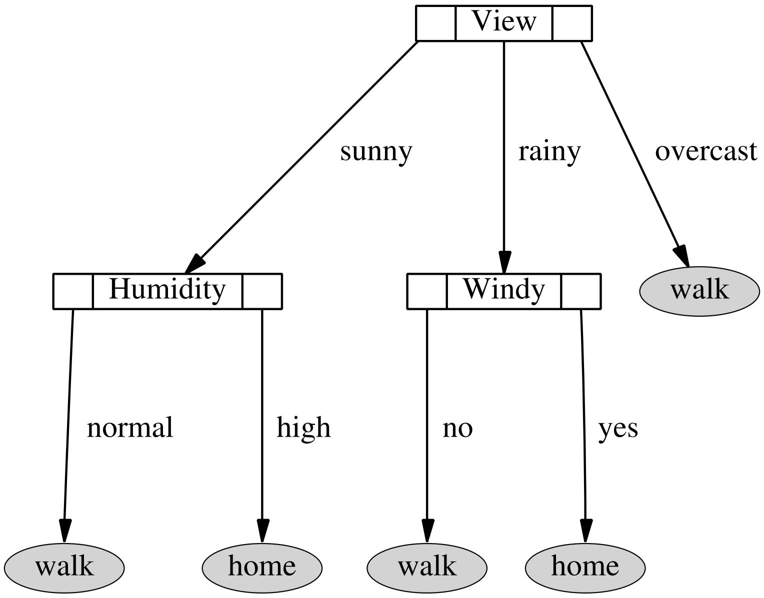


Figure 6.4: A decision tree to determine whether it is recommended to take a walk

Boolean decisions: All decisions necessary for assigning instances to classes must base on relational operations leading to ”true” or ”false” answers. Checking complex conditions which e.g. are expressed in predicate logic cannot be treated.

To render decision tree algorithms feasible for SBS, these preconditions must hold for data point domains as well [181]. Each precondition must be checked:

Property-value problem description: Due to a partition’s property of initial completeness, each component was executed with all data points’ input vectors. For each component, an input vector generates a set of output vectors. Therefore, input vectors are properties characterizing a certain aspect of the components (the behavior issued on that input). The set of all values for an attribute are the elements of its output vector. Furthermore, initial completeness ensures that during the analysis process no more attributes are introduced. If for an input an output cannot be determined, it is allowed to introduce *meta output vectors*. A meta output vector is used to

- make behavior explicit which cannot be visualized otherwise, such as error reactions, file accesses or temporal particularities (indefinite loops).
- specify (yet) unknown values, if executing a component on a given input is not possible or is too costly.

Hence, the required attribute-value description precondition is fulfilled.

Predefined classes: Classes are abstractions of elements which can be characterized by common properties. Properties (attributes) in SBS describe components. Because of the discriminative power of data points and the properties necessity and descriptor uniqueness a class holds only one instance, which is its component. In that sense the term “class” in SBS is equivalent to the term “component”.

The implementations of a component are not considered as the class’ instances. The set of implementations assigned to classes belongs to an abstraction layer which is not subject to the SBS classification process.

Non-overlapping classes: As we demand descriptor uniqueness, the set of data points of a component characterizes exactly this component. Implementations with functionality described identically by data points are assigned to the same component.

Sufficient data: As described in section 6.1.4 on page 120 the main source for data points are well chosen test tuples. If the amount of data is not sufficient, additional data points can be generated by random testing. Due to the partition’s properties *necessity* and *descriptor uniqueness* the discriminative quality of data points does not deteriorate by adding more data points. Additionally, our experiments showed, that for high quality data in most cases the number of data points is not significantly higher than the number of the partition’s components.

Boolean decisions: Boolean decisions on simple relational operators (e.g. $=$, $<$, $\leq$, $>$, $\geq$) demand independence of properties. The trait of SBS components is their independence on internal states. Properties do not depend on each other, in the sense that the order of input vectors makes a difference on the component’s output set. Hence, for each attribute (input) one can decide, whether a condition expressed as boolean decision holds.

All necessary preconditions for applying decision tree algorithms for SBS are given. In the following section the main features of these algorithms are shortly discussed.

Induction of decision trees

The task of a decision tree algorithm is to learn a set of rules (the decision tree) which are hidden in a given set of classified examples. Such training instances are vectors of attribute-value pairs. The idea is to pick a predictive attribute A_p with values $a_1, a_2, \ldots, a_r$ and split the training instances into subsets $S_{a_1}, S_{a_2}, \ldots, S_{a_r}$ consisting only of instances having the corresponding attribute value. If a subset has only instances in a single class, that part of the tree ends with a leaf node labeled with the single class. In the other case, when instances belonging to different classes are encountered, the subset is split again, recursively, choosing and using a predicitive attribute once more. If the predicitive attribute is discrete, the number of values is the upper bound for the number of outgoing edges. In the case of a continous type, a value for a binary decision is chosen, which splits the set into subsets of a balanced number of the instances' classes.

The following algorithm 6.1 is the basic scheme for a divisive induction of univariate decision trees [123]. The initial step, which is not shown in the algorithm, is to generate a global root node for the decision tree on which the algorithm operates further on.

Algorithm 6.1. ***DTI*** *– Decision Tree Induction*

```
ALGORITHM DTI (INPUT Set O) (* Set of observations *)
  IF O = ∅ THEN
    Generate leaf labeled null
        (*No Hypothesis about classification*)
  ELSIF all elements of O belong to the same class c
    Generate leaf labeled c
  ELSE
    Select predictive attribute A_p
      (* attribute with greatest information gain *)
    Generate leaf labeled A_p
    IF A_p is discrete THEN
      FORALL values a_i ∈ A_p DO
        Generate edge a_i leaving A_p
```

```
        Determine $o_i \in O$ with value $a_i$ for Attribute $A_p$
        Call DTI($o_i$)
      ENDFOR
    ELSE (* $A_p$ continuous -- binary decision *)
      select $a_i$ splitting $O$ best
      Generate edge with $A_p < a_i$ (* left branch of DT *)
      Determine $o_i \in O$ where $A_p < a_i$
      Call DTI($o_i$)
      Generate edge with $A_p \geq a_i$ (* right branch of DT *)
      Determine $o_i \in O$ where $A_p \geq a_i$
      Call DTI($o_i$)
    ENDIF
  ENDIF
END DTI.
```

The question remains how to calculate the information gain associated with a predictive attribute A. There are many measures proposed [123] and we present the one used in the ID3-decision tree (*I*terative *D*ichotomizer (version) *3*) algorithm of Quinlan [190, 191]. The information gain measure is based on the notion of information entropy. Entropy expresses the average amount of information in bits in some attribute of an instance. Here p is the probability of the occurrence of a value of an attribute (event). The amount of information (expressed in bits) associated with an event of probability p is $-\log_2(p)$. E.g., if the probability of the occurrence of an value is 0.25, then 2 bits are needed to hold this amount of information. If there are many values possible, then to determine the average number of bits, the amount of information for each value must be multiplied with its probability and summed up. This leads us to the final entropy formula $E = -\Sigma_j p_j \log_2(p_j)$ [199].

Suppose that there are k classes $C_1, C_2, \ldots, C_k$, and that of the n instances classified to this node, n_{c_1} belong to class C_1, n_{c_2} belong to class $C_2, \ldots$, and n_{c_k} belong to class C_k, such that $n = \sum_{i=1}^{k} n_{c_i}$. Let $p_1 = n_{c1}/n, p_2 = n_{c2}/n, \ldots$, and $p_k = n_{c_k}/n$. The initial entropy E at this node is: $E = -p_1 \log_2(p_1) - p_2 \log_2(p_2) - \ldots - p_k \log_2(p_k)$.

Now the instances on each value of the chosen attribute A are split. Suppose that there are r attribute values for A, namely $a_1, a_2, \ldots, a_r$. For a particular value a_j, suppose that there are n_{a_j,c_1} instances in class C_1, n_{a_j,c_2} instances in class $C_2, \ldots$, and n_{a_j,c_k} instances in class C_k, for a total of n_{a_j} instances having attribute value a_j. Let $q_{j,1} = n_{a_j,c_1}/n_j$, $q_{j,2} = n_{a_j,c_2}/n_j, \ldots$, and $q_{j,k} = n_{a_j,a_k}/n_j$, where n_j is the total number of instances in the training set having value a_j for attribute A. The entropy E_j associated with this attribute value a_j at this position is: $-q_{j,1}\log_2(q_{j,1}) - q_{j,2}\log_2(q_{j,2}) \ldots - q_{j,k}\log_2(q_{j,k})$. Now compute the information gain for the attribute A: $E - ((n_1/n)E_1 + (n_2/n)E_2 + \cdots + (n_r/n)E_r)$. Note that n_j/n is the estimated probability that an instance classified to this node will have value a_j for attribute A. Thus the entropy estimates E_j is weighted by the estimated probability that an instance has the associated attribute value.

It is the best to depict these deliberations by an example:

Example 6.2. *Classification of meteorological observations* $\triangledown$
Based on the given example set of already reached decisions one has to determine, if the current weather situation is suited for a walk. The examples are structured with the following set of attributes *view*, *temperature*, *air humidity* and if it is *windy* outside. Here, two classes are evident, which is either the recommendations to "take a *walk*" or to "stay at *home*".

Observation	View	Temperature	Humidity	Windy	Recommendation
1	sunny	hot	high	no	home
2	sunny	hot	high	yes	home
3	overcast	hot	high	no	walk
4	rainy	average	high	no	walk
5	rainy	cold	normal	no	walk
6	rainy	cold	normal	yes	home
7	overcast	cold	normal	yes	walk
8	sunny	average	high	no	home
9	sunny	cold	normal	no	walk
10	rainy	average	normal	no	walk
11	sunny	average	normal	yes	walk
12	overcast	average	high	yes	walk
13	overcast	hot	normal	no	walk
14	rainy	average	high	yes	home

Table 6.1: Meteorological observations

In terms of the example, $k = 2$ since there are just two classes, "walk" and "home". The number of instances is $n = 14$. The number of instances in the classes (n_{c_i}) are $n_{neg} = 5$ and $n_{pos} = 9$. Therefore, the overall entropy of the example set here is $E = -p_{neg}\log_2(p_{neg}) - p_{pos}\log_2(p_{pos}) = -(5/14)\log_2(5/14)) - (9/14)\log_2(9/14)) = 0.9403$. In the example, the first attribute considered is *view* and its first value is *sunny*, so $a_j = sunny$. The values for n_{a_j,c_k} are $n_{sunny,neg} = n_{1,1} = 3$ and $n_{sunny,pos} = 2$. Thus $q_{j,1} = q_{1,1} = n_{1,1}/n_1 = 3/5$ and for the "walk" class $q_{1,2} = 2/5$. The attribute value's entropy then is $E_1 = -3/5\log(3/5) - 2/5\log(2/5) = 0.971$. The entropy values for the values *rainy* and *overcast* are $E_2 = 0.971$, resp. $E_3 = 0$. The overall entropy for the information gain of the attribute *View* therefore is $E - ((n_1/n)E_1 + (n_2/n)E_2 + (n_3/n)E_3) = 0.9403 - ((5/14) * E_1 + (5/14) * E_2 + (4/14) * E_3 = 0.2467$. This is higher (*temperature, windy*) or equal (*humidity*) than the information gain of all other attributes on that level (e.g., *temperature*'s information gain is 0.0292).

Based on these calculations the splitting attribute *view* is chosen and the first level subsets of the decision tree of figure 6.4 on page 130 is built. All other levels are built in the same way. △

For a continuous attribute the best value for splitting the set of instances is chosen in an equal manner.

6.3.2 Using the tree for classification

In general, the decision tree developed in that way is used for classifying new examples. A newly added instance is handled by starting at the root node and determining, if the condition expressed on the node's attribute evaluates to either true or false with the observation's attribute-value. The result of the evaluation directs to the next node. After reaching a leave node, the observation is an element of the class named by the leaf label.

In SBS this phase is called the *browsing* phase. How this is performed is discussed in detail in the following section.

6.4 Integrating decision trees into SBS

In considering the SBS domain as described by attribute-value pairs, it is well suited for machine learning. We adopted the decision tree algorithm C5.0 from Quinlan [192] for performing this task. The C5.0 is an advancement of the ID3 (and its successor C4.5). A free evaluation version is available[4]. In SBS we utilized the demo version C5.0 release 1.14 (for the Solaris operating system). After a partition of the SBS-repository is set up correctly, it holds the components and the data points in the structure already presented in figure 6.3 on page 125. On that basis C5.0 generates a decision tree as browsing aid for the searcher.

6.4.1 The C5.0 algorithm for SBS analysis

Specificities of the SBS application can be considered by parameterizing C5.0, too. The algorithm accepts a variety of options for adjustment. The following aspects are important in the SBS domain:

1 C5.0 constructs decision trees in two phases. A tree is first grown to fit the data closely and is then 'pruned' by removing parts that are predicted to have a relatively high error rate (thus, accepting uncertainty in the final classification). An error occurs, if an observation can be classified to more than one class. Misclassification can be due to wrong initial judgment of a membership of an instance to a class or wrong attribute values. The option's value for pruning confidence `-c` *CF* affects the way that error rates are estimated and hence the severity of pruning. Values smaller than the default (25%) cause more of the initial tree to be pruned, while larger values result in less pruning [189].

   For SBS it is assumed that data points are assigned correctly to their components during the generation phase of a partition. Therefore, we do not expect noise in the form of misclassifications and the pruning confidence for C5.0 is set to 100 percent; `-c 100`.

2 Normally, pruning is not performed to evade noise only, but to simplify the final tree as well. Since we do not use the resulting decision tree for the task of classifying components, but to generate a conceptual description of them, we need the full blown decision tree. This is an additional argument for setting the pruning confidence to 100 percent.

3 Due to the data point's properties *necessity* and *descriptor uniqueness*, we have exactly one observation per class (the component specification's name, its set of input vectors and its set of output vectors). C5.0's option `-m` *CASES* enables the specification of a minimum number of observations which have to be evident in the set of data points supporting the classification. The default value

[4]Demo versions are available for most common operation systems in the down load area of `http://www.rulequest.com/`

is two cases. However, in SBS there is only one observation per class. Thus, the minimum number of cases supporting the tree building process is set to one; `-m 1`. Thereby, the tree is constructed utilizing each observed example.

C5.0 offers some further useful properties. It is possible to specify missing values (thus making the data noisy) or to determine, whether a certain value for an attribute is not applicable for computing a decision. Due to the demanded initial completeness of data points these features are not used. Only in the maintenance phase of the repository this renders practical.

How the construction of decision trees leads to a browsing structure is demonstrated in the next two examples. They differ mainly in their signature; the first signature is numerical based and continuous, the second one deals with discrete string manipulation.

Example 6.3. *A* `number → number` *partition* ▽

In this example 32 mathematical functions implemented in Modula-3 are analyzed. Modula-3 has a rigid typing system, and the partition design benefits therefore from a generalized signature in such a way that the number of components which can populate a partition becomes rather high. The functions can be divided into three groups:

- The majority of the functions are taken from the package `Math`. They require a value of type `LONGREAL` for input and compute a result of type `LONGREAL`. There are 29 of them in the partition.
- Two functions from the package `Pts` of type `REAL → REAL`.
- One function from package `Swap` of type `INTEGER → INTEGER`.

For these components the generalized signature `number→number` is chosen. Please note, that this is not a subtype generalization. The type variable `number` is a wildcard standing for any subrange type holding numerical values. The partition holds all constitutive equal components conforming to that signature specification. For the sake of demonstration, no distinction between implementations and the components (component specifications) is made.

In table 6.2 a list of all Modula-3 functions of the partition and a short description of their semantics is given. Suppose the input parameter is always named "x".

Table 6.2: Components of the partition $P_{\texttt{number}\rightarrow\texttt{number}}$

1	Math.exp	E^x
2	Math.expm1	$(E^x) - 1$, even for small x
3	Math.log	the natural logarithm of x (base E)
4	Math.log10	the base 10 logarithm of x
5	Math.log1p	$\log(1 + x)$, even for small x
6	Math.sqrt	$\sqrt{x}$
7	Math.cos	the trigonometrical cosine of x
8	Math.sin	the trigonometrical sine of x
9	Math.tan	the trigonometrical tangent of x
10	Math.acos	the inverse to cosine, arccosine of x
11	Math.asin	the inverse to sine, arcsine of X
12	Math.atan	the arctangent of x
13	Math.sinh	the sine hyperbolic of x
14	Math.cosh	the cosine hyperbolic of x
15	Math.tanh	the tangent hyperbolic of x
16	Math.asinh	the arcsine hyperbolic of x
17	Math.acosh	the arccosine hyperbolic of x
18	Math.atanh	the arctangent hyperbolic of x
19	Math.ceil	the least integer not less than x
20	Math.floor	the greatest integer not greater than x
21	Math.rint	the nearest integer value to x
22	Math.fabs	the absolute value of x
23	Math.erf	the ”error” function of x
24	Math.erfc	$1.0 - erf(x)$, even for large x
25	Math.gamma	$\log(\lvert\texttt{gamma}(\lvert x\rvert)\rvert)$
26	Math.j0	the zero-order Bessel function of first kind
27	Math.j1	the first-order Bessel function of first kind
28	Math.y0	the zero-order Bessel function of second kind
29	Math.y1	the first-order Bessel function of second kind
30	Swap.SwapInt	Swaps all of the bytes of x
31	Pts.FromMM	Convert from millimeters to points
32	Pts.ToMM	Convert from points to millimeters

The functions were tested carefully and we chose initially 164 input values to generate data points. In this example, the majority of the test cases are randomly generated. The analysis of the test cases showed, that for the partition the property of necessity holds (there are no two input vectors generating identical output vector sets) and the test cases have descriptor uniqueness (there do not exist two output sets of different components which are identical). Additionally, some data points are generated for the sake of striking samples. Since this domain deals mainly with rounding, exponential and trigonometric functions, values like π, $\frac{\Pi}{2}$, 0, 1 are considered as striking for domain experts. The results of input values from $1, \ldots, 9$ are in most cases immediately recognizable, too. However, there are no general rules for obtaining such values, as there are no such general rules known in the field of domain boundary testing as well.

Table 6.3: Data points of the partition $P_{\texttt{number}\rightarrow\texttt{number}}$

Input	Math.exp	Math.expm1	Math.log	Math.log10	Math.log1p	Math.sqrt
0	1	0	1.80E+308	1.80E+308	0	0
1	2.72	1.72	0	0	0.69	1
2	7.39	6.39	0.69	0.3	1.1	1.41
3	20.09	19.09	1.1	0.48	1.39	1.73
4	54.6	53.6	1.39	0.6	1.61	2
5	148.41	147.41	1.61	0.7	1.79	2.24
6	403.43	402.43	1.79	0.78	1.95	2.45
7	1096.63	1095.63	1.95	0.85	2.08	2.65
8	2980.96	2979.96	2.08	0.9	2.2	2.83
9	8103.08	8102.08	2.2	0.95	2.3	3
0.5	1.65	0.65	−0.69	−0.3	0.41	0.71
3.14	23.14	22.14	1.14	0.5	1.42	1.77
1.57	4.81	3.81	0.45	0.2	0.94	1.25
0.79	2.19	1.19	−0.24	−0.1	0.58	0.89
−3.14	0.04	−0.96	1.80E+308	1.80E+308	1.80E+308	1.80E+308
−1.57	0.21	−0.79	1.80E+308	1.80E+308	1.80E+308	1.80E+308
−0.79	0.46	−0.54	1.80E+308	1.80E+308	−1.54	1.80E+308

Input	Math.cos	Math.sin	Math.tan	Math.acos	Math.asin	Math.atan
0	1	0	0	1.57	0	0
1	0.54	0.84	1.56	0	1.57	0.79
2	−0.42	0.91	−2.19	0	0	1.11
3	−0.99	0.14	−0.14	0	0	1.25
4	−0.65	−0.76	1.16	0	0	1.33
5	0.28	−0.96	−3.38	0	0	1.37
6	0.96	−0.28	−0.29	0	0	1.41
7	0.75	0.66	0.87	0	0	1.43
8	−0.15	0.99	−6.8	0	0	1.45
9	−0.91	0.41	−0.45	0	0	1.46
0.5	0.88	0.48	0.55	1.05	0.52	0.46
3.14	−1	0	0	0	0	1.26
1.57	0	1	−2.86E+014	0	0	1
0.79	0.71	0.71	1	0.67	0.9	0.67
−3.14	−1	0	0	0	0	−1.26
−1.57	0	−1	2.86E+014	0	0	−1
−0.79	0.71	−0.71	−1	2.47	−0.9	−0.67

Input	Math.sinh	Math.cosh	Math.tanh	Math.asinh	Math.acosh	Math.atanh
0	0	1	0	0	1.80E+308	0
1	1.18	1.54	0.76	0.88	0	1.80E+308
2	3.63	3.76	0.96	1.44	1.32	1.80E+308
3	10.02	10.07	1	1.82	1.76	1.80E+308
4	27.29	27.31	1	2.09	2.06	1.80E+308
5	74.2	74.21	1	2.31	2.29	1.80E+308
6	201.71	201.72	1	2.49	2.48	1.80E+308
7	548.32	548.32	1	2.64	2.63	1.80E+308
8	1490.48	1490.48	1	2.78	2.77	1.80E+308
9	4051.54	4051.54	1	2.89	2.89	1.80E+308
0.5	0.52	1.13	0.46	0.48	1.80E+308	0.55

Continued on next page

Continued from previous page

3.14	11.55	11.59	1	1.86	1.81	1.80E+308
1.57	2.3	2.51	0.92	1.23	1.02	1.80E+308
0.79	0.87	1.32	0.66	0.72	1.80E+308	1.06
−3.14	−11.55	11.59	−1	−1.86	1.80E+308	1.80E+308
−1.57	−2.3	2.51	−0.92	−1.23	1.80E+308	1.80E+308
−0.79	−0.87	1.32	−0.66	−0.72	1.80E+308	−1.06

Input	Math.ceil	Math.floor	Math.rint	Math.fabs	Math.erf	Math.erfc
0	0	0	0	0	0	1
1	1	1	1	1	0.84	0.16
2	2	2	2	2	1	0
3	3	3	3	3	1	0
4	4	4	4	4	1	0
5	5	5	5	5	1	0
6	6	6	6	6	1	0
7	7	7	7	7	1	0
8	8	8	8	8	1	0
9	9	9	9	9	1	0
0.5	1	0	0	0.5	0.52	0.48
3.14	4	3	3	3.14	1	0
1.57	2	1	2	1.57	0.97	0.03
0.79	1	0	1	0.79	0.73	0.27
−3.14	−3	−4	−3	3.14	−1	2
−1.57	−1	−2	−2	1.57	−0.97	1.97
−0.79	0	−1	−1	0.79	−0.73	1.73

Input	Math.j0	Math.j1	Math.y0	Math.y1	Math.gamma	Swap.SwapInt
0	1	0	1.80E+308	1.80E+308	1.80E+308	0
1	0.77	0.44	0.09	−0.78	0	16777216
2	0.22	0.58	0.51	−0.11	0	33554432
3	−0.26	0.34	0.38	0.32	0.69	50331648
4	−0.4	−0.07	−0.02	0.4	1.79	67108864
5	−0.18	−0.33	−0.31	0.15	3.18	83886080
6	0.15	−0.28	−0.29	−0.18	4.79	100663296
7	0.3	0	−0.03	−0.3	6.58	117440512
8	0.17	0.23	0.22	−0.16	8.53	134217728
9	−0.09	0.25	0.25	0.1	10.6	150994944
0.5	0.94	0.24	−0.44	−1.47	0.57	16777216
3.14	−0.3	0.28	0.33	0.36	0.83	50331648
1.57	0.47	0.57	0.41	−0.37	−0.12	33554432
0.79	0.85	0.36	−0.1	−0.99	0.17	16777216
−3.14	−0.3	−0.28	1.80E+308	1.80E+308	0.02	−33554433
−1.57	0.47	−0.57	1.80E+308	1.80E+308	0.83	−16777217
−0.79	0.85	−0.36	1.80E+308	1.80E+308	1.69	−1

Input	Pts.FromMM	Pts.ToMM
0	0	0
1	2.83	0.35
2	1.42	0.18
3	8.91	1.11
4	4.45	0.55
5	2.23	0.28
6	−8.91	−1.11
7	−4.45	−0.55
8	−2.23	−0.28
9	2.83E+12	3.53E+11
0.5	2.83E+12	3.53E+11
3.14	−2.83E+12	−3.53E+11
1.57	0	0
0.79	0	0
−3.14	−1.78	−0.22
−1.57	63810204	7941317.5
−0.79	−20589722	−2562435.2

For the analysis step the data points are rounded to the sixth decimal place. This is sufficiently exact, since generally, humans are not capable to judge the effect of higher precision. Out of these 164 data points we chose 17 data points per component to appear as classifiers in the resulting decision tree. This number of data points was the result of an experiment where the number of data points has been reduced continuously until their characterizing power was no longer suf-

```
0 <= 0:
:...-1.5707963267949 <= -0.9171523:
    :...1 <= 0.8813736:
    :   :...0.785398163397448 <= 0.6674572:
    :   :   :...1 <= 0.7651977: Math.tanh (1)
    :   :   :   1 > 0.7651977: Math.atan (1)
    :   :   0.785398163397448 > 0.6674572:
    :   :   :...-1.5707963267949 <= -1.233403: Math.asinh (1)
    :   :       -1.5707963267949 > -1.233403:
    :   :       :...1 <= 0.841471: Math.sin (1)
    :   :           1 > 0.841471: Math.erf (1)
    :   1 > 0.8813736:
    :   :...1 > 1:
    :       :...1 <= 2.834646: Math.sinh (1)
    :       :   1 > 2.834646: Swap.SwapInt (1)
    :       1 <= 1:
    :       :...0.5 > 0.5: Math.ceil (1)
    :           0.5 <= 0.5:
    :           :...1.5707963267949 <= 1.253314: Math.floor (1)
    :               1.5707963267949 > 1.253314: Math.rint (1)
    -1.5707963267949 > -0.9171523:
    :...-3.14159265358979 <= 0:
        :...1 <= 1:
        :   :...1 <= 0.3527778: Pts.ToMM (1)
        :   :   1 > 0.3527778: Math.j1 (1)
        :   1 > 1:
        :   :...3.14159265358979 <= -2.834646e+12: Pts.FromMM (1)
        :       3.14159265358979 > -2.834646e+12:
        :       :...1 <= 1.570796: Math.asin (1)
        :           1 > 1.570796: Math.expm1 (1)
        -3.14159265358979 > 0:
        :...-3.14159265358979 <= 3.141593:
            :...1 <= 1.175201: Math.fabs (1)
            :   1 > 1.175201: Math.tan (1)
            -3.14159265358979 > 3.141593:
            :...0.785398163397448 <= 0.7212255: Math.log1p (1)
                0.785398163397448 > 0.7212255:
                :...1 <= 1: Math.sqrt (1)
                    1 > 1: Math.atanh (1)
```

```
0 > 0:
:...0 <= 1.570796:
    :...0 > 1: Math.acos (1)
    :   0 <= 1:
    :   :...1 <= 0.5403023:
    :       :...1 <= 0.1572992: Math.erfc (1)
    :       :   1 > 0.1572992: Math.cos (1)
    :       1 > 0.5403023:
    :       :...3 <= 3: Math.j0 (1)
    :           3 > 3:
    :           :...1 <= 1.718282: Math.cosh (1)
    :               1 > 1.718282: Math.exp (1)
    0 > 1.570796:
    :...-3.14159265358979 <= 0.01557522: Math.gamma (1)
        -3.14159265358979 > 0.01557522:
        :...1 <= -0.7812128: Math.y1 (1)
            1 > -0.7812128:
            :...1 > 0: Math.y0 (1)
                1 <= 0:
                :...9 <= 1.460139: Math.log10 (1)
                    9 > 1.460139:
                    :...2 <= 0.9953223: Math.log (1)
                        2 > 0.9953223: Math.acosh (1)
```

Figure 6.5: The resulting C5.0 decision tree (raw output)

ficient to distinguish between components. Most of these data points are striking samples. With C5.0 one can select explicitly attributes (input vectors in SBS) to support the design of the tree and, thus, raising its understandability. In table 6.3 the whole set of data points for classifying the partition $P_{\texttt{number}\rightarrow\texttt{number}}$ is shown (not all decimal places are given).

Figure 6.5 shows the resulting tree as unformatted output as it is generated by the call to C5.0: `>c5 -f n2n -m 1 -c 100`. The main part of the tree's left branch is given in a more readable form in figure 6.6 on the following page. The knowledge about the data points is stored in the files `n2n.names` (defining the attributes) and `n2n.data` (holding the test cases). With 17 data points per component only, C5.0 was able to classify all components correctly.

The browsing tree of the `number`→`number` is organized in a way that at each decision node the predictive attribute (the input to the components) is given. On each outgoing edge, the conditions are stated, which must hold for all components reachable beneath this node in the tree.

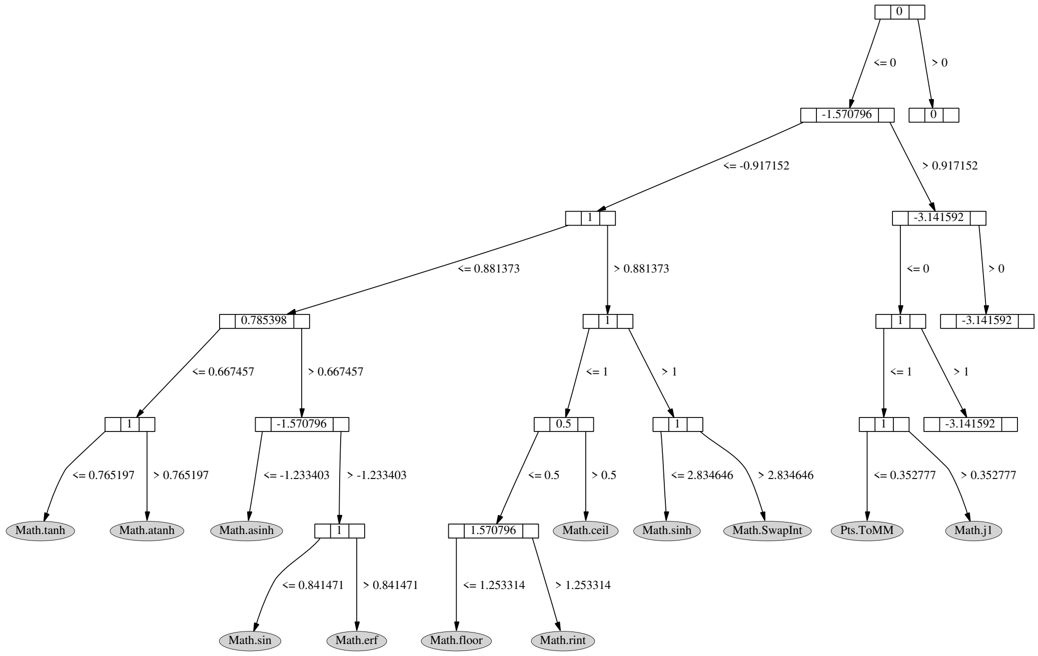


Figure 6.6: A part of the resulting decision tree (transformed from figure 6.5)

The input vector with the highest information gain is '0'. The searcher now decides, if the searched functionality provided with input '0', computes an output equal or smaller than '0' (given in the raw output as `0 <= 0` in the tree). Depending on the answer, the searcher is directed to the next level.

E.g. if a function f is searched which computes the next larger integer of its input (if it isn't already an integer), since $f(0) = 0$, this branch of the tree has to be followed. The next question is presented as: "Should $f(-1.570796) \leq -0.917152$?" Clearly, the next larger integer is -1. Thus, the searcher must follow the true-branch of the tree. After three more questions and correct guesses as answers, the function `Math.ceil` is the only candidate remaining. The Modula-3 function `ceil` calculates indeed the next larger integer of its input.

C5.0 comes with a set of tools as well. Most of them are used for optimizing the classification, based on cost information or cross validation of the classification concept with validation evidence. In the context of SBS the tool `predict` is a good supplement to the browsing structure. On the basis of the classification tree, it offers a query-answer interface to the knowledge base. This comes very near to an interactive search mechanism, where the searcher (after selecting the partition) answers to some queries and the tool determines the best fitting component and judges the certainty of its classification. Here is an example dialogue, where input values are presented and the searcher enters her/his estimates of the desired output after the colon (the answers are given in bold type face).

```
0: 0
-1.5707963267949: -1
1: 1
0.5: 1

    -> Math.ceil [0.06]
```

In the case an answer to a query cannot be provided, the `predict` tool accepts missing values (`?`) or the hint that an attribute's value is not applicable in a specific context (`N/A`). Although this leads eventually to an inaccurate result, such a flexibility is very helpful, if the searcher wants to take a quick look at the content of a partition.

△

The signature of the partition of the next short example are based on a complex input vector containing two elements and a return a boolean.

Example 6.4. *A `string` → `string` partition* ▽
The partition $P_{\texttt{string} \rightarrow \texttt{string}}$ contains 8 predicates implemented as C-functions. They are taken from the ANSI-C standard library [188]. The input vectors are grouped by angle brackets and their elements are separated by a semi-colon. Due to the small number of components and the preconditions *necessity* and *descriptor uniqueness* the set of test cases which is sufficient to describe them is very small. We need only the same number of data points as there are components. The components and their data points are shown in figure 6.7 on the next page.

Since the output types are of discrete nature, this information can be given via the flag `-s` to C5.0 to improve the result of the building process. Figure 6.8 on page 145 presents the formatted result of the execution of `c5 -f string -m 1 -c 100 -s`. △

Input	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

Figure 6.7: A string predicate partition

The benefits of the approach are demonstrated by both examples. Nevertheless, SBS is based on partial specifications and therefore, if a candidate component is presented as final result of the search some further investigation must be performed. This is indeed no strong demand, since the partition is constructed based on the descriptive capability of its data points and therefore, the resulting candidates are already classified according to their structure (signature) and their behavior. Only one component is left and its documentation must be studied anyway. If it emerges, that the functionality searched for is not provided with the resulting component, at least we know that the functionality as specified by the attribute-value pairs collected on the path from the decision tree's root to the leaf node, does not occur anywhere else in the current partition.

6.4.2 SBS-partition maintenance

Due to the initial completeness, all input vectors are input to all components in the library and as a consequence the C5.0 algorithm is able to determine an error-free classification. Necessity and descriptor uniqueness ensure that data points describe all components of a partition. But repositories evolve over time: further components are added and components which do not render valuable, are retracted from the repository. Such changes affect the quality of data points and have to be considered.

Deletion of a component, together with its data points does not demand a re-analysis of the partition, since the properties necessity and descriptor uniqueness are not touched [181].

If a component is added, then such a stable situation is not given. The question of how to obtain the data points for description purpose can be answered by

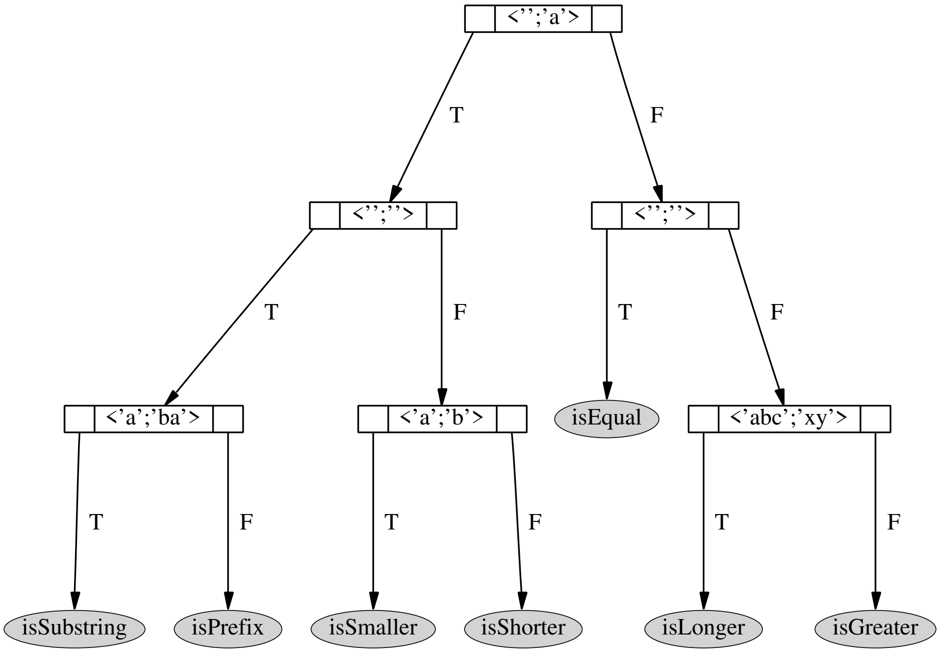


Figure 6.8: The final browsing structure for the string predicates

looking at the component in isolation. However, it is no easy task, to discriminate between the new descriptors and the existing one. Ideally, to maintain initial completeness [181],

1 the components already classified should be executed on the input vectors of the added components, and
2 the added components should be executed with all input vectors of the partition.

These challenges are shown in figure 6.9 on the next page. When new components are added, the previously established data point quality must be retained. But for an evolving repository retaining automatically an established grade of quality is rather illusionary and alternatives have to be investigated. We suggest the following strategies for doing SBS maintenance:

1 The new component is added to the repository regardless of missing input values (missing attributes can be handled by C5.0) and the decision tree is rebuilt at each change. The advantage is a quick integration of new components. It is

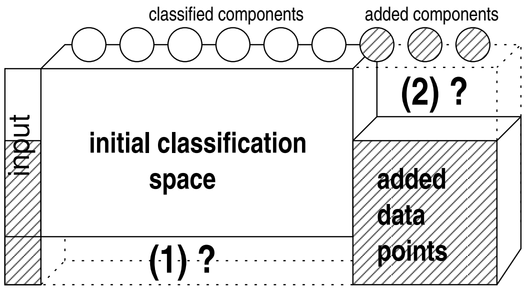


Figure 6.9: The completeness problem in SBS maintenance

likely, that some of its input vectors are chosen as striking samples and therefore an overlap of input vectors exist. Due to the previous quality of the partition‘s data points, the new component can be classified without errors. This strategy is successful for adding some few new components, but with each addition the quality of data points is reduced and, continually, the classification becomes more and more error-prone.

2 The component is integrated immediately and two processing steps are triggered:

  i The new component is executed on each missing input vector of the partition, and together with the output vectors generated in that way they form data points.

  ii All components of the partition are executed on the input vectors of the new component to fill up their missing descriptors.

The second point is not mandatory in every context. If the data points which are generated in the first step are added to the partition and necessity as well as descriptor uniqueness are already obtained, no further data points have to be generated. The goal of this strategy is to maintain the property of *initial completeness*.

The resulting quality of the decision tree is equal to that of the tree generated for the initial partition. The disadvantage is, that executing a component, either the new one or the old ones, is very costly, which we already criticized as a disadvantage of the approach of Podgurski and Pierce [175]. Sometimes it is not feasible at all, because either

- (in the case of the lack of a repository’s execution environments) the new component was provided externally and additional test cases can not be acquired, or

- the components demonstrate different concrete signatures and the execution environment cannot conform to all the particularities, from which the signatures are abstracted.

3 A new component is not added to the repository immediately, but kept separately. If a searcher does not find a functionality in the partition, this add-on list is presented. The presentation in the list is flat, since no relations are established between its components.

The idea of this strategy is to minimize the re-execution effort and only if the list exceeds a certain threshold, its components are integrated into the partition in the way suggested as strategy 2. The advantage is that only periodically re-indexing takes place and a short add-on list of components does not lengthen the search time substantially. As a further disadvantage it turns out to be, that if a not-yet-integrated component is very frequently searched for, then the tree structure does not help to speed up the search.

4 This strategy is based on the assumption that newly added components are easier to execute than components classified long times ago. This is motivated by the fact that new components are currently under test, or were in test a short time ago and their execution environment is still available. As a consequence, for new components missing data points can be generated on demand (this is the partition's area in figure 6.9 on the facing page marked with (2)). Necessity and descriptor uniqueness are not maintained and "old" components are not re-executed to generate necessary data points (area (1) of figure 6.9).

If after a component's addition the partition qualities of necessity and descriptor uniqueness are not as high as required, the partition is divided into subpartition. For each newly generated subpartition necessity and descriptor uniqueness have to hold and, additionally, decision trees must be generated as browsing aids for them. With this strategy, many browse trees exist for a partition in parallel. There is no guidance for the searcher which tree is to be selected and normally, this hampers effective browsing. However, if the sub-partitioning process is not based on the data points only but takes into consideration domain related information as well, multiple decision trees emerge to be an advantage. E.g. a subpartition may contain trigonometric functions and an other one the rounding functions. Then a keyword based decision tree selection, although keywords have to be interpreted, helps to locate the sought component best.

In the case that data points are delivered externally and cannot be generated on demand, this strategy is superior to all other ones. On the other hand, concurrent browsing trees without a clear semantics, do not support the searcher. Nevertheless, due to the mechanism of C5.0, the decision trees in general are not exorbitantly deep and searching for a certain functionality is easier than to scan sequentially through a list of textual descriptions of components.

Additionally, a periodical reunification of the browsing trees, (if execution of components is feasible) is recommended.

Which strategy to choose depends on a variety of parameters. If for instance, automatic execution is feasible, then the best option to choose is to maintain completeness (strategy 2), whereas for a very populated repository this strategy is an illusionary task. On the other hand, catchy textual descriptors support the selection of the correct browsing structure in the concurrent tree approach (strategy 4). However, this strategy rests on a clear semantics of the descriptive texts. Nevertheless, the entirety of these strategies gives some options to the librarian, who is responsible to take care of the discriminating and descriptive power of the repository‘s data points to yield useful browsing structures [181].

6.5 Final remarks to SBS

Static behavior sampling (SBS) renders a practicable method to describe the behavior of reusable components without the need for human interpretation. There are some preconditions which must hold, when applying SBS. The first preconditions is that components must be stateless. The result of a component's computation depends only on its input data. A further preconditions states, that the components´ signatures have to be abstracted to generalized signatures, which helps to group together components with different interface realizations, although they have a common concept. And the third important precondition is the necessary quality of data points which serve as partial specifications. This implies, that some flexibility in data point generation is necessary. If the data points are not producible on demand during the SBS structuring phase (be it initially or be it during maintenance), this data must be delivered by the domain engineers providing the component. However, as already discussed in the previous section, the number of high quality data points necessary to generate a good browsing structure is not very large. Therefore, when the set of test cases coming with the components is numerous enough (a requirement easy to fulfill), to obtain data points is not a hindrance for applying SBS.

6.6 Summary

In this chapter we presented the concept of *static behavior sampling* (SBS). The approach originates from *data point* descriptions which are considered as partial specifications of reusable components. Data points are chosen from test data tuples and they are divided into characterising tuples and striking samples, depending on their purpose: either to support the indexing mechanism or to support the human searcher. It is also discussed, how to select data points from the set of data tuples for each category. Data points then form the knowledge base for the SBS repository. It is described, how the repository is structured into partitions according to the signatures of components. The partitions hold components (resp. component specifications) characterized by data points. To generate a browsing structure within a partition, a decision tree algorithm is used. We demonstrated the benefits of this approach with two examples from different domains. Furthermore, the problem of repository maintenance is discussed and four different strategies to tackle it are presented.

The contribution of this chapter is the novel approach to consider data points as attribute-value descriptors and the way how decision tree structures can be used as a repository browsing structure. The searcher is supported by presenting behavior which helps to chose the most promising search path. In that way, the content of the whole partition can be explored and without having to perform costly component executions, an intuitive component representation is given.

In the next chapter, the idea of SBS is extended towards the treatment of state bearing and complex components. Obviously, in that context the pure input-output information is not sufficient to provide descriptive quality, since besides the actual input, the history of previous inputs to a component determines its next outcome, too. We discuss two techniques for detecting regularities in component histories and present improvements of them to utilize them for producing meaningful descriptions automatically.

7

Behavioral Description of Complex and State-Bearing Components

The approach of static behavior sampling (SBS) characterizes reusable components by descriptions built from examples. It demonstrates many benefits when dealing with state-less components. But the question arises, whether this idea is also applicable when facing a repository filled with components (1) whose behavior depends on their execution history and/or (2) whose signature is an abstraction of a multitude of functional types with different behaviors. In the first case, observing the immediately produced output does not lead to success, since an identical input value for one particular component might not always produce the same output. In the second case, an alternative component characterization must be available, since the interface is too complex to be easily understandable.

Nevertheless, the central idea to use test data as primary source of knowledge is appealing and remains to be the main driver of the approach. There are two main assumptions which must hold when dealing with state-bearing and/or complex components:

1. A significant number of test cases, which serve as a knowledge base, has to be available, and
2. The components must not contain faults which would blur the descriptive power of the available test cases.

The following particularities make the difference to state-less components with respect to SBS analysis:

State: Components are state-bearing, if some or all of their internal data values are kept between calls to them or if such values are persistent. Persistence is the property of data to be available beyond the execution time of the software system maintaining it. Due to this internal state, the input to a component is not the only determinant which constitutes the output. The current state of a component is not visible from outside. Nevertheless, in our approach it is not feasible to "open" a component for observing internal values. Only by analyzing the data processed by the component, can conclusions be drawn about its exact behavior.

An important representative of state-bearing systems is object-oriented software. Nevertheless, any component retaining its internal data space is state bearing, regardless of its development paradigm. States can be maintained by storing values on files or by using global memory.

Complex interface: Components maintaining their internal states tend to be more complex than state-less components. They offer a broader range of behavior than the "single spot" solutions generated by state-less components. The more requirements a component covers in a coherent way, the more it is useful for an application engineer. One might think of a module (like the one presented in section 5.1.2 on page 72), where the application engineer is not primarily interested in one specific function, but in the whole tool set offered by the module. In such situations, browsing through the SBS classification in order to verify the purpose of each function is very cumbersome. In fact, an application engineer would accept many variants of behavior, if only the overall goal is fulfilled. Here, the whole module is considered a component and not a particular function provided by it. SBS does not help substantially in this situation. Whether there is an internal state (for functions or for the module) to be considered is not important, but a description technique based on ideas similar to SBS is needed to describe *complex interfaces*.

Protocol: A component, which is accessible through a complex interface, is often not designed and developed by the application engineer but by a domain engineer external to the application development process. Hence, the fine grained solutions are not designed "in-house" , because they are already implied in the components. Most of the detailed work has already been done by the domain engineers who built the components. In such a situation the main task of the application engineer is to integrate the artifact into the system. Very often, the system's fine design is then changed to ease the task of component incorporation. Incorporation requirements stem from the component's specific architectural prerequisites [81].

Complex components need more context in a system than state-less ones. Such contexts may be provided as standards for file system access or for communicating with external systems [220].

Components with a rich set of behavior perform complex tasks. Such tasks are built on activities which depend on each other and therefore have to be executed in a certain order. Activities dependent causally on each other and cannot be invoked without knowing the sequence specification for them [117].

Many of these prerequisites and context dependencies have to be established as specific sequences of activations of the components´ functionality. This leads to an (internal) *protocol* which must be followed when communicating with a component.

All issues raised above need a different way of describing reusable software than the one SBS offers. Whereas in SBS the focus is on analyzing data transformations of a component with a singular functionality, this focus is not applicable here. The test data

- cannot be partitioned reasonably according to signatures. Due to the multitude of the functional signatures of a component[1] test tuples of one test case may conform to different signatures.
- does not reflect the input-output transformation deterministically. This is due to the output's dependency on the component's internal state.
- holds important information about the order of the activated functionalities. This order is not exploited by SBS.

We therefore focus on the important aspect of sequences of test data. For complex components, most invocations of functionalities have to follow a certain order. This order is important and if it is described well, a searcher can infer the overall purpose of a component from that.

Another important aspect is the change of the purpose of the retrieval task. For simple components it is necessary that relevant assets are separated from the vast amount of available components. This is not that important for complex

[1] Obviously, the component (module, package, or class) has a single signature. But its functions, procedures and methods cannot be browsed coherently by one single decision tree, since one tree conforms to one functional signature.

components! Here, the number of components covering the needed functionalities in general is not large. Thus the question is: Is the component really fulfilling the need as its description is claiming? Maybe the description is trustworthy, but not all documents in the necessary technical detail are available.

To verify the behavior of a component is a challenge, especially, when they are bought as commercials-of-the-shelf (COTS). Then, the buyer has no access to the source code [16] and a reengineering activity aimed at getting confidence by validating internal structures is prohibited in most cases. The problem is already addressed in the literature and there is an ongoing discussion of how to solve it [193, 30, 84].

Therefore, the techniques presented in this chapter have two aims:

1. Supporting component retrieval by allowing the searcher to judge the relevance of a behavior representation, and
2. supporting verification, whether the candidate component performs in the promised way.

This chapter deals with the analysis of complex state-bearing components. For this purpose we discuss algorithms used for inferring knowledge from test data. To adapt them for our needs, we implemented some enhancements. In contrast to SBS the input-output is not directly observable, but an abstraction from it is. Hence, we call the adapted algorithms *abstracted behavior sampling*, ABS for short. This abstraction does not contain any concrete values any more. Values are omitted and only the order of method invocation (without parameters) is reflected abstractly. If this kind of description is not sufficient for determining the behavior of a component, it is always possible to take a focused look at the specific value transformations.

As a representative for the class of state-bearing and/or complex components we discuss object-oriented components. Those facets of object orientation which are necessary for complex and state-bearing components are discussed shortly in the following section.

7.1 Object Oriented Components

Nowadays, the object-oriented (OO) paradigm is a state-of-the-art methodology to build software systems. Because of the paradigms encourage to design modularized and encapsulated units, classes and OO-packages can be more easily reused [183, p13],[17]. Most of the components available on the market today are developed with and for object-oriented systems.

7.1.1 Structural aspects

In general, object-orientation means, that software systems are organized as a collection of discrete *objects* that incorporate data, data structures and behavior [200]. The data enclosed in an object is called its *attributes*, the behavior can be activated by calling the object's *methods*. Ideally, the data forms the internal hidden state, which cannot be accessed directly from outside the object. Access is possible only in interrogating the object. The overall system behavior is formed by the interaction of its objects. Interaction is the communication between objects performed as a call of an object o's methods by an object p (*message passing*). In this way, the object's data structure and the method operating on it are strongly related. This is in contrast to conventional (procedural) programming in which data structures and functions operating on them are only loosely coupled [200]. The difference between procedural system development and object oriented system development is evident, when considering how the functionality is offered. In the former, values conforming to the component's signature are transformed, which sets the focus on the signature aspect. In the latter, the legal sequence of methods calls becomes much more important. It is sure enough that without the value transformation the object invocation is meaningless, however, without considering the method invocation sequence a treatment of single values is worthless.

The intention of the OO-paradigm is to have a system that is built upon independent agents which are responsible for a certain aspect of the system's behavior. In that way, system analysis, design, and implementation should be eased, because objects are well defined and separated. Ideally, they can be designed, analyzed, and maintained in isolation.

An important concept is the notion of a *class*. Classes are the result of an abstractions process, called classification, and group together objects with the same data structures and behavior [200]. They are discussed mainly from two different vantage points in the literature, an intensional and an extensional one. For short, the extensional view focuses on the property of classification, whereas the intentional deals with the generation of objects. In OO-programming, classes serve four different purposes [26];

1 They define templates from which objects are instantiated (intensional).

2. They are sets the elements of which are objects (extensional). A class used for this purpose is called object-warehouse, too.
3. They are a means to group together coherent behavior (intensional), which is similar to modules in conventional programming.
4. They are mechanisms to generate objects (object factories) (intensional).

The general structure of a class is depicted in figure 7.1. It is given in the graphical notion as defined by the industry standard UML (universal modeling language). UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of an object-oriented software system [226].

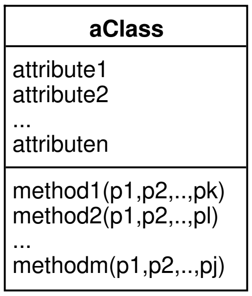


Figure 7.1: The general structure of a class

With respect to SBS the difference between a component built as object and a component built as function (procedure) is self-evident. Methods cannot be equated with functions, since method calls make sense only if used in tight interaction with the object's other methods and its attributes. State-less components are independent from each other.

7.1.2 Behavioral aspects

Behavior of a class can be specified in UML in various ways. Most common, for specifying a class capable of dynamic behavior *statecharts* are used. Statecharts are an extension to finite state automata [103]. Their main construction elements are states and events. A *state* is a condition on attributes during the life of an object of the specified class or an interaction during which it satisfies some condition, performs some action, or waits for some event. Conceptually, an object remains in a state for an interval of time [226]. There are two special states, the initial one and the final one, which depict the begin and end of an object's life cycle.

An *event* is an occurrence that may trigger a state transition. A transition is a relationship between two states indicating that an object in the first state will enter the second state. On such a change of state, the transition is said to “fire”. The event may have parameters [226]. Each event which is understandable by the addressed object conforms to one of its methods. Events may be issued by any object which knows the interface of the addressed one.

A statechart specifies, how an object must react on the receipt of an event instance, depending on its current internal state. For a more complete and in-depth discussion of modeling and designing object-oriented systems the interested reader is refered to an introduction to the object modeling technique OMT, e.g. [200], or to the introductory UML literature, e.g. [99, 71].

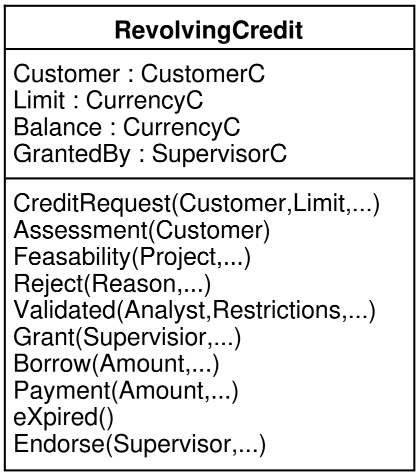
7.1.3 An example from the banking domain

The following example explains the discussed issues by means of a class from the banking domain. It is chosen, because the class demonstrates sufficiently complex behavior without providing too much details, and the behavior is generalizable to render a reusable component, which is due to its “preparation”-“processing”-“finalization” sequence.

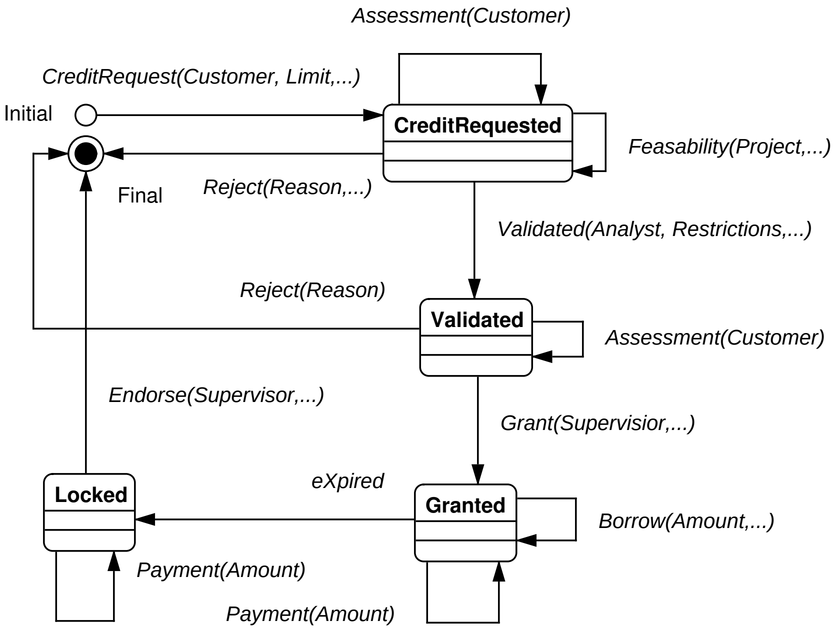
Example 7.1. *Revolving credit example* ▽

In this example from the banking domain we present the structure of a revolving credit class and the behavior its objects may perform. It is in no way complete and many necessary features are not modeled. We use this simplified class for the sake of demonstration only.

A credit is revolving, if the borrower is granted a certain credit limit, which must not be extended. Up to that limit the borrower can borrow money and pay it back. The borrower has to request a credit from the bank, whose analysts are in duty for assessing the credit-worthiness of the borrower and also they must perform feasibility checks (e.g. whether the business for which the money is meant is really promising). Such validation processes may be repeated, depending on the limit requested, or on gathered evidence. The result of the validation process may be to reject or to grant the credit. Grants are given by supervisors. That followed, the borrower is allowed to borrow money from the bank and he/she may pay back a certain amount. In this example, the credit expires, which can be triggered automatically after the end of the credit’s duration or on the initiative of



(a) Class definition



(b) Statechart

Figure 7.2: Structure and behavior of the *RevolvingCredit* Class

a supervisor. After a last series of back-payments, the termination of the credit is endorsed by a supervisor.

In figure 7.2(a) on the facing page the structure of the revolving credit class and in figure 7.2(b) the statechart describing the allowed behavior of its objects is depicted. Additionally to the public methods which are accessible from other classes, some attributes are given. The statechart shows, that e.g. in the state *Validated* only the events *Reject*, *Assessment*, or *Grant* are accepted as legal method call to the object. Apparently, a further feasibility study is not allowed after the analysts has validated a credit request. In this statechart the dependence of the object's behavior on the values of attributes is not specified. If, e.g. a borrower is sending a request for *Borrow*ing a certain amount to a credit object being in the state *Granted*, then one time this may lead to success and another time the request may be rejected. The reaction depends on the value of the parameter *Amount* and on the (hidden) current value of the *Balance*[2]. Both values together determine, whether the requested amount exceeds the credit limit or not.

Different objects of the same class might comply to different curricula, specified as path of sequentially issued events through the state chart. A life cycle of an object is defined by one path leading from the initial to the final state. A credit request may be assessed multiple times, whereas another one is not assessed at all and gets its validation seal by the analyst at once, which may be due to the already known credit-worthiness of the borrower. In testing a class, many objects must be generated and their life cycles should cover all paths from the initial state to the final one.

In figure 7.3 on the next page a set of test cases for the class *RevolvingCredit* is shown. For the sake of readability, all events are abbreviated and indicated by the upper case letters contained in their designators. For example, the event *Reject* is given as *R* in the test set, whereas *eXpired* is denoted as *X*. One line in the test case set is the sequence of events sent to one specific object of the class. Different lines belong to the test sequences of different objects.

The reader may note, that, due to the loops specified by the reflexive transitions in the state chart the number of paths is infinite and the entirety of an object's possible behavior cannot be reflected in test cases. Nevertheless, the main parts of the functionality of the class *RevolvingCredit* can be found in the set of test cases. △

[2]Such dependencies can be specified with statecharts by establishing conditions which must hold on the occurrence of an event. However, since the graph structure is not affected by this additional information we neglect it.

```
C V G B P X E
C A R
C A F V G B B X P E
C A F R
C F V G B P P B P P B P X E
C A A F V G B X P E
C V A G B P X E
C R
C A V G B X P E
C A F V A R
C V R
```

Figure 7.3: Test cases for the class *RevolvingCredit*

This example is used further on for discussing the benefits and deficiencies of the presented test sequence analysis methods. In the following sections, two methods for analyzing sequences of events are discussed and some improvements for adapting them to the domain of describing reusable components are presented.

7.2 Sequence analysis

The challenge of analyzing a set of test cases can be paraphrased as follows: "Given is a set of sequences of method calls to a component. Are there any regularities contained in that sequences?"

The assumption is that such regularities exist. Regularities can be described by formal languages which can be ordered according to their position in the Chomsky hierarchy. This hierarchy was established by Noam Chomsky who characterized these classes as possible models for natural languages [103]. The hierarchy contains four classes of languages (or computing machines) that have increasing complexity: regular (finite-state automata), context-free (push-down automata), context-sensitive (linear bounded automata), and recursive languages (Turing machines). Since we do not know which level of complexity is given in a test set, we assume only the least complex class, which is the class of regular languages. Regular languages can be described by means of *grammar*s[3], or by *finite-state automata* (FSA). Both representations are equivalent. A sequence of tokens which

[3]In general, regular languages are not described by grammars, but by regular expressions. However, because grammars describe higher classes of the Chomsky hierarchy, due to their power regular languages are included as well.

conform to the rules expressed in the grammar, is called a *word* of the grammar. Words are built by tokens, which are elements from the alphabet of the language. In that way tokens are the base symbols of words. Finite-state automata can be deterministic or indeterministic. Indeterminism means that there are more than one possible successor states, when one token is recognized. This is no restriction, since deterministic and indeterministic finite state automata are equivalent and each indeterministic automaton can be transformed into a deterministic one.

Since state charts are an extension to finite-state automata the initial problem statement now can be reformulated more exactly: "Given is a finite set of words of an unknown language which is assumed to be regular. What is the grammar (or deterministic finite state automaton) accepting these words?"

In our problem context words are sequences, which are elements of the test set. A grammar or automaton infered in such a way is a behavioral description of the component. The input to the sequence analyzing process is always a finite set of finite words. This property renders the description of components by grammars or FSAs possible, because finite words can always be represented by an equivalent FSA [103]. Software developers are used to this kind of representations and if it indeed reflects the characteristics of the underlying behavior, then grammars and FSAs are an adequate description form.

As one can see from the given example above, not all the necessary information can be obtained from the test set, such as loops and control path branches which were not executed. Furthermore, it is in general not possible to distinguish in a test case between the serial occurrence of a pattern in the sequence and an (possibly infinite) iteration of this pattern produced by a loop.

In the literature, the detection of regularities in a knowledge base is called *sequence learning* [219]. Learning sequential behavior is an important task in many fields of application, such as planning, reasoning, natural language processing, adaptive control, time series prediction, financial engineering, or DNA sequencing. The problem of inferring formal grammars from words is a subfield of sequence learning and it is refered to as *grammar inference* which has been studied for more than thirty years. Mark Gold [86] proved, that in general the inference of a canonical FSA is NP-hard. A canonical FSA is an automaton accepting the given words with a minimal number of states and not accepting other words than given in the example set.

Hence, in the mid 80ies research in grammar inference concentrated on algorithms using heuristics and additional domain information to infer FSAs from an example set. Dana Angluin [4] developed the RPNI algorithm (regular positive and negative inference) using (positive) examples and (negative) counter-examples as well as an oracle. This oracle (in most cases a domain expert) answered questions about the membership of words to a grammar. Many improvements are suggested to reduce the space complexity by slicing the search space [65] or to decrease time complexity [172, 173].

A specific restriction of the test case domain is that here only positive examples, but no counter examples are given in the test base. This changes the situation, because Gold showed in addition that if only positive examples and no negative examples are given, such a building process is impossible for the general case [86]. When encountering positive examples only, an algorithm has no means to determine whether it is over-generalizing the automaton or not [4]. These results reveal, that algorithms inferring grammars from examples cannot be exact and the quality of their results must be judged otherwise. We may rank inference algorithms according to the following desiderata [1]:

Completeness: The grammar (FSA) should accept all words of the example set. There should be no behavior present which is not described by the grammar (FSA).

Veracity: The grammar should contain no spurious rules (the FSA should have no spurious transitions).

Minimality: The number of rules (transitions) should be minimal to clarify the representation.

Only the first demand can be verified automatically. This is due to the fact, that the original grammar (or FSA) producing the test cases must be known (which is not possible for all components), or in the case the original grammar is available, a canonical representation of it has to be available. However, in comparing results directly, these properties help to judge the quality of inference algorithms. Since the design of the algorithms might be focussed on diverting aspects, the quality of the results may trade off. If, for instance the main goal is to cover all examples completely, the minimality of the resulting grammar could suffer from that demand.

[4] An exact result is achievable only, when a single word is analyzed.

In the next sections, two different approaches are presented. They are chosen, because in comparison with other methods (e.g. based on neural nets [51], Markov models [145, 168], or graph cluster analysis [1]) the quality of the results are equal or superior when compared regarding the above mentioned criteria.

7.3 Prefix analysis

The approach presented in this section analyzes example words which are seen as sequences of tokens and generate a FSA which reflects the patterns they exhibit. The origin of the approach is attributed to the work of Biermann and Feldman [25]. The proposed algorithm *k*TAIL was developed by Cook [51] who uses it as tool for the automatic recovery of software development processes from log data. We improved it in two ways:

- We enclosed explicit start states and final events. This enables a better understanding of the resulting FSAs.
- We added a heuristics to built clusters of states to detect such states which belong together.

The main idea is to determine on the basis of behavior recognized up to now (prefixes) the *future* of a sequence. Prefixes contain the sequence of tokens from the first symbol read, up to the currently read token, hence a sequence of length n is built by n prefixes. E.g. a sequence abc is decomposed into the set of prefixes $\{a, ab, abc\}$. A future is given by the the next k tokens which occur in the input stream. The value of k is given as parameter to the algorithm. A given prefix may lead to different futures, as well as a future of length k may be reachable by different prefixes. Two different prefixes are put to the same equivalence class, if they have the same future. Finally, a state in the FSA is equivalent to an equivalence class.

7.3.1 The prefix algorithm *k*TAIL in detail

The alphabet A, from which the words of the language are built, is the set of events $e_1, e_2, \ldots, e_n$. A word of the language is a sequence of events. The *k*TAIL algorithm operates on the sample set S_+ which contains words of the language. The set of all prefixes in S_+ is denoted by P. Any word w can be divided into a prefix p and a tail t in such a way that $w = p \circ t$. The operator '$\circ$' is the concatenation operator. Furthermore, the set T_k contains all words of length l, with $l \leq k$ composed from A.

An equivalence class E contains all prefixes with identical tails. This implies that every prefix is at least member of one equivalence class, but prefixes can be member of more than one class:

$$\forall (p, p') \in E,\ \forall t \in T_k \mid (p \circ t) \in P \wedge (p' \circ t) \in P.$$

Each equivalence class E_i corresponds to a state in the resulting FSA and for the reason of simplification we denote a state with the label of the respective equivalence class E_i as well. After having generated states the transitions are to be defined. Formally, a transition is a triple $\langle E_s, E_d, e\rangle$, where E_s is the source state, E_d is the destination state, and e is the event triggering the transition.

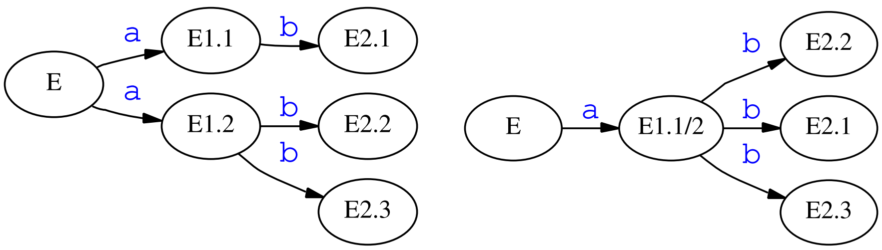
From the source state E_s a transition labeled with an event e taken from A leads to a destination state E_d, when there is a prefix $p \in E_s$ in such a way that the prefix $p \circ e$ is in E_d:

$$\forall \langle E_s, E_d, e\rangle\ \exists p \in E_s \mid p \circ e \in E_d$$

If only one transition is outgoing from E_s on the occurrence of the event e, this transition is deterministic. If there are more than one, it is indeterministic.

The algorithm produces correct and complete FSAs, but it tends to introduce to many states which is due to the algorithms restriction to look only k events into the future. If the future which follows k is equal for some prefixes as well, then the states are redundant. Cook improved the initial algorithm from Biermann and Feldman by analyzing the states which follow a transition and merge those who demonstrate identical behavior.

Let $\mathcal{E}_d$ be the set of all states, which are the destination of the transitions from E_s labeled with the same event e, formally $\langle E_s, \mathcal{E}_d, e\rangle$. If all transitions outgoing from the states in $\mathcal{E}$ are labeled identically as well, these states can be merged. All transitions from E_s to the states in $\mathcal{E}_d$ are merged as well. Such an activity is depicted in figure 7.4 on the facing page with a value of k larger than 1. The states “E1.1” and “E1.2” are redundant, because from “E” they are reached by the same token “a” and their successor states are reached by identical tokens as well (figure 7.4(a)). The tokens recognizable from their successor states may be different. These redundant states are finally merged in the simpler graph of figure 7.4(b). Obviously, a parameter value of $k = 1$ means that no simplification can take place: for each accepted token a state of its own is generated and no more than one transition outgoing from a state can be labeled with the same token.



(a) FSA with redundant states

(b) FSA after merging redundant states

Figure 7.4: Simplifying a FSA ($k > 1$)

Example 7.2. *Sequence analysis with k*TAIL ▽

Here, a simple sequence of events generated from the alphabet {`a, b, c`} is given (taken from [51]). There are some repetition patterns present: a `a-b-c` iteration and `b-a-c` iteration:

```
a b c a b c b a c b a c a b c b a c b a c ...
... a b c b a c a b c a b c b a c b a
```

From that example, the importance of a correct adjustment of the parameter k becomes obvious. Since the length of the iteration patterns is 3 (with the assumption, there is really a loop in the automaton generating that sequence), a value of $k > 3$ raises the level of determinism (since more accuracy is added by looking deeper into the future) but the characteristics of the behavior are not reflected anymore in the resulting automaton. If the algorithm tries to find patterns of a length which is larger then the length of existing patterns, then the resulting FSA specifies a behavior which does not reflect the patterns. This is due to the fact that the pattern search is started in an iteration i and ends in a subsequent iteration $i + k$.

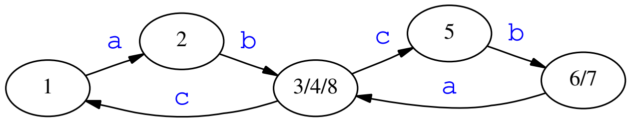


Figure 7.5: The FSA generated with kTAIL (k=2)

For the resulting finite state automaton, given in figure 7.5, a value of $k = 2$ is chosen. For any value k larger than 2 the states of the FSA are labeled with numbers[5]. In the example, the state with the label "3/4/8" is the result of the optimizing process where the intermediate states are merged. Note, that there is an indeterminism in the FSA, because there are two transitions labeled with `c` which can fire in state "3/4/8". This is due to the merging step and the algorithm sees two different possible futures after recognizing the token `c`. The algorithm performs well in detecting the iterations contained in the sequence. △

7.3.2 Improvements of kTAIL

Explicit initial state and final states

From the FSA generated by kTAIL the initial state and the final states cannot be recognized, unless the underlying words are known. This can be confusing, particularly when the resulting structure is complex and contains loops involving initial and end states. Our first improvement therefore is to mark these states explicitly. A FSA is defined to have one initial state and a set of end states [103]. In the resulting FSA the initial state is marked with the meta symbol `start`. The initial state is a pseudo construct, since it is introduced for the purpose of defining the begin of the FSA only and there is no transition to it. Any equivalence class can become a final state, which is depicted by the meta event `end` in the graph.

To fulfill these requirements we add these two meta symbols to the words when parsing the input. Due to the design of the kTAIL algorithm this additional information leads to the desired results.

Strongly connected subgraphs

Sometimes, repetitive pattern form a larger pattern. This is especially true for reusable components. Consider the *RevolvingCredit* example, where two large

[5] For $k = 1$, since the future of prefixes is determined by single tokens from the alphabet, each generated state is labeled with the token leading to the respective state.

groups of activities can be identified: the activities necessary for validating the request and the activities dealing with the “consummation” of the credit. These large patterns can be identified in the event sequences as well.

One further improvement to the understandability of the FSA therefore is the option to restructure it, if the evidence for such a larger pattern is given. If it is demanded, the *k*TAIL algorithm searches for *strongly connected components* (scc) in the graph of the FSA. Strongly connected components are supposed to group states which conceptually belong together in the way, that the states in that component accept the same event patterns. This change step is not an algorithmical enhancement to get better results, but it comes from the analysis domain. If there is no rational in the domain to group strongly connected vertices, to perform this graph regrouping step is not recommended. The original idea of determining strongly connected components in graphs to determine clusters of behavior was already suggested by [1]. In their paper, the authors examine workflow logs and re-engineer process models from them. Clusters are considered as those steps of the process which conceptually belong together and therefore, can be seen as subprocesses.

A strongly connected component is recognized as follow: In a directed graph, two vertices are strongly connected, if there exists a path from the one to the other. A single vertice is always strongly connected. A directed graph is strongly connected, if all its vertices are strongly connected. A maximal strongly connected subgraph of a directed graph is called its strongly connected component [224, p 100]. An efficient algorithm for computing strongly connected component is proposed in [27].

Strongly connected components can be collapsed to one vertice, which forms a condensed graph. In our adaptation we do not collapse them, but mark them as subgraphs. This allows to identify at a quick glance regions of behavior, which can be considered in isolation. In the following continuation of the previous example 7.1, we apply the improved *k*TAIL algorithm to the given event sequence.

We implemented *k*TAIL from scratch and the improvements here presented are added as well. The program is realized in the programming language SETL2.

Example 7.3. *The k*TAIL *algorithm applied to example 7.1 on page 157* ▽

The result of the analysis of the event sequences with *k*TAIL (k=1) is shown in figure 7.6 on the following page. The parameter value is chosen, because the iterations evident in the sequences are considered to have a maximum length of 1. The FSA is complete with respect to the available test data, since each test case is accepted by the FSA. Please note, that no repetition of *Feasibility* events is in the event sequence, which leads to the missing reflexive edge for the state `F`. Albeit, this is no deficiency of the algorithm, but an incompleteness of the test set.

Due to the cluster formed by the strongly connected components two behaviorally different regions can be identified. The strongly connected components are depicted as gray boxes in the graph of the FSA. They conform well with the understanding of the domain in the way that the component, containing the states `A`, `F`, and `V`, is concerned with the initialization process, whereas the second component deals with the behavior of the credit transaction itself.

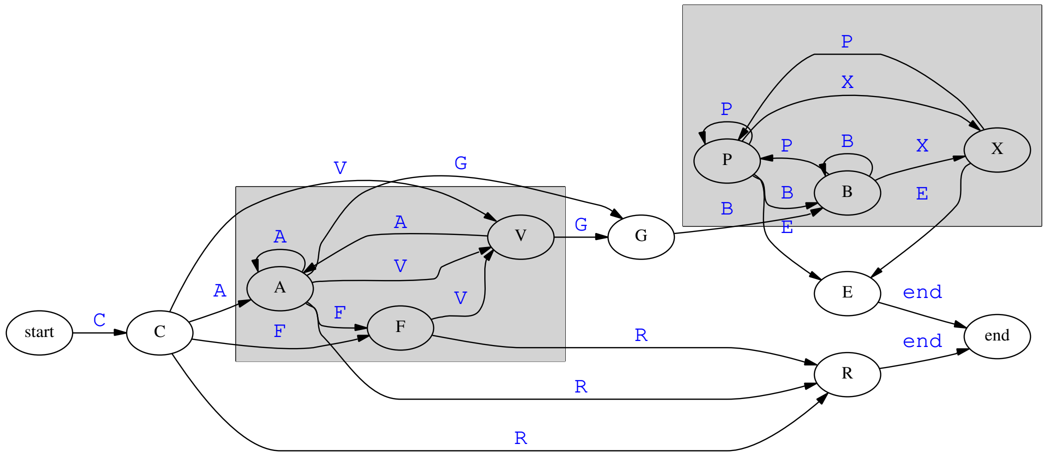


Figure 7.6: *RevolvingCredit* behavior as inferred by *k*TAIL

The algorithm was not able to detect the difference between the *CreditRequested* state and the *Validated* state, because in the latter an event can be accepted which is legal in the former state as well. Because of the observation of a future of length 1, the state accepting an *Assessment* is always the same. This leads to the given state `V`, which allows an event sequence, where after the request is *Validated*, an event *Feasibility* is accepted. Such a sequence

1 does not appear anywhere in the test cases and
2 is not defined in the statechart specification as well.

Thus, the veracity[6] of the FSA is not given in this example. With a higher

[6]Veracity is the property to which extend spurious transitions occur in the FSA (defined on

value for k, the described effect can be avoided, but then the number of states is growing larger.

Regarding minimality, the resulting FSA is affected by the choice of the parameter k as well. Here, each accepted token leads to a state of its own. E.g. on a conceptual level, the states (and the intermediate transactions) `start` and `C`, resp. `E`, `R`, and `end` can be merged. Choosing a larger k leads to more states, (and to simplifications afterwards) and this demands more transitions as well. Also here, the structure of the examples determines the necessity of transactions. △

7.3.3 Evaluation of *k*TAIL

Regular positive inference conducted with *k*TAIL is promising under the following preconditions:

- The value of k must be adjusted correctly to reflect the behavior embodied in the sequences of words. If k is too small, repetition patterns are splitted, if it is too large, these patterns are not detected correctly. Thus, the resulting FSA, although correct, does not render a good description, since the representation such obtained is not conforming with the searchers expectations.
- The length of repeating patterns should not vary. If loops with different iteration lengths are evident in the test cases, then in general it is not possible to adjust k correctly, leading to the effects mentioned in the previous point.

When the domain is understood, the two parameters (k, strongly connected components) are enough to adjust the *k*TAIL algorithm. Furthermore, only one pass over the input data is sufficient for analysis. This gives a certain degree of freedom to the analyst to experiment with values for k to generate a adequate description.

A further advantage is clearly the behavior's representation as a FSA. Application engineers are normally used to formal descriptions on the graphical level and FSAs are a standard specification language for repetitive patterns.

As already discussed with the example, *completeness* of the *k*TAIL with respect to the available event sequences is given. The simplification step does not affect completeness: Only states are merged, but transitions are not deleted (with the exception of redundant transitions). *Veracity* depends heavily on the structure

---

page 162).

of the event sequences and the correct choice of k. It cannot be guaranteed by the algorithm. This is the case with *minimality* either. A canonical automaton which has a minimal number of transitions and does only accept words specified by the behavior (defined by the event sequences), was not the aim of the algorithm's design.

Prefix-analysis algorithms like *k*TAIL define states by identifying identical histories of events which have identical futures of a given length. Heuristics introduced by (1) determining the length of the future and whether (2) the capability of states to accept common events define subgraphs, lead to state abstraction resulting in the final FSA. The correct adjustment of domain parameters finally is the decisive factor for the quality of the components behavioral description.

The idea of the algorithm presented in the following section is different. Not the history from the start of the sequence, but only a small "window" of observation is the basis for the abstraction. Identical patterns are then considered to represent the same concept. The most significant difference is the fact, that such abstractions are regrouped *hierarchically*. A further difference is the representational concept: Not finite state automata, but grammar representations yield the final description.

7.4 Pattern analysis

In this section we present the algorithm SEQUITUR developed by Nevill-Manning [163] and an extension to it for enhancing the resulting representation to handle multiple sequences and detect repetition patterns.

7.4.1 Pattern encoding with SEQUITUR

SEQUITUR generates context-free grammars from a given word. In fact, although the grammar belongs to this complexity class, the languages these grammars recognize are only regular ones, since each language described by the generated grammar is finite and produces only one word. Thus, SEQUITUR is a "context-free" technique for describing repetitive patterns.

Basically, the algorithm analyzes a sequence and substitutes each occurring repetition by a rule. The aim of the algorithm is twofold. First, it generates a shorter representation of the input word, thus encoding it as grammar. Second, as a by-product a structural explanation of this sequence is provided.

A context free grammar consists of a set of rules of the form $A \rightarrow \alpha$, where A is a non-terminal symbol and α denotes a string of terminal symbols and non-terminal symbols. For a rule in that form A is called the header of the rule or the rule's name as well. Consider the example sequence `a b c a b`, where a repetition of the subsequence `a b` is given. SEQUITUR replaces it with a nonterminal and introduces a new rule with that symbol (nonterminals are represented starting with a $ and the grammar's start symbol is given as **S**):

```
S  → $A c $A
$A → a b
```

A patterns is given by a *digram*. A digram consists of exactly two symbols (thus the name *di*gram) which may be terminals or non-terminals. The grammars produced share the following two properties:

Digram uniqueness: No pair of adjacent symbols exists more than once in the grammar.

Rule utility: Every rule is used more than once.

The algorithm starts a new analysis process with generating an initially empty start rule. When a new symbol is read from the input stream it is appended to the start rule. The last two symbols of the right hand side (rhs) of the start rule form a new digram. *Digram uniqueness* is checked every time a new symbol is appended. If the new digram appears anywhere else in the grammar, then the following situations may occur:

- The digram is a complete rhs of a rule. Then the last two symbols of the start rule's rhs (the new digram) are substituted by the rule's name.
- The digram is a partial sequence of a rhs. Then a new rule is introduced with the digram at the rhs. Each occurrence of the digram in the grammar is substituted by the new rule's name.

This process may trigger iteratively further checks and substitutions, because on each check the grammar's structure may change.

Longer rhs are formed by enforcing the second property, *rule utility*. If a rule is used only once in the grammar, it is deleted and its name is substituted by its rhs. This property should prohibit an overly complex structure of the grammar. As continuation of the example above, lets suppose one more symbol, `c`, is present at the input. The property digram uniqueness forces the introduction of a new rule, leading to this grammar:

$$\begin{aligned} S &\rightarrow \$B\ \$B \\ \$A &\rightarrow \mathtt{a\ b} \\ \$B &\rightarrow \$A\ \mathtt{c} \end{aligned}$$

Now we see, that rule $A is used only once, as rhs of rule $B. Following the command of rule utility, $A is deleted and its body substitutes its name in $B, which gives the final grammar:

$$\begin{aligned} S &\rightarrow \$B\ \$B \\ \$B &\rightarrow \mathtt{a\ b\ c} \end{aligned}$$

We use the sequence already presented for demonstrating the effects of *k*TAIL here as well.

Example 7.4. *Sequence analysis with* SEQUITUR ▽
Given is the sequence presented in example 7.2 on page 165. The SEQUITUR algorithm produces the following grammar:

$$\begin{aligned} S &\rightarrow \$B\ \$F\ \$F\ \$E\ \$B\ \$E\ \$C \\ \$E &\rightarrow \$B\ \$D \\ \$D &\rightarrow \$C\ \mathtt{c} \\ \$B &\rightarrow \mathtt{a\ b\ c} \\ \$F &\rightarrow \$E\ \$D \\ \$C &\rightarrow \mathtt{b\ a} \end{aligned}$$

The algorithm generates a grammar from the example which is complete: The whole sequence is producible by it. The graph consist of six rules which form a hierarchical grammar. Since there is no recursion, the length of the producible word is restricted. △

7.4.2 Discussion of SEQUITUR

Due to the substitution of multiply occurring sequences (either terminals, or terminals and nonterminals) the SEQUITUR encodes an input sequence correctly and produces a complete grammar. However, although the algorithm detects repetitions in the input, this does not describe the behavior of the component sufficiently, since only one possible spectrum of activity is captured.

The term repetition refers to a sequence of at least two symbols which appear anywhere else in the sequence of events. However, sometimes the domain demands another view point onto repetition. In the example, the length of loops is at least 3. Because of the property of rule utility, one of the loop patterns was detected ($B), the other one is hidden in the rules $C and $D. If a domain shows such regularities clearly, the description produced should reflect them better. SEQUITUR lacks this possibility to adapt itself to *domain knowledge*.

It is evident from the example as well, that there are iterations in the sequence (and not restricted repetitions only). The aim of a behavioral description is not to find one for an example only, but *to abstract* from it to an evident general concept. Obviously, since the length of the sequence is finite, this abstraction cannot be exact. E.g. in the example above, the subsequence `a-b-c` appears many times in such a way that we can assume with a high degree of certainty to face an iteration. Iterative behavior is in general reproducible many times more than it is present in the example.

A set of test cases contains many event sequences. SEQUITUR is able to encode only one and the knowledge such obtained is not comprised in the following analyses. Since different event sequences demonstrate the behavior of one component, these intermediate grammars must be subsumed into a final one.

In the next section we present the algorithm MSEQ (multiple SEQUITUR), which is an extension to SEQUITUR, where all the mentioned requirements are considered. The algorithm is implemented in the high level programming language `SetL2` and given in detail in appendix C.1.

7.4.3 MSEQ, an improvement of SEQUITUR

Pattern length variation

The algorithm MSEQ is based on SEQUITUR, which we implemented from scratch. The first enhancement is the option to vary the length of patterns. In the standard approach a repetition is given, if a pattern of length 2 (a *di*gram) is found elsewhere in the grammar. In the implementation the pattern length can be varied by defining the value of the parameter `n`. Its default value is 2 and in that case MSEQ behaves like the standard algorithm. The domain engineer (or librarian) now can use her/his knowledge about a sequence's repeated structures by adjusting the correct length of a $n$gram.

Example 7.5. MSEQ *with varying* n*gram length* ▽
Here the effect of adapting the parameter to domain requirements is demonstrated. The sequence of example 7.2 is analyzed with `mseq -i inputfile -n 3`, which leads to the following grammar (the parameter `-i` determines the input file):

S $\rightarrow$ $A $C $C $A $B $A $A $B `b a`
$C $\rightarrow$ $A $B $B
$A $\rightarrow$ `a b c`
$B $\rightarrow$ `b a c`

In contrast to the grammar of example 7.4 obtained by the default value, both patterns manifest themselves as rule bodies (of rules $A and $B). Because of the minimal length of 3 for patterns to be substituted, the number of rules such obtained is only 4 (compared to 6 rules with `-n 2`). △

The reader may note, that the pattern length does not influence the length of rules. Also with a length n larger than 2, due to the effects of rule utility, rule bodies may have a length smaller than n. Furthermore, the grammars obtained by a pattern larger than 2 may not differ from that produced by the standard SEQUITUR (except for rule names). Such effects depend heavily on the structure of the underlying sequences. If e.g. a sequence `a b c a b c` is analyzed with `-n 2` or `-n 3` does not make a difference. The pattern leads to the substitution with $A `c` $A `c` in both cases. The first time because of the hierarchical intermediate result and the succeeding application of rule utility, the second time because of the predefined substitution restrictions.

The parameter for varying the pattern length is used for adapting the algorithm to recognize specific repetition structures in the event sequences.

Iteration heuristics

Up to that point grammars produced by MSEQ do a good job in describing a sequence. The encoding produces a lossless representation of the initial order of events. On the other hand, exactly this sequence is described, whereas a more general description of the behavior of a component producing this (and other) event sequences is not provided, although some hints are there. Consider the sequence "`...a b c a b c...`". If iterative behavior is expected, such an example could be the evidence for a loop. This is the reason, why we introduce the option to gather this information from the example. The librarian specifies it by setting the `-r` parameter with the number of repetitiveness. If a pattern (of arbitrary length) is occurring successively at least r times in the grammar, it is considered as the effect of a loop. All successive patterns (with the exception of the first pattern) are deleted and the remaining first pattern is parenthesized by the meta symbols '`(`' and '`)+`'. The symbol '`+`' indicates that the iterated pattern must occur at least one time [103]. We have prefered the iterative representation to the (correct) recursive representation, since recursive structures are not that simple to understand and our aim is to generate an easy to understand representation. This is not in accordance with formal language theory, where such a meta-symbol is not known. However, since the complexity class of the languages inferred is only regular, this "mixture" of regular expressions and context free grammars is acceptable.

Please note, that we do not demand a minimal iteration of length r in the resulting grammar. The repetition indicator r defines the lower bound which supports the presumption of a loop existence only. The resulting grammar accepts languages with iterations less than r as well, but the simplifications which would be possible are not performed.

The presented algorithm 7.1 `RepetitionCheck` is part of MSEQ and it is called after the standard procedures `NGramUniqueness` and `RuleUtility` were employed. It gets the grammar g and the parameter r as inputs and returns a grammar in which all successive repetitions of length r are substituted by a loop construct. Since this algorithm may change the rules, afterwards rule utility must be checked again. The grammar is a set of rules, which are pairs in the form (lhs, rhs). The right hand side rhs is organized as list of symbols.

The idea is to generate for every existing rhs the set of all of its sublists which can be repetitive patterns. Obviously, such a pattern must not be longer than $\#rhs/r$, since it should occur at least r times in rhs. The generation of sublists of length l, with $l \leq \#rhs/r$, is performed by the auxiliary function `PowerList`. Since we try to detect as large patterns as possible, the sublist m with maximal length is selected from that power list. Indeed, if such an m exists at least r times within rhs, then the sequence $\underbrace{m, \dots, m}_{r\text{-times}}, m, \dots, m$ is transformed to $(m)+$. The function checking the existence of a sublist in a list is named

`IsSubList(sublist, list)`. The transformation of a sequence to a loop is performed by the function `SequenceToIteration` which gets the pattern m, the minimum sequential occurrence r of that pattern, the rule name rhs of the rule in which the repetition takes place, and the grammar g as input. The result of the function is a new grammar. When a repetition pattern is found and substituted by its loop counterpart, the analysis does not stop: within the pattern m further repetition may exist as well.

Here the complete algorithm is presented, whereas input- and output parameters are denoted with ↓, resp. ↑, and in-out ones are given as ↓↑.

Algorithm 7.1.

```
PROCEDURE RepetitionCheck(↓↑g, ↓r) ;
  BEGIN
    FOR (lhs, rhs) ∈ g LOOP
      pl := PowerList(rhs);
      FOR m ∈ pl |
            ∀m' ∈ pl | #m ≥ #m' LOOP
        IF IsSubList(m * r, rhs) THEN
          g := SequenceToIteration(m, r, lhs, g);
        END IF;
        pl := pl \ m;
      END LOOP;
    END LOOP;
  END RepetitionCheck;
```

Whereas the parameter n introduces domain heuristics by affecting the encoding length, r does it by abstracting from repetitions. However, both adaptations change the structure of the resulting grammar. The value of r is not dependent on n, it may be smaller or larger than n. Here, the responsibility lies by the person producing the grammar to select the best values for the parameter. However, the effects of the parameters are directly observable and their variation is immediately reflected in the results. In that way this enhancement helps the describer in adapting the description to the domain and component peculiarities. As an example, consider the sequence “`a b c a b c a b c`”. The standard SEQUITUR produces this description:

```
S  → $A $A $A
$A → a b c
```

The repetition is clearly reflected in the grammar, although when someone searches for a loop, this grammar might be confusing. If the analysis is run with MSEQ (`mseq -i inputfile -r 2`), the resulting grammar may describe the behavior of the underlying component better, thus supporting the searcher more effectively.

```
S → (a b c)+
```

In the next example the sequence from example 7.2 on page 165 is checked for repetitions of length 2.

Example 7.6. *Detecting iterations* ▽
As already stated, the component producing that sequence is considered to have loops of length 3. Due to the loop assumption, a single repetition is seen as evidence for the loop, thus the value for r is set to 2. The algorithm was called with `mseq -i inputfile -n 3 -r 2`. Here is the resulting grammar:

```
S  → $A ( $A ( $B )+ )+ $A $B ( $A )+ $B b a
$A → a b c
$B → b a c
```

In contrast to the results obtained by the standard SEQUITUR (example 7.4 on page 172) the grammar is more intuitive with respect to the event sequence. Additionally, this representation with only three rules is more compact and therefore, better suited for the description purpose. △

The enhancements of MSEQ presented up to now deal with adaptations for respecting the domain better and improving the descriptiveness of the grammars for reflecting the behavior of the components producing the analyzed sequences. To reach our goal of obtaining a partial specification of the component, more than one sequences must be considered and the knowledge such recovered must be unified to one description. This is the topic of the next section.

Multiple sequence analysis with MSEQ

Multiple sequence analysis aims at merging the grammars which are produced by MSEQ. It bases on two assumptions:

1. The event sequences which are analyzed are produced by the same component.
2. Structural identical rules are equal, although non-terminals contained in them are not.

The idea is to establish an *unified grammar* (UG) and to check for each single grammar G, if its rules are already contained in UG. If a rule is not yet member of UG, then it is added to it. The algorithm has to ensure, that all words producible by G are producible by UG as well. In analyzing multiple sequences, for all grammars infered, MSEQ generates unique rule header names (with the exception of start rules). Hence, a circular reference can never be introduced during grammar merging. As a consequence, unique rule names allow a top down approach for building the unified grammar.

In detail, during analysis these situations must be considered:

- There is a rhs-match between $r \in G$ and $gr \in UG$.
  1. If r is the *start rule* of G, then the following alternatives must be verified further:
     - The corresponding rule $gr \in UG$ is the start rule of UG. In such case, nothing need to be done, because the starting rule r is already member in UG.
     - The rule gr is not the start rule of UG and its name is not part of the start rule either. Since the rule r can be triggered as start rule too, the symbol for alternative "|" is appended to the start rule of UG, as well as the name of gr(!). This has to be done, because the bodies of r and gr are identical and thus no renaming in UG is necessary.
  2. In the case that neither r nor gr are start symbols, each occurrence of gr's name in the UG is replaced by the name of r. In this way the new rule r is connected to the other rules in UG (which may be incorporated later as well).
- If there is no right hand side match (the rhs of r was not detected anywhere in the UG), the rule r it must be added to the unified grammar UG. If r is the start rule in G, then its body is appended as alternative to the start rule of UG. If r is not the start rule of grammar G, r is added without modifying any existing rule in UG.

Obviously, the predicate used for determining the equality of two rules' rhs must not base on the rule's name (their left hand side). The challenge for the comparison of two rules' right hand sides is given by the following three requirements: (1) the names of rules have to be neglected and (2) all non-terminals contained in it must be checked for equality as well and (3) alternatives of rules in the form $A $\rightarrow$ $B | $C must be considered. In that way, rhs are validated recursively, when determining their equality.

Algorithm 7.2 merges a grammar G with an unified grammar $UnifGrammar$. The following representational conventions are respected: If the left hand side of a rule r is addressed, then this is given as $r.lhs$, resp. $r.rhs$ for the right hand side. A rule s is selected from a grammar F by restricting the grammar with $F(s)$.

To simplify the representation of the algorithm, the following auxiliary procedures or functions are introduced:

`Eq(`$\downarrow lhs1$`, `$\downarrow lhs2$`):BOOLEAN:` This predicate is used for comparing the rhs of two rules. It performs a deep analysis of the two lists, whereas rule names are recursively checked for equality with `Eq`. The predicate is specified as follows [103]: If a word w is inferable from a rule r with respect to a grammar G, (symbolically $r \overset{G*}{\Rightarrow} w$) and the same word w is inferable from a rule p with respect to a grammar G' ($p \overset{G'*}{\Rightarrow} w$), then r and g are equal, $r \Longleftrightarrow p$.

Add($\downarrow r$, $\downarrow\uparrow G$): This procedure is used for adding a construct r to a grammar, where r may be a rule or a right hand side of a rule. It can be employed in two ways: (1) If called with a rule G and a right hand side r, it adds the rhs as alternative to the rule, or (2) if called with a grammar G and a rule r, it puts the rule to the grammar's rule set.

`Replace(`$\downarrow olhs$`, `$\downarrow nlhs$`, `$\downarrow\uparrow G$`):` In grammar G all occurrences of the old non-terminal $olhs$ are replaced by the non-terminal $nlhs$.

`GenNewRule(`$\downarrow rhs$`):Rule:` This function generates a new rule $\$NT \rightarrow rhs$ from the incoming right hand side rhs and the internally generated non-terminal symbol NT.

Algorithm 7.2.

```
PROCEDURE MultipleGrammarMerge (↓↑ UnifGrammar, ↓ G);
BEGIN
  IF UnifGrammar = {} THEN UnifGrammar := G;
  ELSE
    FOR aRule ∈ G | aRule ∉ UnifGrammar LOOP
      IF ∃uGRule ∈ UnifGrammar | Eq(uGRule.rhs, aRule.rhs)
      THEN
        IF aRule.lhs = StartSym THEN
          IF uGRule.lhs ≠ StartSym ∧
              uGRule.lhs ∉ UnifGrammar(StartSym).rhs
          THEN
            Add(UnifGrammar(StartSym), uGRule.lhs);
          END IF;
        ELSEIF uGRule.lhs = StartSym THEN
          Add(UnifGrammar, aRule);
        ELSE Replace(uGRule.lhs, aRule.lhs, UnifGrammar);
        END IF;
      ELSE
        IF aRule.lhs = StartSym THEN
          newRule := GenNewRule(aRule.rhs);
          Add(UnifGrammar(StartSym), aRule.lhs)
        ELSE
          newrule := aRule;
        END IF;
          Add(UnifGrammar, newrule);
      END IF;
    END LOOP;
  END IF;
END MultipleGrammarMerge;
```

This procedure is activated in MSEQ when the −m option is set. A merged grammar is checked for $n$gram uniqueness and rule utility like any other grammar too. In the following example the effects of rule merging are demonstrated in detail stepwise.

Example 7.7. *Grammar merging with* MSEQ ▽
Given are the following two event sequences, stored in the input file `sequence1`:

```
a b c d b a a b c a c d a b c a b c b a d
a b c a b c d b a a c d
```

Obviously, there are triple of patterns evident in the examples, which is `a b c` in the first and second sequence. Patterns which can be recognized when analyzing both examples together are `d b a` and `a c d`. The sequences are analyzed with `mseq -i sequence1 -n 3`, which generates these grammars:

1. Grammar (2 rules):
S → \$A d b a \$A a c d \$A \$A b a d
\$A → a b c

2. Grammar (2 rules):
S → \$B \$B d b a a c d
\$B → a b c

The merging process takes the first grammar as starting point for unifying them. When the first rule of the second grammar G is analyzed, it is detected, the rhs is not existing in the universal grammar UG. Since this rule is the start rule in G, it is renamed and added as alternative to the start rule of UG:

S → \$A d b a \$A a c d \$A \$A b a d |
\$C
\$A → a b c
\$C → \$B \$B d b a a c d

The reader might note, that the referenced non-terminal \$B is not yet contained in UG. This is fixed after reading the next rule from grammar 2. Searching the body of this rule in UG, the algorithm detects that their rhs matches and no start rule is involved. Following the algorithm's directive for that case, each occurrence of the lhs of UG's rule is substituted by the name of the rule of G, which generates this intermediate grammar:

S → \$B d b a \$B a c d \$B \$B b a d |
\$C
\$B → a b c
\$C → \$B \$B d b a a c d

Next, the n-gram uniqueness for the unified grammar is checked. The procedure for performing this task in MSEQ is very similar to the procedure specified for the original SEQUITUR, expect for the facts that (1) it is executed after the unified grammar is generated completely and (2) for all rules in UG all n-grams are generated. The procedure takes a rhs and scans it for n-grams from the left to the right. In the example, the first 3-gram, which occurs multiple times, is the pattern "$B d b". Hence, a new rule $D $\rightarrow$ $B d b is generated and each occurrence of its rhs in UG is substituted by $D. Scanning the rules further, only one more pattern of length 3, which is "a c d", is repeated. Thence, the introduction of the rule $E $\rightarrow$ a c d is triggered. This leads to the following intermediate result:

```
S  → $D a $B $E $B $B b a d |
     $C
$D → $B d b
$E → a c d
$B → a b c
$C → $B $D a $E
```

The last step of multiple grammar merging is the standard rule utility check. Only the rule $C is used once. It is deleted and its occurrence in the start rule is replaced by its body, which gives the final grammar:

```
S  → $D a $B $E $B $B b a d |
     $B $D a $E
$D → $B d b
$E → a c d
$B → a b c
```

From the examples we can see, that MSEQ is able to detect regularities spread over different sequences. The algorithm produces a comprehensible grammar, which is complete. Many of the patterns are abstracted to rules, thus helping to understand the behavior producing the patterns.

However, due to the intermediate results some effects may be sometimes surprising. When we look at the two separately produced grammars, the 3-gram d b a is repeated, too. However, it is not directly visible in the final unified grammar. This is due to the left-to-right scanning of a rule's rhs leading to patterns containing non-terminals as well. In this example, the structure of the sequences determines, that `d b a` is not understood as pattern of its own, because the sequence `a b` was already a part of the previously captured pattern `a b c d b`. Both views are correct. △

The quality of the result depends heavily on the structure of the underlying event sequences. Two factors are decisive, which are (1) the number of repetitions and (2) the length of the loops. This second aspect determines the structure of the unified grammar obtained by applying MSEQ to the test cases of the revolving credit component (example 7.1 on page 157). Here, the length of loops is 1, and the number of test cases is too small to include larger patterns of behavior.

Example 7.8. MSEQ *applied to the* RevolvingCredit *sequences* ▽
The test sequences are analyzed with `mseq -i inputfile -n 2 -m`, which is explained with the repetition length of 1. Due to the short input sequences and their low degree of repetition most of the grammars infered are the trivial one: the input word.

1. Grammar (1 rule(s)):
S → `C V G B P X E`

2. Grammar (1 rule(s)):
S → `C A R`

3. Grammar (1 rule(s)):
S → `C A F V G B B X P E`

4. Grammar (1 rule(s)):
S → `C A F R`

5. Grammar (3 rule(s)):
S → `C F V G` $E $E $D `X E`
$D → `B P`
$E → $D `P`

6. Grammar (1 rule(s)):
S → `C A A F V G B X P E`

7. Grammar (1 rule(s)):
S → `C V A G B P X E`

8. Grammar (1 rule(s)):
S → `C R`

9. Grammar (1 rule(s)):
S → `C A V G B X P E`

10. Grammar (1 rule(s)):
S → `C A F V A R`

11. Grammar (1 rule(s)):
S → `C V R`

Merged Grammar:

```
S → $M $Z |            $D → B P
    $S R |             $X → B X P E
    $S $R B $X |       $E → $D P
    $U R |             $M → C V
    C $R $E $E $D X E | $U → $S F
    $S A $R $X |       $N → V G
    $M A $Z |          $R → F $N
    C R |              $Z → G B P X E
    $S $N $X |         $S → C A
    $U V A R |
    $M R
```

For helping to understand the underlying regularities, this grammar is apparently dissatisfying. MSEQ tries to find repeated patterns of length 2 which are not the main characteristics of the sequences. Some of these regularities are evident anyhow. Most of the start rule alternatives begin with the rule $S, indicating the sequence *CreditRequest Assessment* as a common order of events. This is clearly a main part of the component's behavior.

Because of the poor number of repetitions many base rules, containing only terminals on their right hand sides are produced. However, this is a positive side of the result. If a searcher searches for that component, she/he may not start at looking for large grain patterns, but to start a bottom-up search. While browsing through base rules, the understanding of the behavior is supported much better, than browsing top-down. △

The implementation of MSEQ offers two more parameters. The first, `-a`, is used for explaining the genesis of rules by skipping the step of rule utility check. In that way, intermediate rules are not deleted and all patterns which are found during the n-gram uniqueness check are visible further on. The second parameter `-v` (verbose) causes the algorithm to report about intermediate steps.

Evaluation of MSEQ

Regular positive inference conducted with MSEQ is a promising approach, if the structure of the event sequences shows regularities clearly. Then, due to various parameters, the algorithm can be adapted to various needs. The length of the searched pattern, as well as the frequency of repetitions can be tuned. Furthermore, which is the most significant enhancement to the basic SEQUITUR a treatment of multiple sequences is offered.

The algorithm MSEQ generates rules for all occurring sequences. As a consequence, information is not omitted. Thus, with respect to the quality properties presented in section 7.2 on page 162, it is *correct* with respect to the given sequences. Furthermore, MSEQ does not produce spurious rules, which is also the case when merging is performed. The algorithm does not accept more words than are given as examples. Merging finds similarities in different sequences, but does not "interbreed" them, in the way that a new rule comprises parts of different other rules (which could introduce spurious rules). However, if the algorithm is asked to produce iteration abstractions, the number of sequences such producible is infinite. Base SEQUITUR is not able to produce an infinite word.

In contrast to *k*TAIL we rate the *veracity* of MSEQ to be higher, since from a structural point of view, only identical subsequences are produced iteratively, whereas with *k*TAIL nearly arbitrary patterns may be generated. Regarding *minimality*, MSEQ is restricted by the parameter n determining the minimal length of the patterns to be recovered. Anyhow, we rate MSEQ in front of *k*TAIL, since the former can be adapted better to a domain and, therefore, generates lesser rules in the average.

As opposed to *k*TAIL, with respect to correctness, veracity, and minimality the quality of the results produced with MSEQ is rated higher. However, the representation concept produced is problematic and depends on the structure of the underlying examples. Formal languages expressed as context free grammars are readable, but when the language such expressed is large, the concept expressed may not be detectable easily. For application engineers the "first glance" comprehensibility of finite state automata is rated better. The reader might verify this statement by comparing the outcome of the revolving credit analysis performed with *k*TAIL (example 7.3 on page 167) with that of MSEQ (example 7.8).

7.5 Final remarks to ABS methods

In contrast to state-less components analyzing state-bearing and/or complex components is based on different assumptions and aims at different goals. Due to the complex interface, the step to determine signatures for them is not crucial, since a partition of a repository described by signatures would hold a rather small number of components. Thus, browsing through the detected behavior is not the main goal of the descriptions developed in this chapter. The main goal is to verify if a identified component is performing in the way as it is promised by the conventional description. Therefore, the ABS methods (MSEQ and *k*TAIL) describe particular components in isolation and do not distance themselves from other candidates. The way, how a repository for complex components is organized in detail, is not influenced by the description obtained with ABS methods.

Nevertheless, rudimentary browsing can be supported for MSEQ. Grammar rules are a partially description of a special aspect of the component. Due to the hierarchical description and the left to right analysis of input words, a top down approach is not meaningful. The start rules, although containing main characteristics, are in general abstract representations. Without refining the referenced rules in its body, a searcher cannot infer detailed behavior from it. On the other hand, when the base rules (rules with only terminal symbols in the body) are examined, the situation changes. Here, building blocks of the behavior can be detected immediately, such as "the credit request event always proceeds the assessment event `C A`", or "`X E`" (eXpire, Endorse) is a common sequence. In such a way, the next level of rules (non-terminals and terminals mixed) is investigated. Naturally, such minimal *bottom-up* behavior browsing is restricted to one component only, since the names of the component's methods are not standardized throughout the repository. This renders impossible a browsing technique as counterpart of the one suggested for SBS.

The situation for *k*TAIL is similar. As the underlying alphabet (the event names) is not standardized, the structure of finite state automata is the only clue to generalize its behavior. Here we suggest to group together FSAs according to the number of their strongly connected components, since such subgraphs hint to subprocesses of distinguishable functionality. In that way, a very rough grained structure for components based on their FSA-descriptions can be produced by dividing the repository into complexity partitions. Complexity is described by the number of subprocesses performed by its components. This approach demands a very high level of approximation of behavior from a searcher. At least, this *top-down* search strategy helps to separate monolithic-behavioral components (no subprocesses) from transaction-oriented behavioral components.

To conclude, both techniques are equally capable of handling the problem of generating descriptions automatically. Which is to chose in an individual case depends on

- the repetitive structures in the test sequences,

- the necessity for a hierarchical representation, and
- the familiarity of searchers with the resulting representation.

Hence, MSEQ and kTAIL do not replace each other, but are used as equally valid analysis and description techniques.

Apparently, MSEQ and kTAIL are applicable for other purposes than producing descriptions for reusable components as well. In [196] both techniques (in their base form) are used in the field of program understanding. Here, trace data produced by debugging a running program is such depicted comprehensibly. We see potential in other fields too, especially in reengineering legacy code and for maintenance purposes.

7.6 Summary

In this chapter we discussed two different techniques for realizing ABS methods (abstracted behavior sampling) to encode patterns one can identify in test cases. They generate descriptions automatically for reusable complex and state-bearing components. All ABS algorithms developed in this chapter were implemented from scratch. Presented with the example of object-oriented components, the problem of positive regular inference is stated, which deals with the regeneration of behavior hidden in sequences of examples. Positive regular inference is characterized by the knowledge, that a general solution is not possible. Hence, heuristics must be introduced to render an efficient technique possible.

The first algorithm analyzed is kTAIL, whose idea is to infer states from the component's produced words (event sequences). Words are used to build finite state automata (FSAs). For application engineers, FSAs are a usual description of a software's behavior, thus, this technique is promising. A state is defined as a prefix of words, from which a common partial sequence (future) of length k can be derived. We improved the basic algorithm (1) by adding explicit initial states and end states and (2) by clustering strongly connected subgraphs. Both adaptations help to raise the descriptive capability of the resulting FSAs. Under the assumption of a correct selected observation window of length k the kTAIL algorithm generates understandable descriptions, if the underlying set of sequences does not show to much variations. The small degree of freedom offered by the algorithm's only parameter is considered to be problematic. If e.g. the domain shows different lengths of repetitions, such an inflexible look into the k future of a word's prefix may not be sufficient.

The second algorithm presented is MSEQ, which is based on SEQUITUR, a technique developed to encode patterns contained in a single input sequence. MSEQ generates context free grammars, describing the regular language produced by a set of sequences. Our adaptations can be divided into two groups:

1. Implementing flexibility to allow the adaptation to domain requirements. The first adaptation is to parameterize the *length* of the patterns to be detected. Some domains show patterns of an explicit length. If this is known, the algorithm provided with this specific information is able to generate a grammar reflecting the underlying sequences better. The second adaptation enhances the algorithm to abstract from the given word to a more general form. It is based on the assumption, that a consecutive occurrence of a pattern of arbitrary length points to an *iteration* in the component producing the sequence. Such a repetition is compressed to an iterative representation in the resulting grammar.
2. Abstracting from *multiple sequences* to a common description. Base SEQUITUR is able to exploit the information of a single sequence only. We enhanced it to analyze patterns which are spread across different sequences, resp. the grammars inferred from them. In that way, a description for all analyzed sequences is produced.

The contribution of this chapter is its approach to abstract from simple input value transformation to the order dependencies introduced by complex and state-bearing components. To reach that aim we designed and implemented enhancements of known sequence analysis algorithms to render them more flexible for our needs. Although browsing is not (and need not be) supported, the descriptions such produced automatically from test data are useful for understanding and verifying the behavior of reusable components.

8 Conclusion

In this work techniques were developed for describing reusable components automatically. With respect to the analysis techniques, components are divided into two different classes:

State-less components: Components of this class depend on input data only, when computing its output. In chapter 6, the *static behavior sampling* technique (SBS) was developed for analyzing the behavior demonstrated by components.

State-bearing and/or complex components: Behavior demonstrated by these components does not only depend on input but on internal states, too. In chapter 7 the *abstracted behavior sampling* methods (ABS) were developed for the treatment of this class of components. ABS is well suited for components whose interfaces are too complex to be described by simple input-output specifications.

Although both techniques start from similar ground, they are different in nature as far as their aims are concerned. SBS supports the reuse of state-less components by providing an indexing structure. The information contained in a component's interface is transformed into an form independent from implementation – the component's signature. How signatures serve as a means to describe structural aspects of a components interface is discussed in chapter 5. Signatures are then used to partitionize a repository of reusable components.

On the basis of a searcher's anticipatory knowledge, she/he formulates a query which finally results in the selection of the most promising SBS-partition. A query consists of a signature description of the wanted component and it may be relaxed by allowing various signature forms as valid result. Such valid forms could be subtypes or name equal signatures, constitutive equal ones, or structural equal ones. Having selected a partition, the SBS-method guides the searcher through a browsing process by iteratively offering examples as partial specifications of the searched functionality. In most cases, it is not necessary to know exactly what functionality a component performs as long as it can be distinguished from other components within the partition.

The intention of ABS methods is different. Like in SBS, the descriptions are automatically produced from test data. But due to the generally highly specialized nature of components, the emphasis of the proceeding task is on the verification of an initial hypothesis. Such a hypothesis is given by standard descriptions. Reuse is supported by easing the task of providing non-ambiguous descriptions automatically which are more easily understandable by application engineers.

8.1 Evaluation of this work

8.1.1 Challenges addressed

Various problems have been addressed in this thesis. They stem from the changes evoked by globally distributed and separated software development processes which consequently induces separated communication forms. These challenges have been discussed in detail in section 3.2. They are reiterated here and for each challenge we provide the results which have been achieved by this work:

Challenge 1: (page 31) How can people working in the context of decoupled software development processes generate component descriptions which are interpreted in completely identical ways?

Data points and test sequences are tightly related to the functionality of components. They stem directly from the problem domain. This does not need to be the case with names and textual descriptions, which can be chosen arbitrarily. Furthermore, SBS, *k*TAIL and MSEQ depend much more on the quality of data points and input sequences than on the capability of experts to tune the algorithms correctly to their particular needs. Therefore, the description is neither influenced by the software process nor does it depend on a profound understanding of the application domain.

Challenge 2: (page 33) How can a search technique support an application engineer in such a way that she/he doesn't need extensive training to access the repository effectively?

The seamless integration of reuse techniques into every-day working processes is a main success criterion for reuse. Opportunistic reuse (especially reuse-by-anticipation) can only work if there is no substantial extra effort needed to locate a component. With the techniques developed in this work, the burden of generating abstractions of components is completely shifted from humans to algorithms. Neither domain engineers nor application engineers need to be trained to produce correct abstractions.

With SBS the description is not an abstraction of the components' behavior, since it can be characterized in terms of a guidance for locating components. In spite of the abstract results produced by ABS, there are no restrictions as far as its comprehensibility is concerned. Quite on the contrary, ABS represents concepts which are well familiar to software engineers.

Challenge 3: (page 34) How can a describer be supported in producing a stable description which allows

- to index and maintain a library without losing the description's expressiveness (concept) and
- to perform effective searches in the query space?

One of the main problems when a repository is changing over time is the shift of important notions used for keywords. Librarians are then faced with introducing new keywords which did not exist at the time the repository was designed. SBS and ABS based repositories are not connected with keywords, and evolving concepts do not change the classification structure. As a matter of fact, a repository changes over time and adapting the partition of an SBS repository is a recurrent task. However, the librarian is not bothered with the task of generating new classification structures. Generalized signature matching helps to infer partitions automatically from the new component's interface. Even the browsing structure is adapted semi-automatically, after the maintenance strategy has been decided.

With ABS the situation is different, since the descriptions obtained are specifications of individual components which do not aim at distinction among them. How a repository of complex components itself is organized in detail is not discussed in this work.

For generating descriptions, specialists are not needed anymore: Due to the nature of the inference process the descriptions reflect the behavior of the components. Support for searching is provided

- by structurally motivated partitions of the repository and a browsing structure (SBS), and
- by presenting the behavior of a component on a conceptual level (ABS).

Challenge 4: (page 35) How can component descriptions which precisely reflect their functionality be generated automatically?

This is done by viewing data points and test sequences as partial specifications. Departing from the assumption that the data on which the behavioral knowledge base is built reflects the characteristics of its components sufficiently, the algorithms are able to extract that knowledge and to present it to the searcher.

Challenge 5: (page 36) How can it be ensured, that a component's description only depends on its functionality and not on the describer's personal, social, and cultural context?

The fundamentals for a high quality SBS- and ABS-description rest with the quality of data points and traces. However, even if these data is chosen wrongly, the resulting representation of the components behavior is not mistaken. If the discriminative and descriptive power may be small, the effective localization of a component is severely hampered. In such a case browsing supports the localization of components to a certain extent. But if the underlying knowledge base of test cases (data points and traces) is sufficiently large, such a case should not happen. Then, due to the automatic generation of descriptions and the restricted tuning possibilities, this challenge is fulfilled. The tuning of the algorithms does not change the interpretation of the results, only the clarity and readability depend on it.

Challenge 6: (page 36) How can a component be described non-ambiguously in a way which is natural for software developers?

Input-output transformations, finite state machines and formal grammars are part of the every day life of software developers. The resulting structures are non-ambiguous with respect to the quality of data points, resp. test sequences.

Challenge 7: (page 37) How can a reuser verify quickly, whether components behave in an unexpected way?

In SBS this verification is part of the browsing process which performs like an iterative query and answer game. It is not that easy with state-bearing and complex components, since the alphabet of the formal languages must be interpreted. With this type of components, the aim is not to restrict the candidate set step-by-step, because it is small enough. Here, the verification step is much more important. If the question is reformulated to "How can the reuser verify quickly, whether a component behaves in an *expected* way?", then the answer is: "*k*TAIL and MSEQ support this task by providing abstractions of correct behavior". If the reuser is able to recognize inappropriate behavior, then she/he can answer the question quickly thanks to the results delivered by ABS.

All techniques presented in this work give an insight into the behavior of components without overwhelming the searcher with too much detail. SBS and MSEQ allow a hierarchical, top-down refinement of the search as well.

8.1.2 Drawbacks

Under certain circumstances SBS and ABS methods are not appropriate for describing components. If the data structures were too complex to be understood, behavior based sampling would not support reuse.

1. Components, which transform multimedia streams or
2. software, which implements data compression algorithms, or
3. complex human-computer-interfaces

are representative examples for reusable components which cope with such complex data structures. In such cases the details of input-output transformations would confuse the searcher.

Here the field of data visualization plays an important role. If the data and the transformations performed can be represented in a compact and comprehensive form, SBS and ABS methods have ground to stand on. But then the interpretation of the concepts of representation is shifted to humans again. This once more generates some of the problems discussed in this work; problems which were intended to be solved by SBS and ABS. In such cases the question arises whether or not SBS and ABS methods were adequate for describing and retrieving components. All components mentioned above are very specialized and a textual description may be sufficient to characterize them.

A further area of complex components is excluded from a behavior based description. If there are *no step-by-step* activities visible in the test cases, generating an abstraction of activity sequences as grammar or FSA would not help. The main requirement for a successful analysis based on ABS methods is an existing and meaningful order of events produced or emitted by the software. If the length of repeating patterns is beyond the capability of human observations or if the sequence is produced randomly, then the methods presented in this work would not be the right choice.

8.2 Future work

Data preparation

We concentrated on the generation of descriptions of reusable components on the basis of behavior which is provided by data points (selected from test data) and by test sequences. The necessary quality of the test data is discussed in chapter 6, where some suggestions are made about how to obtain them. An important step, which was not discussed in this thesis, would be to define input filters for test data in order to prepare them for SBS and ABS. At first sight, this seems to be an easy task. However, many components demand data as input and/or produce output, which can only be treated on a symbolic level. File access, device control or network transmissions are examples of such a data. Thus, in front of our tool set, we need a layer transforming *non-representable* structures to input which can be handled and output which can be represented.

Similar to an input-output preparation step, non-representable structures must be adequately visualized. In this work, the results of the knowledge mining activities are represented in a quite modest quality. We need a further step to visualize and abstract complex results. The aim of this step would be to prepare the data for human cognition in such a way that its input-output characteristics would be easier to grasp.

Tool set

The algorithms developed are implemented prototypically and hence there is more work left to be done. As the emphasis of the implementations was not on efficiency, the algorithms should be redesigned to meet the needs of an integration into a production environment. A further task would be a fusion of the tools into one homogeneous and seamless tool set.

Process integration

SBS and ABS have autonomous status and can be used as stand-alone techniques. But better effects could be achieved if they were integrated into one software development process as a supportive technical canon. To work out the details of this process, its preconditions and the requirements, in which SBS and ABS were an integrated part of a life cycle model, would be an important task which should be pursued in future.

8.3 Synopsis

The idea of software reuse was introduced nearly 35 years ago. Researchers have correctly identified the field as a complex mixture of technical, organizational and managerial challenges. In this thesis, the technical aspect of generating non-ambiguous descriptions with data mining techniques has been worked out. The utilized data has been selected from test data. At first sight it might appear as detouring from test cases to descriptions, but in order to obtain non-ambiguous specifications which are easy to understand, this strategy is well justified.

In the literature, an approach similar to SBS and ABS on the basis of natural language can be found (e.g. see [32, 33, 35]). In their work, textual software documentations are analyzed with the aim to produce automatically obtain significantly shorter abstractions. Here, some restrictions, such as the limitation to use only texts from a specific domain, to be confined to inter-sentence analysis, or to rely on specifically annotated texts have to be considered. SBS and ABS are not restricted in that way, but, as already discussed, they reveal other drawbacks. Therefore, the test data approach and the natural language approach render alternative techniques which may support each other well.

The techniques developed in this thesis are not exclusively designed for software reuse. E.g. mining for the semantics of software is the subject of reengineering as well, due to the fact that legacy code is analyzed and its behavior must be verified. Legacy code is part of an existing and running system producing many input-output transformation which are equivalent to test cases. SBS and ABS are practical for determining the functionality of a system without the need to dig into its source code. This idea has already been considered in the work of [182], where a library of reusable components delivers reference descriptions. Chunks which are extracted from legacy components are matched against the reference descriptions by using its data points. If a chunk demonstrates the same behavior as a component, then the component's description is valid for the chunk, too. When they do not match, as a positive side effect, the chunk can be considered as a candidate for reuse, since its functionality was not contained in the repository yet. Apart from its importance in software reuse this application is also an example for a usage of behavior-based techniques.

Thus, SBS and ABS are easy-to-use tools for verifying hypotheses about executable software in all those cases in which a standard documentation of software is not sufficiently trustworthy.

Bibliography

[1] AGRAWAL, R., GUNOPULOS, D., AND LEYMANN, F. Mining Process Models from Workflow Logs. In *6th International Conference on Extending Database Technology - EDBT'98, Valencia, Spain* (March 1998), H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, Eds., vol. 1377 of *Lecture Notes in Computer Science*, Springer, pp. 469 – 483.
. . . . 162, 163, 167

[2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers – Principles, Techniques, and Tools*, 1st ed. Addison-Wesley Publishing, 1986.
. . . . 76

[3] ALBRECHTSEN, H. Software Information Systems: information retrieval techniques. In Hall [90], ch. 6, pp. 99–127.
. . . . 33

[4] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Information and Computation 2*, 75 (1987), 87–106.
. . . . 162

[5] ARANGO, G. Domain Analysis – From Art to Engineering Discipline. In *Proceedings of the 5th International Workshop on Software Specificaions and Design* (1989), Association for Computing Machinery, pp. 152–159.
. . . . 13

[6] ARANGO, G., AND PRIETO-DIÀZ, R. *Domain Analysis and Software Systems Modeling*. Vol. 1 of Prieto-Diàz [186], 1991, ch. Domain Analysis Concepts and Research Directions.
. . . . 13, 49, 51

[7] ARMOUR, P. G. The Case for a New Business Model - is software a product or a medium. *Communications of the ACM 43*, 8 (August 2000), 19–22.
. . . . 47

[8] ATKINSON, S. Cognitive Deficiencies in Software Library Design. In *Proc. of Asia-Pacific Software Engineering Conference and International Computer Science Conference* (December 1997), IEEE Computer Society Press, pp. 354–363.
. . . . 27

[9] ATKINSON, S. Examining Behavioural Retrieval. In Latour [124].
. . . . 54

[10] ATKINSON, S., AND DUKE, R. A Methodology for Behavioural Retrieval from Class Libraries. *Australian Computer Science Communications 17*, 1 (January 1995), 13–20.
. . . . 54

[11] BAAR, T., FISCHER, B., AND FUCHS, D. Integrating Deduction Techniques in a Softare Reuse Application. *Journal of Universal Computer Science 5*, 3 (March 1999), 52–72.
. . . . 19, 56

[12] BACKUS, J. The History of Fortran I, II, and III. *IEEE Annals of the History of Computing 20*, 4 (October 1998).
. . . . 8

[13] BAEZA-YATES, R., AND RIBEIRO-NETO, B. *Modern Information Retrieval*. Addison Wesley, 1999.
. . . . 36, 39, 44

[14] BANCILHON, F., DELOBEL, C., AND KANELLAKIS, P., Eds. *Building an object-oriented database system : the story of O2*. The Morgan Kaufmann series in data management systems. Morgan Kaufmann, 1992.
. . . . 89

[15] BASILI, V. R., AND ABD-EL-HAFIZ, S. K. A Method for Documenting Code Components. *Journal of Systems Software 34*, 2 (August 1996), 89–104.
. . . . 35, 36, 54, 56

[16] BASILI, V. R., AND BOEHM, B. COTS-Based Systems Top 10 List. *IEEE Computer 34*, 5 (May 2001), 91–93.
. . . . 154

[17] BASILI, V. R., BRIAND, L. C., AND MELO, W. L. How Reuse influences Productivity in Object-Oriented Systems. *Communications of the ACM 39*, 10 (October 1996), 104 – 116.
. . . . 9, 155

[18] BASILI, V. R., AND ROMBACH, H. D. Support for Comprehensive Reuse. *Software Engineering Journal, IEE British Computer Society 6*, 5 (September 1991), 303–316.
. . . . 11

[19] BASS, L., AND KAZMAN, R. Architecture-Based Development. Tech. Rep. CMU/SEI-99-TR-007, Carnegie Mellon Software Engineering Institute, CMU/SEI Pittsburgh, PA 15213-3890, April 1999.
. . . . 22

[20] BATORY, D. Product-Line Architectures, Generators, and Reuse. Professional Development Seminar at SSR99 Los Angeles, CA, May 1999.
. . . . 16

[21] BEIZER, B. *Black Box Testing – Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.
. . . . 120, 121

[22] BELLETTINI, C., DAMIANI, E., AND FUGIN, M. G. User opinions and rewards in a reuse-based development system. In *Proceedings of the 5th Symposium on Software Reusability – SSR'99* (Los Angeles CA USA, 1999), ACM Press, pp. 151–158.
. . . . 16

[23] BELLETTINI, C., DAMIANI, E., AND FUGINI, M. G. Automatically Polling User Opinions to Tune the Effectiveness of a Software Reuse System. In Wang [227], pp. 443–449.
. . . . 16

[24] BEN-SHAUL, I., AND KAISER, G. Coordinating distributed components over the internet. *IEEE Internet Computing 2*, 2 (March 1998), 83–86.
. . . . 17

[25] BIERMANN, A. W., AND FELDMAN, J. A. On the Synthesis of Finite State Machines from Samples of their Behavior. *IEEE Transactions on Computers 21*, 6 (June 1972), 592–597.
. . . . 163

[26] BLASCHEK, G. Objektorientierte Programmierung. In Rechenberg and Pomberger [194], ch. 4, pp. 529–552.
. . . . 155

[27] BLOEM, R., GABOW, H. N., AND SOMENZI, F. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Formal Methods in Computer Aided Design*, J. W. A. Hunt and S. D. Johnson, Eds., LNCS 1954. Springer-Verlag, November 2000, pp. 37–54.
. . . . 167

[28] BOEHM, B. A Spiral Model of Software Development and Enhancement. *IEEE Computer 21*, 5 (May 1988), 61–72.
. . . . 29

[29] BOEHM, B. Managing Software Productivity and Reuse. *IEEE Computer 32*, 9 (September 1999), 111–113.
. . . . 15

[30] BOEHM, B., AND ABTS, C. COTS Integration: Plug and Pray? *IEEE Computer 32*, 1 (January 1999), 135–138.
. . . . 13, 154

[31] BOEHM, B. W. *Software Engineering Economics*. Prentice Hall, 1981.
. . . . 9

[32] BOUCHACHIA, A. *Information retrieval techniques for software retrieval*. PhD thesis, Klagenfurt University, Austria, 2001.
. . . . 4, 195

[33] BOUCHACHIA, A., AND MITTERMEIR, R. T. Coping with Uncertainty in Software Retrieval Systems. In *Proceedings of the 2nd International Workshop on Soft Computing Applied to Software Engineering (SCASE'01)* (Enschede, The Netherlands, February 2001), pp. 21–30.
. . . . 50, 195

[34] BOUCHACHIA, A., AND MITTERMEIR, R. T. Feature Selection for an Adaptive Categorization. In *Proceedings of the International Conference on Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing (SNPD'01)* (Nagoya, Japan, August 2001). 50

[35] BOUCHACHIA, A., MITTERMEIR, R. T., AND POZEWAUNIG, H. Document Identification by Shallow Semantic Analysis. In *Proceedings of the 5th International Conference on Applications of Natural Language to Information Systems (NLDB'2000)* (Versailles – France, June 2000), Springer Verlag. 195

[36] BRERETON, P., AND BUDGEN, D. Component-Based Systems: A Classification of Issues. *IEEE Computer 33*, 1 (November 2000), 54–62. 20

[37] BROWN, A. W., AND WALLNAU, K. C. The Current State of CBSE (Component based Software Engineering. *IEEE Software 15*, 15 (September 1998), 37–46. 21

[38] BUCCI, P. A Program Editor to Promote Reuse. In Latour and Wentzel [127]. 32

[39] BUCCI, P. Conceptual Program Editors. In Latour [124]. 32

[40] BURBECK, S. Using Signatures to Improve Smalltalk Productivity and Reuse. ftp.sunet.se/pub7/lang/smalltalk/Squeak/docs/-SmalltalkSignatures.htm, 1995. (current February 16th, 2000). 114

[41] BURD, E., AND MUNRO, M. A method for the identification of reusable units through the reengineering of legacy code. *The Journal of Systems and Software 44*, 2 (December 1998), 121–134. 13

[42] CARBONELL, R. S. M. J. G., AND MITCHEL, T. M., Eds. *Machine Learning: An Artificial Intelligence Approach*, vol. 1. Tioga Publishing

Company, Palo Alto, CA, USA, 1983.
. . . . 214, 219

[43] CARDELLI, L. A Semantics of Multiple Inheritance. *Information and Computation 76* (1988), 138–164.
. . . . 89

[44] CARDELLI, L. Structural Subtyping and the Notion of Power Type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages* (January 1988), ACM Press, pp. 70–79.
. . . . 68, 86, 87, 88

[45] CARDELLI, L., AND WEGNER, P. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys 17*, 4 (December 1985), 471 – 523.
. . . . 67, 68, 69, 86

[46] CASTAGNA, G. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems 17*, 3 (1995), 431–447.
. . . . 89

[47] CHEN, Y., AND CHENG, B. H. C. Facilitating an Automated Approach to Architecture-based Software Reuse. In *Proceedings of the 12th International Automated Software Engineering Conference* (Nevada, USA, 1997), pp. 238–245.
. . . . 58

[48] CHEN, Y., AND CHENG, B. H. C. Formalizing and Automating Component Reuse. In *Proceedings of 9th International Conference on Tools with Artificial Intelligence – TAI 97* (Newport Beach, California, November 1997), pp. 94 – 101.
. . . . 54

[49] CLEMENTS, P. C., AND NORTHROP, L. M. Software architecture: an executive overview. In *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*, A. W. Brown, Ed. IEEE Computer Society Press, 1996, pp. 55–68.
. . . . 21

[50] CLERMONT, M. Generalisierter Signaturvergleich als Mittel zum Software Retrieval. Master's thesis, Universität Klagenfurt, Austria, 1999.
. . . . 57

[51] COOK, J. *Process Discovery and Validation through Event-Data Analysis*. PhD thesis, Software Engineering Research Laboratory, Department of Computer Science, University of Colorado, Boulder, CO, USA, November 1996.
. . . . 163, 165

[52] COOK, W. R., HILL, W. L., AND CANNING, P. S. Inheritance is not Subtyping. In *7th Annual ACM Symposium on Principles of Programming Languages* (San Francisco, California, January 17–19, 1990), ACM Press, New York, pp. 125–135.
. . . . 87, 89

[53] COWARD, P. D. A Review of Software Testing. In Dorfman and Thayer [63], pp. 277 – 286. Reprinted from *Information and Software Technology*, Vol. 30, No. 3. Apr. 1988, P. David Coward, "A Review of Software Testing".
. . . . 120

[54] COX, I. J., MILLER, M. L., MINKA, T. P., PAPATHOMAS, T. V., , AND YIANILOS, P. N. The Bayesian Image Retrieval System, PicHunter: Theory, Implementation, and Psychophysical Experiments. *IEEE Transactions on Image Processing, Special Issue on Image and Video Processing for Digital Libraries 9*, 1 (2000), 20–37.
. . . . 114

[55] COX, I. J., MILLER, M. L., OMOHUNDRO, S. M., AND YIANILOS, P. N. Target Testing and the PicHunter Bayesian Multimedia Database Retrieval System. In *Proceedings of the 3rd Forum on Research and Technology Advances in Digital Libraries (ADL'96)* (1996), pp. 66–75.
. . . . 114

[56] CREECH, M. L., FREEZE, D. F., AND GRISS, M. L. Using Hypertext in Selecting Reusale Software Components. In *Proceedings of the 3rd Conference on Hypertext* (San Antonio, TX, USA, 1991), ACM Press, pp. 25–38.
. . . . 53

[57] DAMIANI, E., AND FUGINI, M. G. Fuzzy techniques for software reuse. In *ACM Symposium on Applied Computing* (1996), pp. 552 – 557. 32

[58] DAVIS, R., SHROBE, H., AND SZOLOVITS, P. What is a Knowledge Representation? *AI Magazine 14*, 1 (1993), 17–33. . . . . 48, 49

[59] DEVANBU, P., BRACHMAN, R. J., SELFRIDGE, P. G., AND BALLARD, B. W. LaSSIE: a Knowledge-based Software Information System. *Communications of the ACM 34*, 5 (May 1991), 35–49. 50

[60] DEVANBU, P. T., PERRY, D. E., AND POULIN, J. S. Next generation software reuse. *IEEE Transactions on Software Engineering 26*, 5 (May 2000), 423–424. 8

[61] DILLER, A. *Z : An Introduction to Formal Methods*, 2nd ed. Wiley, 1999. 69

[62] Software Reuse Executive Primer. DOD Software Reuse Initiative Program Management Office, (`http://dii-sw.ncr.disa.mil/reuseic/pol-hist/primer/`, current May, 2000), April 1996. 9

[63] DORFMAN, M., AND THAYER, R. H., Eds. *Software Engineering*, vol. 1. IEEE Computer Society Press, Los Alamitos, California, Washington, et. al., 1997. . . . . 203, 208

[64] DORN, J., AND GOTTLOB, G. *Künstliche Intelligenz*, 2nd ed. Vol. 1 of Rechenberg and Pomberger [194], 1999, ch. 7, pp. 975–998. 49

[65] DUPONT, P. Incremental Regular Inference. In *Proceedings of the Third ICGI-96* (Montpellier, France, 1996), L. Miclet and C. Higuera, Eds., vol. 1147 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 222–237. 162

[66] EDWARDS, S. Good Mental Models are Necessary for Understandable Software. In Latour and Wentzel [127].
. . . . 29

[67] FERBER, R. *Dokumentsuche und Dokumentenerschließung*, 2nd ed. Vol. 1 of Rechenberg and Pomberger [194], 1999, ch. 4, pp. 909–930.
. . . . 39

[68] FISCHER, B. Specification-Based Browsing of Software Component Libraries. In *13th IEEE Conference on Automated Software Engineering – ASE'98* (Honolulu, Hawaii, October 1998), D. Redmiles and B. Nuseibeh, Eds., IEEE CS Press.
. . . . 32, 54, 56

[69] FISCHER, B., AND SNELTING, G. Reuse by Contract. In *Proc. ESEC/FSE-Workshop on Foundations of Component-Based Systems* (Zürich, Suisse, September 1997), pp. 91–100.
. . . . 19, 227

[70] FISCHER, G., HENNINGER, S., AND REDMILES, D. Cognitive tools for locating and comprehending software objects for reuse. In *13th International Conference on Software Engineering – ICSE'91* (Austin, Texas, May 1991), IEEE CS Press, pp. 318–328.
. . . . 34

[71] FOWLER, M., AND KENDALL, S. *UML distilled*, 2nd ed. Addison-Wesley, 1999.
. . . . 157

[72] FRAKES, W., AND TERRY, C. Software Reuse: Metrics and Models. *ACM Computing Surveys 28*, 2 (June 1996), 415ff.
. . . . 8, 19

[73] FRAKES, W. B., Ed. *Third International Conference on Software Reuse – Advances in Software Reusability* (Rio de Janeiro, Brazil, November 1994), IEEE Computer Society Press.
. . . . 213, 216

[74] FRAKES, W. B., BIGGERSTAFF, T. J., PRIETO-DIÀZ, R., MATSUMURA, K., AND SCHAEFER, W. Software reuse – is it delivering? In *Proceedings*

of the 13th International Conference on Software Engineering (May 13 - 17 Austin, TX USA, 1991), ACM, pp. 52–61.
. . . . 15, 31

[75] FRAKES, W. B., AND FOX, C. J. Sixteen Questions About Software Reuse. *Communications of the ACM 38*, 6 (June 1995), 75 – 87.
. . . . 11

[76] FRAKES, W. B., AND GANDEL, P. B. Representation Methods for Software Reuse. *Conference Proceedings on Ada Technology in Context: Application, Development, and Deployment* (October 1989), 302–314.
. . . . 39, 47, 51, 52

[77] FRAKES, W. B., AND GANDEL, P. B. Representing Reusable Software. *Information and Software Technology 32*, 10 (December 1990), 653 – 663.
. . . . 47

[78] FREITAG, B., AND AVINI, K. A Hypertext-Based Tool for Large Scale Software Reuse. In *Proceedings of the 6^{th} Conference on Advanced Information Systems Engineering (CAiSE*94)* (Utrecht, The Netherlands, 6-10 June 1994).
. . . . 52

[79] GANTI, V., GEHRKE, J., AND RAMAKRISHNAN, R. Mining Very Large Databases. *IEEE Computer 32*, 8 (August 1999), 38–45.
. . . . 116, 128

[80] GAO, J. Z., CHEN, C., TOYOSHIMA, Y., AND LEUNG, D. K. Engineering on the Internet for Global Software Production. *IEEE Computer 32*, 5 (May 1999), 38–47.
. . . . 17, 36, 37

[81] GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. Architectural Mismatch: Why Reuse is So Hard. *IEEE Software 12*, 6 (November 1995), 17–26.
. . . . 22, 153

[82] GARLAN, D., AND SHAW, M. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering* (Singapore, 1993), V. Ambriola and G. Tortora, Eds., World Scientific Publishing

Company, pp. 1–39.
. . . . 21

[83] GAWECKI, A., AND MATTHES, F. Integrating Subtyping, Matching and Type Quantification: A Practical Perspective. In *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96* (Linz, Austria, 1996), Springer Verlag.
. . . . 85

[84] GHOSH, A. K., AND VOAS, J. M. Inoculating Software for Survivability. *Commun. ACM 42*, 7 (July 1999), 38–44.
. . . . 154

[85] GIBBS, W. W. Sofware's Chronic Crisis. *Scientific American* (1994), 86–95.
. . . . 8

[86] GOLD, E. Complexity of Automatic Identification from given Data. *Information and Control 10* (1978), 302–320.
. . . . 161, 162

[87] GORDON, D. A Graphical User Interface in Ada for Domain-Specific Reuse Libraries. In *Conference proceedings on TRI-Ada '92* (Orlando, FL USA, November 16 - 20 1992), pp. 309 – 320.
. . . . 32

[88] GRISS, M., ADAMS, S. S., BAETJER, H., COX, B. J., AND GOLDBERG, A. The economics of software reuse. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Aplications* (October 6-11 1991), ACM Association for Computing Machinery, pp. 264–270.
. . . . 9

[89] GROSSMAN, D. A., AND FRIEDER, O. *Information Retrieval : Algorithms and Heuristics*. The Kluwer International Series in Engineering and Computer Science. Kluwer, Boston, Mass. USA, 1998.
. . . . 41, 44

[90] HALL, P. A. V., Ed. *Software Reuse and Reverse Engineering in Practice*, 1 ed., vol. 12 of *Applied Information Technology 12*. Chapman & Hall,

London, New York, et. al., 1992.
. . . . 197

[91] HALL, P. A. V., AND JIN, L. The Re-engineering and Reuse of Software. In Dorfman and Thayer [63], pp. 444–460.
. . . . 8, 12

[92] HALL, R. J. Generalized Behaviour-based Retrieval. In *International Conference on Software Engineering – ICSE93* (Baltimore, MD, May 1993), IEEE Computer Society, IEEE Computer Society Press.
. . . . 62, 113

[93] HALLSTEINSEN, S., AND PACI, M., Eds. *Experiences in Software Evolution and Reuse*. Research Reports – Esprit. Springer Verlag, Berlin, Heidelberg, 1997.
. . . . xv, 8

[94] HENNINGER, S. Developing Domain Knowledge Through the Reuse of Project Experiences. In Samadzadeh and Zand [203], pp. 186–205.
. . . . 13, 49

[95] HENNINGER, S. Supporting the Construction and Evolution of Component Repositories. In *18th International Conference on Software Engineering - ICSE'96* (Berlin, FRG, 1996), pp. 279–288.
. . . . 49

[96] HENNINGER, S. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Transactions on Software Engineering and Methodology 6*, 2 (1997), 111 – 140.
. . . . 42, 49

[97] HERRMANN, J. *Maschinelles Lernen und Wissensbasierte Systeme*. Springer, Berlin, Heidelberg, 1997.
. . . . 129

[98] HILERIO, I. Herbal-t, enabling integration, interoperability, and reusability of internet components. *ACM SIGSOFT Software Engineering Notes 24*, 2 (March 1999), 49–58.
. . . . 36

[99] HITZ, M., AND KAPPEL, G. *UML@Work – Von der Analyse zur Realisierung*, 1st ed. dpunkt.verlag, Heidelberg, 1999.
. . . . 157

[100] HOCHMÜLLER, E. *AUGUSTA - Eine reuse-orientierte Software-Entwicklungsumgebung zur Erstellung von Ada-Applikationen*. PhD thesis, University of Klagenfurt, May 1992.
. . . . 11, 32

[101] HOCHMÜLLER, E. Software Reuse - it's TEA time! In Latour and Wentzel [127].
. . . . 12

[102] HOCHMÜLLER, E., AND MITTERMEIR, R. T. Rahmenbedingungen für erfolgreiches Software Reuse. In *Wiener IT-Kongress* (March 1993), ADV Wien, pp. 269–284.
. . . . 12

[103] HOPCROFT, J. E., AND ULLMAN, J. D. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1990.
. . . . 156, 160, 161, 166, 175, 179, 226

[104] HOPKINS, J. Component Primer. *Communications of the ACM 43*, 10 (October 2000), 27–30.
. . . . 20

[105] INGALLS, D., KAEHLER, T., MALONEY, J., WALLACE, S., AND KAY, A. Back to the future: the story of squeak, a practical smalltalk written in itself. In *Proceedings of the ACM SIGPLAN conference on Object-oriented programming systems, languages and applications* (Atlanta, GA USA, October 1997), pp. 318–326.
. . . . 114

[106] ISAKOWITZ, T., AND KAUFFMAN, R. J. Supporting Search for Reusable Software Objects. *IEEE Transactions on Software Engineering 22*, 6 (June 1996), 407–423.
. . . . 32, 52, 53

[107] JACKSON, M. The world and the machine. In *Proceedings of the 17th International Conference on Software Engineering* (1995), IEEE Computer

Society Press, pp. 283–292.
. . . . 60

[108] JACOBSON, I., GRISS, M., AND JONSSON, P. *Software reuse : architecture, process and organization for business success*. ACM Press books, 1997.
. . . . 14

[109] JARZABEK, S. From reuse library experience to application generation architectures. In Samadzadeh and Zand [203], pp. 114–122.
. . . . 29

[110] JENG, J.-J., AND CHENG, B. H. C. Specification Matching for Software Reuse: A Foundation. In Samadzadeh and Zand [203].
. . . . 54

[111] JILANI, L. L., MILI, R., AND MILI, A. Approximate Component Retrieval: An Academic Exercise or a Practical Concern? . In Latour [124].
. . . . 32, 55

[112] JOHNSON, L. J. A View From the 1960s: How the Software Industry Began. *IEEE Annals of the History of Computing 20*, 1 (1998), 36–42.
. . . . 7

[113] KARLSSON, E.-A., Ed. *Software Reuse: A Holistic Approach*. Series in Software Based Systems. John Wiley & Sons, Chichester, 1995.
. . . . 45, 228

[114] KATZ, S., DABROWSKI, C., MILES, K., AND LAW, M. Glossary of Software Reuse Terms. National Institute of Standards and Technology (NIST), NIST Special Publication 500–222, December 1994.
. . . . 12, 13, 20, 225

[115] KATZENELSON, J., PINTER, S. S., AND SCHENFELD, E. Type Matching, Type-Graphs, and the Schanuel Conjecture. *ACM Transactions on Programming Languages and Systems 14*, 4 (1992), 574–588.
. . . . xv, 80, 94

[116] *IEEE Knowledge & Data Engineering Exchange Workshop* (Newport Beach, California, November 1997), IEEE CS Press.
. . . . 212, 215

[117] KIRANI, S., AND TSAI, W.-T. Method sequence specification and verification of classes. In *Testing Object-Oriented Software*, D. C. Kung, P. Hsia, and J. Gao, Eds. IEEE Computer Society Press, 1998, pp. 43–53. 153

[118] KOHAVI, R., AND PROVOST, F. Glossary of terms – special issue on applications of machine learning and the knowledge discovery process. *Machine Learning, Kluwer Academic Publishers, Boston 30*, 2/3 (February/March 1998), 271–274. 127

[119] KOHONEN, T. *Self-organizing maps*. Springer series in information sciences. Springer, Berlin, 1997. 2. ed. 50

[120] KOTSCHNIG, H. Semantikgesteuertes Software-Retrieval. Master's thesis, Universität Klagenfurt, 1993. 55

[121] KOZACZYNSKI, W., AND BOOCH, G. Component-Based Software Engineering. *IEEE Software 15*, 15 (September 1998), 34–36. 21

[122] KRUEGER, C. W. Software Reuse. *ACM Computing Surveys 24*, 2 (June 1992), 131–183. .... 9, 36, 39, 56

[123] LANGLEY, P. *Elements of Machine Learning*. Morgan Kaufmann Publisher, Inc., San Francisco, California, 1996. .... 128, 129, 132, 133

[124] LATOUR, L., Ed. *8th Workshop on Institutionalizing Software Reuse (WISR8)* (Columbus, OH, USA, March 23-26 1997). 198, 201, 210, 211, 212, 223

[125] LATOUR, L. The Need For a Cognitive Viewpoint on Software Component Understanding. In *[124]* [124]. 11

[126] LATOUR, L., AND DUSINK, L. Functional Fixedness in the Design of Software Artifacts. In Latour and Wentzel [127].
. . . . 11

[127] LATOUR, L., AND WENTZEL, K., Eds. *7th Workshop on the Institutionalizing of Software Reuse* – WISR7 (St. Charles, Illinois, US, August 28-30 1995).
. . . . 201, 205, 209, 212, 220

[128] LEAVENS, G. T., AND RUBY, C. Specification Facets for More Precise, Focused Documentation. In Latour [124].
. . . . 57

[129] LEE, J., LAI, L. F., FEI, Y., YANG, S. J., AND HUANG, W. T. Contextual Retrival Mechanism for Reusing Task-Based Specifications. In KDE97 [116].
. . . . 58

[130] LEE, N.-Y., AND LITECKY, C. R. An Empirical Study of Software Reuse with Special Attention to Ada. *IEEE Transactions on Software Engineering 23*, 9 (September 1997), 537–549.
. . . . 11

[131] LIM, W. C. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software 11*, 5 (September 1994), 23–30.
. . . . 9

[132] LUBARS, M. D. Domain Analysis and Domain Engineering in IDeA. In Prieto-Diàz [186], pp. 163–178.
. . . . 13

[133] LUNG, C.-H., AND URBAN, J. E. An approach to the classification of domain models in support of analogical reuse. In Samadzadeh and Zand [203], pp. 169–178.
. . . . 13, 42

[134] LUQI, AND GUO, J. Toward Automated Retrieval for a Software Component Repository. In *Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based System (IEEE ECBS), Nashville, USA* (1999), Institute of Electrical and Electronics Engineers, pp. 99–105.
. . . . 59

[135] MAMBELLA, E., FERRARI, R., DE CARLI, F., AND LO SURDO, A. An Integrated Approach to Software Reuse. In Samadzadeh and Zand [203], pp. 63–71.
. . . . 12

[136] MAURER, F., AND KAISER, G. Software Engineering in the Internet Age. *IEEE Internet Computing 2*, 5 (1998), 22–24.
. . . . 17, 36

[137] MAYMIR-DUCHARME, F. A. Architectures Overview: Concepts, Processes, Methods, Tools and DoD Policy & Guidance, May 1999. Professional Development Seminar at SSR99 Los Angeles, CA.
. . . . 12

[138] MCILROY, D. Mass Produced Software Components. In Nauer and Randell [160], pp. 138–155.
. . . . 7

[139] MERKL, D., AND RAUBER, A. Document Classification with Unsupervised Artificial Neural Networks. In *Soft Computing in Information Retrieval*, F. Crestani and G. Pasi, Eds. Physica Verlag & Co, 2000, pp. 102–121.
. . . . 50, 51

[140] MERKL, D., AND TJOA, A. A Self-Organizing Map that Learns the Semantic Similarity of Reusable Software Components. In *Proc. of the 5th Australian Conference on Neural Networks (ACNN'94)* (Brisbane, Australia, February 1994), pp. 13–16.
. . . . 50

[141] MERKL, D., TJOA, A., AND KAPPEL, G. Learning the Semantic Similarity of Reusable Software Components. In Frakes [73].
. . . . 50

[142] MEYER, B. *Eiffel : the Language*. Object-Oriented series. Prentice-Hall, Englewood Cliffs, NJ, USA, 1992.
. . . . 89

[143] MEYER, B. On To Components. *IEEE Computer 32*, 1 (January 1999), 139–140.
. . . . xvii, 22, 24, 225

[144] MICHALSKI, R. S. *Machine Learning: An Artificial Intelligence Approach*. Vol. 1 of Carbonell and Mitchel [42], 1983, ch. A Theory and Methodology of Inductive Learning.
. . . . 128

[145] MICLET, L., AND QUINQUETON, J. *Computer and Systems Sciences*, vol. 45 of *Nato Series F*. Springer-Verlag, 1988, ch. Syntactic and Structural Pattern Recognition, pp. 153–171.
. . . . 163

[146] MILI, A., MILI, F., AND DESHARNAIS, J. *Program Construction: A Heuristic Relational Approach*. Oxford University Press, New York, June 1994.
. . . . 55, 117

[147] MILI, A., MILI, R., AND MITTERMEIR, R. T. A survey of software reuse libraries. *Annals of Software Engineering – Systematic Software Reuse 5* (1998), 349–414.
. . . . 37, 39, 40

[148] MILI, H., AH-KI, E., GODIN, R., AND MCHEICK, H. Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval. *ACM SIGSOFT Software Engineering Notes 22*, 3 (May 1997), 89–98. also in: SSR '97. Proceedings of the 1997 Symposium on Symposium on Software Reusability, pages 89-98.
. . . . 47

[149] MILI, R., MILI, A., AND MITTERMEIR, R. T. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions On Software Engineering 23*, 7 (July 1997), 445 – 460.
. . . . 53, 54, 55, 117

[150] MITCHELL, T. M. Machine Learning and Data Mining. *Communications of the ACM 42*, 11 (November 1999), 31–36.
. . . . 116

[151] MITTERMEIR, R. T. Specification Aspects for Software Reuse. In *Proceedings 1^{ièr} Sèminaire International sur le Gènie Logiciel à Oran* (Algeria, 1988).
. . . . 55

[152] MITTERMEIR, R. T. *Software Re-use*. Workshops in Computing. Springer Verlag, Utrecht, 1989, ch. Design-Aspects supporting Software Re-use, pp. 115–119.
. . . . 55

[153] MITTERMEIR, R. T. Software Reuse: New Chances and New Challenges. In *Proc. 2ieme Forum International d' Informatique Applique* (Tunis, Tunisia, March 1996), A. Ferchichi and J.-P. Giraudin, Eds., pp. 86–105.
. . . . 9

[154] MITTERMEIR, R. T., AND POZEWAUNIG, H. Classifying Components by Behavioral Abstraction. In Wang [227], pp. 547–550.
. . . . 123, 124

[155] MITTERMEIR, R. T., POZEWAUNIG, H., MILI, A., AND MILI, R. Uncertainty Aspects in Component Retrieval. In *Proceedings of the 7th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems – IPMU98* (July 1998), pp. 564–571.
. . . . xv, 33, 34, 37

[156] MIURA, T., AND SHIOYA, I. On Complex Type Hierarchy. In KDE97 [116], pp. 156 – 164.
. . . . 58

[157] MÜHLHÄUSER, M. *Multimedia*, 2nd ed. Vol. 1 of Rechenberg and Pomberger [194], 1999, ch. 2, pp. 855–874.
. . . . 52

[158] MYERS, G. J. *The Art of Software Testing*. Wiley & Sons, New York, 1979.
. . . . 120, 121

[159] NADA, N., TUWAIM, S., AL HOMOD, S., AND TOPALOGLU, N. Y. Issues in Reuse Based Software Development. In Wang [227], pp. 519–526.
. . . . 16

[160] NAUER, P., AND RANDELL, B., Eds. *Software Engineering; Report on a Conference by the NATO Science Committee (Garmisch, Germany)* (Brussels, Belgium, October 1968), NATO Scientific Affairs Division.
. . . . 7, 213

[161] Neighbors, J. M. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transaction on Software Engineering 10*, 5 (September 1984), 564–574.
. . . . 13

[162] Neighbors, J. M. An Assessment fo Reuse Technology after Ten Years. In Frakes [73], pp. 6–13.
. . . . 9, 32, 225, 226, 228, 229

[163] Nevill-Manning, C. G., and Witten, I. H. Detecting Sequential Structure. In *Proceedings of the Workshop on Programming by Demonstration, ML'95* (July 1995).
. . . . 170

[164] Nielsen, J. *Multimedia and hypertext : the Internet and beyond*. Morgan Kaufmann Publishers, San Diego, Calif., 1999.
. . . . 52

[165] Noack, J., and Schienmann, B. Obektorientierte Vorgehensmodell im Vergleich. *Informatik Spektrum 22*, 3 (June 1999), 166–180.
. . . . 29

[166] Novak Jr., G. S. Creation of Views for Reuse of Software with Different Data Representations. *IEEE Transactions on Software Engineering 22*, 12 (December 1995), 993–1003.
. . . . 59, 62

[167] Novak Jr., G. S. Software Reuse by Specialization of Generic Procedures through Views. *IEEE Transactions On Software Engineering 23*, 7 (July 1997), 401 – 417.
. . . . 59, 62

[168] Oates, T., Fioiu, L., and Cohen, P. R. Using Dynamic Time Warping to Bootstrap HMM-Based Clustering of Time Series. In Sun and Giles [219], pp. 35–52.
. . . . 163

[169] Ostertag, E., Hendler, J., Prieto-Diàz, R., and Braun, C. Computing similarity in a reuse library system: an AI-based approach. *ACM Transactions on Software Engineering and Methodology 1*, 3 (July 1992),

205–228.
. . . . 48, 225

[170] PAGEL, B.-U., AND SIX, H.-W. *Software Engineering – Band 1: Die Phasen der Softwareentwicklung*. Addison-Wesley, 1994.
. . . . 7

[171] PANDE, H., AND RYDER, B. Static type determination for C++. In *Proceedings of the 1994 USENIX C++ Conference: April 11–14, 1994, Cambridge, MA* (Berkeley, CA, USA, April 1994), U. Association, Ed., USENIX, pp. 85–97.
. . . . 93

[172] PAREKH, R., AND HONAVAR, V. Learning DFA from Simple Examples. In *LNAI 1997* (1997), vol. 1316, Springer, pp. 116–131.
. . . . 162

[173] PAREKH, R., NICHITIU, C., AND HONAVAR, V. A Polynomial Time Incremental Algorithm for Regular Grammar Inference. Tech. Rep. 97-03, Department of Computer Science, Iowa State University, 1997.
. . . . 162

[174] PODGURSKI, A., AND PIERCE, L. Behaviour Sampling: A Technique for Automated Retrieval of Reusable Components. In *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia, May 1992), 14, ACM, ACM Press, pp. 349–361.
. . . . 60, 113

[175] PODGURSKI, A., AND PIERCE, L. Retrieving Reusable Software by Sampling Behavior. *ACM Transactions on Software Engineering and Methodology 2*, 3 (July 1993), 286 – 303.
. . . . 60, 61, 113, 146

[176] PORT, D., AND KAISER, G. Collaborative Technologies for Evolving Software Systems. *IEEE Internet Computing 2*, 6 (November 1998), 79–83.
. . . . 36

[177] POSTON, R. M., Ed. *Automating Specification-Based Software Testing*, vol. 1. IEEE Computer Society, 1996.
. . . . 218

[178] POSTON, R. M. Specification-Based Testing: What is it? How can it be Automated. In Poston 1996 [177], ch. 1, pp. 9 – 21.
. . . . 117

[179] POULIN, J. S. Reuse: Been There, Done That. *Communications of the ACM 42*, 5 (May 1999), 98–100.
. . . . 16, 17

[180] POULIN, J. S., AND WERKMAN, K. J. Melding Structured Abstracts and the World Wide Web for Retrieval of Reusable Components. In Samadzadeh and Zand [203], pp. 160 – 168.
. . . . 32

[181] POZEWAUNIG, H., AND MITTERMEIR, R. T. Self Classifying Components - Generating Decision Trees from Test Cases. In *Proceedings of the 12th International Conference on software engineering & Knowledge Engineering – SEKE2000* (Chicago, Ill, USA, July 2000), S.-K. Chang, Ed., pp. 352–360.
. . . . 120, 126, 130, 144, 145, 148

[182] POZEWAUNIG, H., AND RAUNER-REITHMAYER, D. Support of Semantics Recovery during Code Scavenging using Repository Classification. In *Symposium on Software Reusability – SSR'99* (Los Angeles, CA, May 1999), ACM, pp. 65–72.
. . . . 13, 196

[183] PRESSMAN, R. S. Software Engineering. In *Software Engineering*, M. Dorfman and R. H. Thayer, Eds. IEEE Computer Society Press, 1997, pp. 57–74.
. . . . 18, 155

[184] PRIETO-DIÀZ, R. Classification of Reusable Modules. In *Software Reusability – Concepts and Models*, T. J. Biggerstaff and A. J. Perlis, Eds., vol. I of *Frontier Series*. acm Press, Reading, Massachusetts, 1989, ch. 4, pp. 99–123.
. . . . 44

[185] PRIETO-DIÀZ, R. Implementing Faceted Classification for Software Reuse (Experience Report). In *Proc. 12th International Conference on*

Software Engineering, Nice, France (March 1990), IEEE Computer Society Press, pp. 300–304.
. . . . 44

[186] PRIETO-DIÀZ, R., Ed. *Domain Analysis and Software Systems Modeling*, vol. 1. IEEE Computer Society Press, Washington, DC, USA, 1991.
. . . . 197, 212

[187] PRIETO-DIÀZ, R. Implementing Faceted Classification for Software Reuse. *Communications of the ACM 43*, 5 (May 1991), 88 – 97.
. . . . xvii, 44, 45

[188] PRINZ, P. *ANSI-C Guide – das kompakte Nachschlagewerk für alle C-Compiler*. IWT-Verlag, Vaterstetten bei München, 1993.
. . . . 143

[189] QUINLAN, J. R. *C5.0: An Informal Tutorial*. RuleQuest Research Pty Ltd, 30 Athena Avenue, St Ives NSW 2075, Australia.
. . . . 136

[190] QUINLAN, J. R. *Machine Learning: An Artificial Intelligence Approach*. Vol. 1 of Carbonell and Mitchel [42], 1983, ch. Learning efficient classification procedures and their application to chess end games, pp. 463–482.
. . . . 133

[191] QUINLAN, J. R. Learning logical definitions from relations. *Machine Learning 5*, 3 (1990), 239–266.
. . . . 133

[192] QUINLAN, J. R. *C4.5 – Programs for Machine Learning*. The Morgan Kaufmann series in machine learning. Morgan Kaufman Publishers, San Mateo, CA, USA, 1993.
. . . . 129, 135

[193] RAKIC, M., AND MEDVIDOVIC, N. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. In *Proceedings of the 2001 Symposium on Software Reusability, SSR'01* (Toronto, Ontario Canada, May, 18–20 2001), acm press, pp. 11–18.
. . . . 154

[194] RECHENBERG, P., AND POMBERGER, G., Eds. *Informatik-Handbuch*, 2nd ed., vol. 1. Carl Hanser Verlag, 1999.
. . . . 200, 204, 205, 215, 220

[195] REIFER, D. J. *Practical Software Reuse – Strategies for Introducing Reuse Concepts in Your Organization*. Wiley Computer Publishing, New York, et.al., 1997.
. . . . xv, 9, 12, 15, 29, 225

[196] REISS, S. P., AND RENIERIS, M. Encoding Program Executions. In *Proceedings of the 23rd International Conference on Software Engineering – ICSE2001* (Toronto, Canada, May 2001), pp. 221–230.
. . . . 187

[197] REYES, A. A., AND RICHARDSON, D. J. Siddhartha: A technique for building domain-specific test synthesizers. In *Proceedings 14th IEEE International Conference on Automated Software Engineering* (Cocoa Beach, FL, USA, 1999), IEEE Computer Society Press.
. . . . 117

[198] RIBEIRO, A. N., AND MARTINS, F. M. A Fuzzy Query Language for a Software Reuse Environment. In Latour and Wentzel [127].
. . . . 32

[199] ROHLING, H., AND MAY, T. *Informatik-Handbuch*, 2nd ed. Vol. 1 of Rechenberg and Pomberger [194], 1999, ch. Informations- und Codierungstheorie, pp. 191–216.
. . . . 133

[200] RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey, 1991.
. . . . 155, 157

[201] SALTON, G., AND BUCKLEY, C. Term-Weighting Approaches in Automatic Text Retrieval. In Sparck Jones and Willett [215], pp. 323–328.
. . . . 44

[202] SALTON, G., WONG, A., AND YANG, C. S. A Vector Space Model for Automatic Indexing. In Sparck Jones and Willett [215], ch. Models,

pp. 273–280.
. . . . 43

[203] SAMADZADEH, M., AND ZAND, M., Eds. *Proceedings of the ACM SIGSOFT Symposium on Software Reusability* (Seattle, Washington USA, April 1995), Association for Computing Machinery.
. . . . 208, 210, 212, 213, 218, 223

[204] SARACEVIC, T., KANTOR, P., CHAMIS, A. Y., AND TRIVISON, D. A Study of Information Seeking and Retrieving. Background and Methodology. In Sparck Jones and Willett [215], ch. Evaluation, pp. 175–190.
. . . . 36

[205] SCHÄFER, W., PRIETO-DIÁZ, R., AND MATSUMOTO, M., Eds. *Software Reusability*. Workshop Series. Ellis Horwood, New York, London, et. al., 1994.
. . . . 12

[206] SCHUMANN, J., AND FISCHER, B. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of the 12th International Automated Software Engineering Conference* (Nevada, USA, November 1997), IEEE CS, pp. 246–254.
. . . . 56, 57

[207] SEACORD, R. C., HISSAM, S. A., AND WALLNAU, K. C. Agora: A Search Engine for Software Components. *IEEE Internet Computing 2*, 6 (November 1998), 62–72.
. . . . 36, 37

[208] SEDGEWICK, R. *Algorithms in MODULA-3*. Addison-Wesley, Reading, Massachusetts, USA, 1993.
. . . . 35

[209] SIEGEL, J. Component and Object Technology: A Preview of CORBA 3. *IEEE Compter 32*, 5 (May 1999), 114–116.
. . . . 21

[210] SINDRE, G., CONRADI, R., AND KARLSSON, E.-A. The REBOOT Approach to Software Reuse. *Journal of Systems and Software 30*, 3 (September 1995), 201–212.
. . . . 45

[211] SNELTING, G., FISCHER, B., GROSCH, F.-J., KIEVERNAGEL, M., AND ZELLER, A. Die inferenzbasierte Softwareentwicklungsumgebung NORA. *Informatik - Forschung und Entwicklung 9*, 3 (1994), 116–131. in german. 55, 56, 57

[212] SNYDER, W. K. *The SETL2 Programming Language*. Courant Institute of Mathematical Sciences, `http://www.cs.nyu.edu/cs/faculty/paige/courses/setl/report.ps`, current September, 2001, New York University, NY10012, September 1990. 84

[213] SPARCK JONES, K. Search Term Relevance Weighting given Little Relevance Information. In Sparck Jones and Willett [215], pp. 329–338. 44

[214] SPARCK JONES, K., AND WILLETT, P. Overall Introduction (to Information Retrieval). In *[215]* [215], pp. 1–8. 39

[215] SPARCK JONES, K., AND WILLETT, P., Eds. *Readings in Information Retrieval*. Multimedia Information and Systems. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1997. . . . . 220, 221, 222

[216] SPINELLIS, D. Explore, Excogitate, Exploit: Component Mining. *IEEE Computer 32*, 9 (September 1999), 114–116. 13

[217] SPIVEY, J. M. An introduction to Z and formal specifications. *IEEE Software Engineering Journal 4*, 1 (January 1989), 40–50. 53

[218] STOPPER, A. Spezifikation und Normalisierung objekt-orientierter Systeme. Master's thesis, Universität Klagenfurt, 1991. 55

[219] SUN, R., AND GILES, C. L., Eds. *Sequence Learning – Paradigms, Algorithms, and Applications*, vol. 1828 of *Lecture Notes in Computer Science*. Springer Verlag, 2001. . . . . 161, 216

[220] SZYPERSKI, C. *Component Software : Beyond Object-Oriented Programming*. ACM Press, Harlow, England, 1998.
.... 20, 21, 153

[221] THOMASON, S. What is a Component. *IEEE Computer 33*, 1 (November 2000), 55.
.... 24

[222] THOMPSON, S. *Type Theory and Functional Programming*. Addison-Wesley, 1991.
.... 58, 100

[223] THOMPSON, S. *The Craft of Functional Programming*. Addison-Wesley, 1996.
.... 100

[224] THULASIRAMAN, K., AND SWAMY, M. N. S. *Graphs – Theory and Algorithms*. Wiley Interscience, New York, 1992. I 237701 –¿ 30-5.
.... 167

[225] TRUMP, D. Using the WWW and the Internet to Support Corporate Reuse. In Latour [124].
.... 36, 37

[226] Unified Modeling Language (UML) 1.3 specification. Object Management Group, Needham, MA, USA, `http://www.omg.org/technology/documents/index.htm`, current September, 2001, March 2000.
.... 156, 157

[227] WANG, P. P., Ed. *JCIS'98 – 4th Joint Conference on Information Sciences* (RTP, North Carolina, USA, October, 23–28 1998), Association for Intelligent Machinery.
.... 199, 215

[228] WASMUND, M. The Spin-Off Illusion: Reuse is Not a By-Product. In Samadzadeh and Zand [203], pp. 219–221.
.... 11

[229] YE, Y., AND FISCHER, G. Promoting Reuse with Active Reuse Repository Systems. In *Proceedings of the 6th International Conference on Software Reuse - ICSR 6* (Vienna, Austria, June 2000), W. B. Frakes, Ed., Lecture

Notes in Computer Science, Springer Verlag, pp. 302–317.
. . . . 31, 32

[230] YE, Y., FISCHER, G., AND REEVES, B. Integrating Active Information Delivery and Reuse Repository Systems. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)* (NY, November 8–10 2000), D. S. Rosenblum, Ed., vol. 25, 6 of *ACM Software Engineering Notes*, ACM Press, pp. 60–68.
. . . . 32

[231] YE, Y., AND REEVES, B. An Active and Intelligent Agent for Component Location. In *Proceedings of Software Symposium 2000* (Kanazawa, Japan, June 21–23 2000), Software Engineer Association, pp. 67–74.
. . . . 32

[232] ZAREMSKI, A. M., AND WING, J. M. Signature matching: a Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology 4*, 2 (April 1995), 146 – 170.
. . . . 57, 58, 67, 68, 72

[233] ZAREMSKI, A. M., AND WING, J. M. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology 6*, 4 (October 1997), 333 – 369.
. . . . 57, 59

A

Glossary

The main part of this glossary is from the US National Institute for Standards and Technology (NIST) Glossary of Software Reuse Terms [114]. If other sources are used, they are stated explicitly.

ABS: *A*bstracted *B*ehavior *S*ampling. Techniques used to abstract and visualize information about the hidden order of events existing in test cases. ABS is the generic term for the algorithms *k*TAIL and MSEQ. → Ch. 7

AIRS: *AI*-base *R*euse *S*ystem developed by Ostertag et al. [169] on the basis of a knowledge base and a case based reasoning system. → Sec. 4.2.1

Asset: Any product of the software development life cycle that can be potentially reused. This includes: architectures, requirements, designs, code, lessons learned. → [195]

Black-box reuse: The assets can be used without any modifications. → [162]

Characterising tuple: A input-output pair of data such that the input induces discriminative output with respect to all components in a partition of a SBS-Repository (see section 6.3 on page 119).

COTS: *Commercial Off-The-Shelf* – Reusable software for the purpose of incorporating it into products that is available commercially.

Component: An asset which is a primary product of the implementation phase of the software lifecycle, a programming element [143]. A component might be represented as binary or as source code. → Sec. 2.5

Sec. 6.2 ← **Component set $\mathcal{C}$:** The set of all reusable components in a SBS-repository.

Sec. 6.2.3 ← **Component output set:** The set O_{c_j} containing all outputs of a specific component c_j. Related term: *descriptor*.

Sec. 6.1 ← **Data point:** A *data tuple* used as a basis for further behavior analysis and stored in a SBS repository. Data points can be obtained either from test cases, characterising tuples or striking samples.

Sec. 6.1 ← **Data tuple:** A pair $(\vec{i}, \vec{o})$ of two vectors. The input vector $\vec{i}$ holds all elements which are input to a function, whereas the output vector $\vec{o}$ holds all elements which are the result of the function call.
Synonym: *Test tuple*.

Sec. 6.2.3 ← **Descriptor:** The whole set of output vectors of a specific component in a SBS-partition.

Sec. 2.3 ← **Domain:** A domain is a distinct application area that can be supported by a class of systems with similar requirements and capabilities. A domain may exist before there are software systems to support it. Examples for domains are avionic control centers or human resource management systems.

Domain Model: A product of *Domain Analysis* which provides a representation of the requirements of the domain. The Domain Model identifies and describes the structure of data, flow of information, functions, constraints, and controls within the *domain* that are included in the domain. The domain model describes commonalities and variabilities among requirements for software systems in the domain.

Domain Preparation: see *Domain Definition*.

Domain Definition: The process of determining the scope and boundaries of a domain in order to define the major functions and capabilities within the domain, what functions and capabilities are excluded from the domain, and what interactions exist with external domains.

[103] ← **FSA:** *F*inite *S*tate *A*utomaton: A finite state automaton (or finite state machine, FSM, or FA) is a mathematical model to represent regular grammars.

[162] ← **Granularity of reuse:** Granularity refers to the seize of the assets. *Large-grain reuse* indicates, that the assets are very large (thousands up to millions of

lines of code). *Small-grain reuse* indicates that the assets are very small (up to several hundreds of lines of code).

Horizontal domain: A domain that provides information or services to more than one domain. Examples of Horizontal Domains include communications, graphical user interfaces, and databases.

Institutionalized Reuse: see *Systematic Reuse*

Meta output vector: A meta output vector is used to → Sec. 6.3.1

- render behavior explicitly which cannot be visualized otherwise, such as error reactions or temporal particularities (indefinite loops).
- allow for specifying (yet) unknown values, if the execution of a component on a given input is not possible or to costly.

MSEQ: The *m*ultiple SEQUITUR algorithm. It enhances the standard algorithm to handle domain knowledge, to abstract from iterations and to analyze multiple grammars. → Sec. 7.4.3

Ontology: The hierarchical structuring of knowledge about things by subcategorizing them according to their essential (or at least relevant and/or cognitive) qualities.

Opportunistic Reuse: The ad hoc Reuse of Assets in the development of Software Systems using a software development process that has not been altered to accommodate Systematic Reuse. In Opportunistic Reuse, the developer determines where Reuse can be applied to develop a Software System without the organized use of Domain Engineering products during successive stages of a Software Engineering process.
Synonym: *Ad-hoc reuse*.

Path symbol: The symbol ">" used to indicate the concatenation of labels of edges of signature graphs which takes place during graph flattening. → Sec. 5.4.2

Plug-in compatibility: The most common form of conditions which support black box reuse-retrieved components be reused "as is". If a component C is specified in the form of $pre_s\ C\ post_s$ and a query for it as $pre_q\ C\ post_q$, then plug-in compatibility is given if $(pre_q \Rightarrow pre_c) \wedge (pre_q \wedge post_c \Rightarrow post_q)$. → [69]

Product line: A (software) product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission.

[113] ← **REBOOT:** *RE*use *B*ased on *O*bject *O*riented *T*echniques. A project sponsored by the European community from 1991 to 1994 with the aim to study, develop, evaluate and disseminate advanced methodologies for reuse-driven and object-oriented software development with the emphasis on planned reuse.

[162] ← **Repository:** The storage where reusable assets together with historic and ancillary organization information are kept.

Chap. 2 ← **Reuse:** Software reuse is the practice of using existing software assets to develop new applications. Reusable software assets can be executable programs, code segments, documentation, requirements, design and architectures, test data and test plans, or software tools.

Reuse is the process of implementing or updatingsoftware systems using existing software assets.

Chap. 6 ← **SBS:** *S*tatic *B*ehavior *S*ampling. The technique of exploiting historical data of component executions to build an indexing and classification structure for browsing a partition of a repository.

Sec. 6.2.1 ← **SBS-Component:** The partial functional specification built by data points of a SBS repository. SBS-Components are not directly reusable, but their implementations are.

Searcher: A person trying to locate a component in a repository.

Sec. 6.2.2 ← **Signature congruent components C:** The set of components stored in a partition P_σ. All elements of $\mathbf{C}$ comply to the generalized signature σ.

Sec. 6.2 ← **SBS-Repository:** A repository based on SBS.

Chap. 5 ← **Signature:** An implementation independent, abstract representation of structural type information. Signatures comprise descriptions for data types, functions, and interface specifications of modules.

STARS: The Software Technology for Adaptable, Reliable Systems program is a joint program by different USA government agencies in cooperation with Boeing, Loral Federal Systems, and Loral Defense Systems-East. The goals is to increase software productivity, reliability, and quality by integrating support for modern software development processes and reuse concepts within software engineering environment (SEE) technology.

Systematic Reuse: Software development which is guided by an organized use of domain engineering products (including domain model, domain architecture, and other assets) during successive stages of a software engineering process to facilitate planned reuse of assets.
Synonym: *Institutionalized reuse*.

Striking sample: A input-output pair which is immediately attributable to a certain component by a domain knowledgeable searcher. $\rightarrow$ Sec. 6.1.3

Test tuple: See *Data tuple*.

Value set: The set $V_i = \{\vec{o}_{i_1}, \vec{o}_{i_2}, \ldots, \vec{o}_{i_n}\}$ containing all values generated by the execution of the components $c_1, c_2, \ldots, c_n$ when provided with the input vector $\vec{i}_i$. $\rightarrow$ Sec. 6.2.3

Vertical domain: A domain which addresses aspects of a single function or application area. Examples include payroll systems, automated weapons systems, robotic control systems. A Vertical Domain draws on capabilities from all horizontal domains that support its purpose.

White-box reuse: The assets must be modified before they can be reused. $\rightarrow$ [162]

B

General Signatur Matching Implementation

The SETL2 packages given in this section build the system for computing a generalized signature structure and provides with matching relations to operate on the generalized signature. The import relations holding between the various packages are depicted in figure B.1. In this section the `main` program and `Util` package are not shown, since they do not contain any specific algorithms related to the signature matching problem.

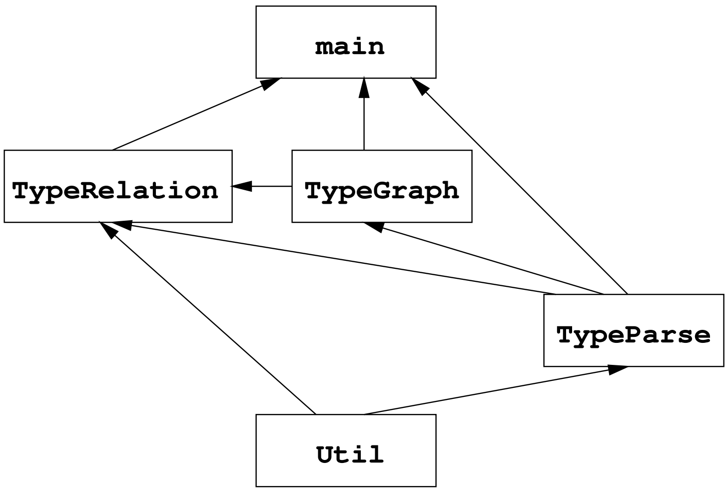


Figure B.1: The import relations of the signature matching system

B.1 The `TypeParse` Package

The package provides the main functionality for scanning and parsing a file containing type definitions. It exports three items: (1) the type operatores which can be expanded (called by the package `TypeRelation`) (2) `parseSig` provides the scanner and parser for analyzing the definitions given in the file `sigFile` (called by the main program) and (3) `readSubTypes` which generates an structure holding the a-priori knowledge about base subtype relations (called by the main program)

```
PACKAGE TypeParse;
-- Package for generating AST-like signature set from
-- signature definition file.

-- Signature Grammar
-- Declaration     ::= Module | Types.
-- Module          ::= 'MOD' SigName '=' Types.
-- Types           ::= TypeDec {';' TypeDec} '.'.
-- TypDec          ::= TypeName ['=' TypeBody].
 --TypeBody        ::= FunOp | CrossOp | UnionOp.
 --FunOp           ::= '->' '(' TypeOperandList ')'
 --                    '(' TypeOperandList ')'.
 --CrossOp         ::= 'X' '(' TypeOperandList ')'.
 --UnionOp         ::= 'U' '(' TypeOperandList ')'.
 --TypeOperandList ::= TypeOperand {',' TypeOperand}.
 --TypeOperand     ::= Label '/' (TypeName | TypeBody).
 --TypeName        ::= [ModuleName'.']Label.
 --ModuleName      ::= Label.
 --SigName         ::= Label.
 --Label           ::= char{char|digit}.
 --digit           ::= '0'|'1'|...|'9'.
 --char            ::= 'a'|'b'|...|'z'|'A'|'B'|..|'Z'.

-- End of Signature Grammar

 VAR
  ExpandableTypeOps := {"X", "U"}; -- n-ary typeoperators

  PROCEDURE parseSig(rd sigFile); -- returns set of signatures
  -- parse a file for type grammars (s.a. for the grammar)
  -- a typedec must be specified in one line
  -- comments are filtered out (lines beginning with %)
  -- returns the set of all types of the form
  -- {[type1],[type2],...}
  -- a type may be "typename" or
  --    typename, [op, [operand1, operand2,...]]
  -- operands are of the form [label, typedec]

  PROCEDURE readSubTypes(stdfile);
  -- reads a priori subtype definitions from the stdfile and
  -- generates an initial subtype graph
  -- the format of the definition in the stdfile is (one per line):
  -- T <: P
END TypeParse;
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

PACKAGE BODY TypeParse;
USE Util; -- import string tools

VAR
  -- Standard TypeOperators and Base types
  operatorSet := {"X", "->", "U"},
    -- set of recognizeable typ operators
  basetypeSet := {};
  -- e.g. {"int","bool","char", "real"}; -- standard base types

  -- #######################PROC####################
  PROCEDURE parseSig(rd sigFile);
  VAR
    lc := 0, -- line counter
    pc, -- counter for position within a line
    consume := true,
    ast := {},
    str := "",
    DelimChars := "/(),",
    DelimSet := {"/","(",")",","},-- delimiter set
    types := {},  -- contains types als maps {[name, typebody],...}
    fHandle := open(sigFile, "text-in");

  -- BEGIN
    geta(fHandle, str); -- get a line
    UNTIL eof() LOOP
      lc +:= 1; -- increment line counter
      pc := #str; -- position counter
      WHILE str /= om -- empty file encountered
        AND str /= "" -- empty string
        AND str(1) IN " \t" LOOP -- strip leading white spaces

            str := str(2..);

      END LOOP;
      IF str /= om AND str /= "" AND str(1) /= "%" THEN
        -- skip commented lines
        types with:= TypeDec(str); -- Start the parsing process
      END IF;
      geta(fHandle, str); -- get next line (next type declaration)
    END LOOP; -- UNITL eof

    close(fHandle);

    RETURN types;

  -- #######################PROC####################
  -- ### parse typestring ### ---
  -- TypDec = string [TypeBody]
  -- t should be of the form "transloc  = X (a/point, r/point)"

  PROCEDURE TypeDec(t); -- TypDec = TypeName ['=' TypeBody]
    VAR
      token := "";
```

```
--BEGIN
  token := getNextToken(t, consume); -- TypeName detection
   IF Util.isString(token) THEN -- is token str less special chars
    IF getNextToken(t, consume) = "=" THEN
    -- type body reached (complex type)
      RETURN [token, TypeBody(t)];
    ELSE -- basic type is 0-ary type operator
      AddBaseType(token);
      RETURN [token,[token,[]]];
    END IF;
  ELSE
    error("'"+token+"' is no string!");
  END IF;
END TypeDec;


-- ########################PROC#####################
-- ### TypeBody = FunOp | CrossOp | UnionOp
-- ########################PROC#####################
PROCEDURE TypeBody(rw str);
VAR
  typeOperandList,
  operator := "";
--BEGIN
  operator := getNextToken(str, consume);

  CASE operator
  WHEN "->" => typeOperandList := [operator, FunOp(str)];
  WHEN "X" =>  typeOperandList := [operator, CrossOp(str)];
  WHEN "U" =>  typeOperandList := [operator, UnionOp(str)];
  OTHERWISE =>
    typeOperandList := om;
    error(operator+ " is not a correct type operator!");
  END CASE;
  RETURN typeOperandList;

END TypeBody;

-- ########################PROC#####################
-- ### FunOp = '->' TypeOperandList  TypeOperandList
-- ########################PROC#####################
PROCEDURE FunOp(rw str);
  VAR inOpList, outOpList;

  IF getNextToken(str,consume) = "(" THEN
    IF getNextToken(str, NOT(consume)) /= ")" THEN
      inOpList := TypeOpList(str);
    ELSE
    -- empty input list is allowed for ->
      inOpList := [];
    END IF;
  ELSE
    error("No left bracket found for " +
        "left hand side TypeOperandList!");
  END IF;

  IF getNextToken(str,consume) /= ")" THEN
```

```
    error("No right bracket found for "+
        "left hand side TypeOperandList");
  END IF;

  IF getNextToken(str,consume) = "(" THEN
    IF getNextToken(str, NOT(consume)) /= ")" THEN
      outOpList := TypeOpList(str);
    ELSE
    -- empty output list is allowed for ->
      outOpList := [];
    END IF;
  ELSE
    error("No left bracket found for "+
        "right hand side TypeOperandList!");
  END IF;

  IF getNextToken(str,consume) /= ")" THEN
    error("No right bracket found for "+
        "right hand side TypeOperandList");
  END IF;

  RETURN [["˜I", ["X", inOpList]], ["˜O", ["X", outOpList]]];
END FunOp;

-- #######################PROC####################
-- ### CrossOp = ’X’ TypeOperandList
-- #######################PROC####################
PROCEDURE CrossOp(rw str);
  VAR opList;

  IF getNextToken(str,consume) = "(" THEN

    opList := TypeOpList(str);
  ELSE
    error("No left bracket found for "+
        "left hand side TypeOperandList!");
  END IF;

  IF getNextToken(str,consume) /= ")" THEN
    error("No right bracket found for "+
        "left hand side TypeOperandList");
  END IF;

  RETURN opList;
END CrossOp;

-- #######################PROC####################
-- ### UnionOp = ’U’ TypeOperandList
-- #######################PROC####################
PROCEDURE UnionOp(rw str);
  VAR opList;

  IF getNextToken(str,consume) = "(" THEN

    opList := TypeOpList(str);
  ELSE
    error("No left bracket found for "+
```

```
        "left hand side TypeOperandList!");
  END IF;

  IF getNextToken(str,consume) /= ")" THEN
    error("No right bracket found for "+
        "left hand side TypeOperandList");
  END IF;

  RETURN opList;
END UnionOp;


-- #########################PROC#####################

PROCEDURE TypeOpList(rw str);
VAR
  list := [];
-- BEGIN
  list with:= TypeOperand(str);

  WHILE   getNextToken(str,not(consume)) = "," LOOP
    getNextToken(str,consume);
    list with:= TypeOperand(str);
  END LOOP;
  RETURN list;
END TypeOpList;



-- #########################PROC#####################
--
PROCEDURE TypeOperand(rw str);
  --TypeOperand = label '/' (TypeName | TypeBody)
VAR
  typedec, label;
-- BEGIN
  label := getNextToken(str,consume);
  IF getNextToken(str,consume) = "/" THEN

    IF getNextToken(str,not(consume)) IN OpSet() THEN
      -- look a head
      typedec := TypeBody(str);
    ELSE -- TypeName
      typedec := getNextToken(str,consume);
    END IF;
  ELSE
    error("\tWrong label-typedec delimiter for '"+label+"'.");
  END IF;
  RETURN [label, typedec];
END TypeOperand;


-- #########################PROC#####################
--### parse typestring ### ---
PROCEDURE getNextToken(rw str, consume);
-- in str, out string without the token
  VAR
```

```
      token := "",
      hstr := str, -- copy the content of str to auxiliary string
      stopChars := "\t %"; -- tab space comment comma

  --BEGIN
    -- remove leading stop characters (tabs, spaces)
    WHILE any(hstr,stopChars) /= "" LOOP
      NULL;
    END LOOP;
    IF hstr(1) IN DelimSet THEN -- is here a delimiter
      -- substract first char from string
      token := len(hstr, 1);
    ELSE
      token := break(hstr, stopChars+DelimChars);
      -- breaks string on stop list
    END IF;

    IF consume THEN
    -- not in look ahead mode, no consuming of characters
      str:=hstr; -- now it is true
    END IF;

    RETURN token;
  END getNextToken;

  -- #######################PROC####################
  PROCEDURE error(msg);
    print(lc, ":", pc-#str, "\tERROR! ", msg);
    stop;
  END error;

  --################# P R O C E D U R E ##############################
  PROCEDURE AddBaseType(token);
    basetypeSet with:= token;
  END AddBaseType;

  --################# P R O C E D U R E ##############################
  PROCEDURE OpSet();
    RETURN operatorSet;
  END OpSet;

END parseSig;

  --################# P R O C E D U R E ##############################
  PROCEDURE readSubTypes(stdfile);

  -- reads a priori subtype definitions from the stdfile and
  -- generates an initial subtype graph
  -- the format of the definition in the stdfile is (one per line):
  -- T <: P

  VAR
    sbt, spt, -- sub- and supertype names
    stg := {},
      -- sub type graph, hold elements of the form [s, t, "<:"]
    line, -- a read line from the file
    fhandle; -- subtype definition file
```

```
    fHandle := open(stdfile, "text-in");

    -- BEGIN

      geta(fHandle, line); -- get a line
      UNTIL eof() LOOP
        WHILE line /= om -- empty file encountered
          AND line /= "" -- empty string
          AND line(1) IN " \t" LOOP -- strip leading white spaces

            line := line(2..);

        END LOOP;
        IF line /= om AND line /= "" AND line(1) /= "%" THEN
        -- skip commented lines

          sbt := break(line, " <:");
          IF "<:" IN line THEN
            span(line, " <:");
            spt := break(line, " \t\n");
            stg with:= [sbt, spt, "<:"];
           ELSE
            print("Wrong subtype definition syntax!"+
                " '<:' expected!");
            EXIT;
          END IF;
        END IF;
        geta(fHandle, line);
        -- get next line (next type declaration)

      END LOOP;
      close(fHandle);
    RETURN stg;
  END readSubTypes;

END TypeParse;
```

B.2 The `TypeGraph` Package

As its name indicates the `TypeGraph` package provides functions used to generate graphs from type relations. The package exports the constants `RootDesignator` and `BaseTypeDesignator` (used in package `TypeRelation`) and the functions `buildTypeGraph` (getting in type structures and producing a graph repesentation and a signature representation) (called from the main program). Furthermore, some auxiliary procedures to set the layout, transform graph structures to external formats (dot-files) and to get specific information from the graph `getNodeSet` are provides as well.

```
PACKAGE TypeGraph;
  VAR
    RootDesignator := "ROOT",  -- the root type anchor
```

```
    BaseTypeDesignator := "BASETYPE",

    -- the dividign sign for separating
    -- typenames and operators internally
    opSign := "%";

  PROCEDURE buildTypeGraph (rd originSigs, wr g, wr sigs, rd ab);
  -- computes graph of types
  -- in:  originalSigs: Set of all Signatures,
  --      ab: alternative base type representation flag
  -- out:   g ... Graph representation of originalSigs
  -- out:   sigs ... set of signatures with all structures
  --      expressed with a name

  PROCEDURE setLayout(rd verticefont, rd verticeshape,
        rd graphdir, rd vspace, rd linkfont, rd linkcolor);
  -- sets diverse layout parameter, if needed.
  -- If one layoutparamter should remain
  -- untouched, insert the emtpy string,

  PROCEDURE Visualize(rd g, rd label, rd outfile);
  -- for generating dot-file to express type hierarchy

  PROCEDURE getNodeSet(rd g);
  -- auxiliary task, generate a set of all type names

END TypeGraph;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

PACKAGE BODY TypeGraph;

USE TypeParse; -- importing compatible type operator set
VAR
  -- global visualization style parameters
  nodefont := "Times-Roman", -- PS-Font
  nodeshape := "ellipse",
    -- box, circle, doublecircle, diamond, polygon, epsf,...
  layoutdir := "LR",  -- alternative TB (top 2 bottom)
  nodesep :=  0.3,    -- space between nodes on the same rank
  orderdir := "out",  -- set to out, if initial ordering should
                -- be respected in graph layout
  edgefont := "Courier",
   edgecolor := "blue";

  --######################################--
  PROCEDURE setLayout(  verticefont, verticeshape,
                graphdir, vspace, linkfont, linkcolor);
  --BEGIN
  IF verticefont /= "" THEN nodefont := verticefont; END IF;
  IF verticeshape
    IN {"box", "circle", "doublecircle", "diamond", "polygon"} THEN
    nodeshape := verticeshape;
  END IF;
  IF graphdir IN {"TB", "LR"} THEN
    layoutdir := graphdir;
  END IF;
```

```
  IF vspace /= "" AND vspace /= 0 THEN
    -- space between nodes on same rank
    nodesep := vspace;
  END IF;
  IF linkfont /= "" THEN edgefont := linkfont; END IF;
  IF linkcolor
    IN {"red", "blue", "green", "yellow", "black", "brown"} THEN
    edgecolor := linkcolor;
  END IF;

  END setLayout;


--################# P R O C E D U R E #################################
  PROCEDURE Visualize(rd g, rd label, rd outfile);
  VAR
    nodeset := getNodeSet(g), -- the set of all edges in the graph
    ofh; -- output file handle

  -- for generating dot-file to express type hierarchy
  -- BEGIN

    IF outfile /= OM THEN  -- file name given?
        ofh := open(outfile, "text-out");

      -- print header information and general style paramters
      printa(ofh, "digraph \"type hierarchy\" {");

      -- styles for the whole graph
      printa(ofh, "graph [\n\tfontsize = \"14\" \n\tfontname "+
          "= \"Times-Roman\" \n\tfontcolor = \"black\""+
          " \n\tlabel = \"’" + outfile+"’ at ", date() +
          " - "+ time(), "\" \n\tcolor = \"black\" \n\trankdir"+
          " =\"", layoutdir, "\" \n\tnodesep=",nodesep,
          "\nranksep=0.0", "\nmclimit=\"10.0\"",
          "\n\tordering=\"", orderdir,"\"","]");

      -- styles for the nodes
      printa(ofh, "node [\n\tfontsize = \"14\" \n\tfontname = \"",
          nodefont, "\" \n\tfontcolor = \"black\" \n\tshape = \"",
          nodeshape, "\" \n\tcolor = \"black\" ]");

      -- styles for the edges
      printa(ofh, "edge [\n\tfontsize = \"18\" \n\tfontname = \"",
          edgefont, "\" \n\tfontcolor = \"", edgecolor,
          "\" \n\tcolor = \"black\" ] ");


      -- print the vertice set
      FOR v in nodeset LOOP
        CASE
        WHEN v(2) = BaseTypeDesignator  =>
          putVertice(v(1)+opSign+v(2), v(1), "doublecircle",
              "blue", ofh); --(node, label, shape, filehandler)
        WHEN v(2) = RootDesignator =>
          putVertice(v(1)+opSign+v(2), v(1), "box", "blue",ofh);
        OTHERWISE =>putVertice(v(1)+opSign+v(2), v(2), "circle",
```

```
          "black", ofh);
      END CASE;
    END LOOP;

    FOR e IN g LOOP
      CASE
      WHEN e(1) = "root"+opSign+RootDesignator =>
        putEdge(e(1), e(2), e(3), "blue", ofh);
      OTHERWISE =>
        putEdge(e(1), e(2), e(3), "black", ofh);
      END CASE;
    END LOOP;
    -- print footer
    printa(ofh, "}");

      close(ofh);
  ELSE -- print output to stdout
     print(g);
  END IF;



-- ##########################################################
-- ##  not visible auxiliar procedures
-- ##########################################################

  PROCEDURE putEdge(s, d, t, color, fh);
    --(source, destination, label, color, filehandle)

    printa(fh, "\"", s, "\" -> \"", d,  "\" [");
    printa(fh, "\tfontcolor = \"", color,"\"");
    printa(fh, "\tcolor = \"", color,"\"");
    printa(fh, "\tlabel = \"", t, "\"");printa(fh, "]");
  END putEdge;

PROCEDURE putVertice(v, label, shape, fontcolor, fh);
  --(node, label, shape, filehandler)
  -- put vertice of string form to file fh in dot format
  -- BEGIN
    printa(fh, "\"", v, "\" [");
    printa(fh, "\tlabel = \"", label,"\"");
    printa(fh, "\tshape = \"", shape,"\"");
    printa(fh, "\tfontcolor = \"", fontcolor, "\"");
    printa(fh, "]");
  END putVertice;
END Visualize;

--################# P R O C E D U R E ##############################
--# build an AST-like graph for internal representation
--################################################################

PROCEDURE buildTypeGraph (rd originSigs, wr g, wr sigs, rd ab);
    -- input is the set of signatures {[TypeOp, [Oplist]]}
    -- returns graph of type context
    -- a graph is a set with vertices of the form
    --    [source, target, label]
    -- and returns the set of explicit sigs, where all
```

```
    --    implicitly declared types are named uniquely

VAR
  pos := 0, -- position counter for struct operators
  hsigs := originSigs, -- sigs must be restored
               -- without explicit types
  opNum := 1, -- for generating unique operator enumeration
          -- for implicit types
  nodeset:= {},   -- e.g. {["Point","X"],...}, -- [name, node type]
  operandlist, -- list of operands to a certain type
  operatorType; -- holds the type operator -Y,X,U

  --BEGIN
   sigs := originSigs;
  g := {}; -- graph with edges [source, target, label]

  -- build a seperate set of all vertices
  FOR typename IN  domain sigs LOOP
      -- for all declared (named) types

    CASE
    WHEN typename = sigs(typename)(1) =>
      -- base type found; name = TypeOp

      IF ab THEN -- alternative base type representation:
        -- base types are linked to root,
        -- necessary if only base types are analyzed
        g with:= [  "root"+opSign+RootDesignator,
            typename+opSign+BaseTypeDesignator, typename];
      END IF;

      nodeset with:= [typename, BaseTypeDesignator];

    OTHERWISE =>
      operatorType := sigs(typename)(1);
      nodeset with:=  [typename, operatorType];

      -- insert vertice from root to types
      g with:= ["root"+opSign+RootDesignator,
            typename+opSign+operatorType, typename];

      -- next: analyze the oplist and substitute implicitly
      -- defined types by their explicitly defined counterparts
      analyzeOperandList([typename, sigs(typename)],
                  nodeset, hsigs);
    END CASE;
  END LOOP;
  -- now use the explicit types instead of implicit ones
  sigs := hsigs;
  -- vertice set is now completed
  -- build edgeds

  FOR node IN nodeset | node(2) /= BaseTypeDesignator LOOP
    operandlist := sigs(node(1))(2);

    pos := 1;
    FOR operand IN operandlist LOOP
```

```
    -- insert edge grom current type to component types
    IF nodeset(operand(2)) /= om THEN
    -- referenced type is defined
      IF node(2) = "X" THEN -- Struct operator found
      -- here also the position of the operator is imporant
        g with:= [node(1)+opSign+node(2),
              operand(2)+opSign+nodeset(operand(2)),
              operand(1)+" ["+str(pos)+"]"];
      ELSE
        g with:= [node(1)+opSign+node(2),
              operand(2)+opSign+nodeset(operand(2)),
              operand(1)];
      END IF; -- Struct (X) operator found
    ELSE  -- referenced type in operand list is not declared
      print("Undefined type from '",node(1)+opSign+node(2),
          "', labeld with '",operand(1),"'! EXITING");
      stop;
    END IF;
    pos +:= 1;
  END LOOP;
END LOOP;
nodeset with:= ["root",RootDesignator];
-- insert meta vertice "root"

--################ P R O C E D U R E ##########################
-- initially called from BuilTypeGraph
PROCEDURE analyzeOperandList(rd typedec, rw nodes, rw sigs);
VAR
  newTypeName := "", -- for storing new node names
  newTypeDec := typedec;

-- BEGIN
  FOR operand IN typedec(2)(2) LOOP
    -- for every element in the operand list of the root type
    -- generate explicit type and
    -- substitute implicit with explicit form
    -- implicit type declaration e.g.:
    --  ["s", ["X", [["a", "int"], ["b", "bool"],
    --    ["c", "bool"]]]]
    -- explicit type declaration e.g.:
    --    ["r", "R"]

    -- operand is   ["X", [["a", "int"], ["b", "bool"],
    --          ["c", "bool"]]]
    IF is_Tuple(operand(2)) THEN
      -- implicit type declaration found
      -- generate an unique type name for new explicit type
      newTypeName := operand(1)+str(opNum)+"-xT";opNum +:= 1;

      -- delete operand from typedec
      sigs less:= newTypeDec;
        -- delete previous type version of current type

      newtypedec := replace( [operand(1), newTypeName],
                      operand, newtypedec);

      sigs with:= newtypedec;-- e.g.[operand(1),newTypeName];
```

```
          sigs with:= [newTypeName, operand(2)];
          nodes with:= [newTypeName, operand(2)(1)];
          -- generate  explicit node [typename, Operatortype]
          --  makeExplicit(operand);
          analyzeOperandList( [newTypeName, operand(2)],
                        nodes, sigs);
        END IF; -- if implict type declaration found
      END LOOP; -- all operands handled
    END analyzeOperandList;
  END buildTypeGraph;

  --################# P R O C E D U R E #############################
  -- replace substitutes oldList by newList in list
  PROCEDURE replace(rd newList, rd oldList, rd list);
     IF is_tuple(list)  THEN
      IF oldList = list THEN
        list := newList;
      ELSE
        FOR i IN [1..#list] LOOP
          -- depth first
          list(i) := replace(newList, oldList, list(i));
        END LOOP;
      END IF;
    END IF;
    RETURN list;
  END replace;


  PROCEDURE getNodeSet(rd g);
  -- auxiliary task, generate a set of all type names
  VAR
    nodeset := {};
  -- BEGIN
    FOR e IN g LOOP -- e is [FROM, TO, LABEL]
      nodeset with:= [break(e(1),TypeGraph.opsign), rbreak(e(1),
                 TypeGraph.opsign)];
      nodeset with:= [break(e(2),TypeGraph.opsign), rbreak(e(2),
                 TypeGraph.opsign)];
    END LOOP;
    RETURN nodeset;
  END getNodeSet;

END TypeGraph;
```

B.3 The `TypeRelation` Package

The definitions of this package are used by the main program. The implementation contains functions for `flattening` signature graph structures, checking for subtype properties (`isSubType`), and calculating constitutive equality (`areConstitutiveEqual`).

```
PACKAGE TypeRelation;
```

```
  PROCEDURE flatten(rd tn, rd g); -- returns expanded signature
  PROCEDURE isSubType(rd s, rd p, rd g, rw ssb);
    -- input:   typename s, typename p, type graph g,
    --        subtypegraph with a priori defintions
    -- output:  TRUE, is s is a subtype of p, FALSE otherwise,
    --        side effect: subtypegraph sg

  PROCEDURE areConstitutiveEqual(rd s, rd p, rd g);
    -- purpose: Check if s and p are constitutive type equal
    --  (strucutre checking without labels)
    -- input:   types s, p, type graph g
    -- output:  TRUE, if s and p are constitutive type equal,
    --        FALSE otherwise
END TypeRelation;

--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

PACKAGE BODY TypeRelation;
  USE
    Util, -- import string tools
    TypeParse, -- import Definitions from Parsing
    TypeGraph;
  VAR cnt := 0; -- global counter variable for creating unique names

  --################# P R O C E D U R E ###########################
  PROCEDURE flatten(rd tn, rd g);
  -- merge nodes, if they have compatible typeoperators
  -- Input is explicit typename tn and
  -- type graph g (a set with elements [FROM, TO, LABEL])

  VAR
    visitedlist := {},-- holds all already analyzed vertices
    -- get the "pure" type of typename tn
    mg := getSubStruct(tn, g),
    -- build the internal type represenation of tn by
    -- concatenating the type name with the internal division
    -- symbol for types and the type operator
    theType := tn+TypeGraph.opsign+TypeGraph.getNodeSet(mg)(tn);

  --BEGIN
    -- all preparations done, now merge type within the
    -- "pure" structure of the type tn

    RETURN tmerge(theType, mg); -- call t(ype)merge

    --################# P R O C E D U R E ########################
    -- PROCEDURE tmerge t(ype)merge -- defined within merge
    PROCEDURE tmerge(curType, mg);
        VAR
          -- determine operand of the operator list
          -- ol is set with elements [tn, typenode]
          -- e.g. {["b [2]", "int%BASETYPE"], ["a [1]", "R%X"]}
          ol := getOperandList(getTypeName(curType), mg),
        sl, -- and operands of its successors
        isChanged := FALSE, -- remembers if type is merged
          e; -- edges in the graph
```

```
-- BEGIN

  visitedlist with:= curType;
-- needed for determining recursions

  FOR e IN ol LOOP
    -- check merge condition for all operands
    IF getOperator(e(2)) = getOperator(curType)
      -- the same type operators?
      AND
      getOperator(e(2)) IN TypeParse.ExpandableTypeOps
      -- are the operators mergeable at all
      AND
      getTypeName(e(2)) /= getTypeName(curtype)
      -- no direct recursion
    THEN
    -- merging must be performed, structure is changed
    isChanged := TRUE;

      -- get all successors operands of current operand
      sl := getOperandList(getTypeName(e(2)),mg);

      FOR se IN sl LOOP
        -- successor operands are now operands of tn
        -- now eliminate direct way
      -- and redirect the current edge e such that
        -- curType->e2->se ==> tn->se
        mg with:= [curType, se(2),
          Util.squeeze(e(1)+">"+se(1))];

      END LOOP;
    -- delete e from mg
    mg less:= [curType, e(2), e(1)];

    -- only if the intermediate vertice e(2) is not
    -- referenced anymore, we can delete it
    IF NOT isReferenced(e(2), mg) THEN
      -- is e(2) the target of any e in mg?
      -- if not, delete e(2) from the graph
      delete(mg, {e(2)});
    END IF;
    END IF;
  END LOOP;

-- the next part of the procedure determines,
-- how to go on further: either
-- 1) to start from the root-Type, or
-- 2) analyze the elements of the operand list
IF isChanged THEN -- is the graph restructured?
  -- start analyzis from the beginning
  mg := tmerge(theType, mg);
ELSE
  -- work into depth with new operand list

    sl := getOperandList(getTypeName(curType), mg);
```

```
        FOR se IN sl |
          NOT se(2) IN visitedList AND
          getOperator(se(2)) /= "BASETYPE"
            AND
          se(2) /= curType
        LOOP
          mg := tmerge(se(2), mg);
        END LOOP;
      END IF; -- isChanged

      RETURN mg;
    END tmerge; -- procedure

  END flatten;

  --################# P R O C E D U R E ############################
  -- auxiliar procedure for checking, if a certain type aType
  -- is referenced from somewhere within the graph g
  PROCEDURE isReferenced(aType, g);
  VAR e := [];
    RETURN
      EXISTS e IN g | e(2) = aType;
  END isReferenced;

  --################# P R O C E D U R E ############################
  PROCEDURE getSubStruct(rd tn, rd g);
  VAR
    e, -- an edge in the graph g
    hg := {}; -- auxiliary structure,

-- BEGIN
     hg := getSS(tn,g, hg);
    -- add the path from root to hg
    IF EXISTS e in g | getOperator(e(1)) = TypeGraph.RootDesignator
                 AND getTypeName(e(2))=tn THEN
      RETURN hg with e;
    ELSE
      -- tn is not an explicit type!
      print("'", tn, "' is not declared! Exiting...");
      STOP; -- that should never happen
    END IF;

  END getSubStruct;

  -- ########################PROC#####################
  PROCEDURE getSS(rd tn, rd g, rd hg);
  VAR
    e; -- an edge in the graph g
-- BEGIN
    FOR e IN g | NOT (e IN hg) LOOP
      IF getTypeName(e(1))=tn THEN
        hg +:= getSS(getTypeName(e(2)), g less e, hg with e);
      END IF;
    END LOOP;
    RETURN hg;
  END getSS;
  -- ########################PROC#####################
```

```
PROCEDURE isSubType(rd s, rd p, rd g, rw ssb);
-- input:   typename s, typename p, type graph g,
--      a priori sub type definitions
-- output:  TRUE, is s is a subtype of p, FALSE otherwise
VAR
  alreadyChecked := {}, -- which nodes are already checked,
  -- used to avoid infinitly recalculating
  -- recursivle type definitions
  types := TypeGraph.getNodeSet(g);
-- BEGIN
  IF isSubT(s, p) THEN
    ssb with := [s, p, "<:"];
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;

  -- #######################PROC####################

  PROCEDURE isSubT(rd s, rd p); -- the main <: function
  -- BEGIN
    IF [s, p] IN alreadyChecked THEN
      RETURN TRUE; -- recursivley definitions are equal
    ELSE
      alreadyChecked with:= [s,p];
                   -- add the current check to the set
        IF s = p THEN -- types have identical names?
          RETURN TRUE; -- then they are identical types!
        ELSEIF TypeOpOf(s, types) = TypeOpOf(p, types) THEN
               -- AXIOM: identical types are always subtypes
          CASE TypeOpOf(s, types)
            WHEN TypeGraph.BaseTypeDesignator =>
               -- check if s and p are defined in the
               -- a priori set ssb (global to issubt)
              RETURN isAprioriSubT(s, p);
            WHEN "X" =>
              RETURN isCrossSubT(s,p);
            WHEN "U" =>
              RETURN isUnionSubT(s,p);
            WHEN "->" =>
              RETURN isFuncSubT(s,p);
            OTHERWISE =>
              RETURN FALSE;
            END CASE;
        ELSE
          RETURN FALSE; -- types have different
                        -- type operators
        END IF;
    END IF;
  END isSubT;

  -- #######################PROC####################

  PROCEDURE isUnionSubT(rd s, rd p);
  -- purpose: check subtype relations of U types
  VAR
```

```
    op_s := [], -- a element of the subtype operand list
          -- [typename%typeoperand, label]
    op_p := [];

    RETURN FORALL op_p IN getOperandList(p,g)
      | EXISTS op_s IN getOperandList(s,g)
        | op_s(1) = op_p(1) AND -- labels must be identical
          isSubT(break(op_s(2), TypeGraph.opsign),
                 break(op_p(2), TypeGraph.opsign)) ;

  END isUnionSubT;

  -- #########################PROC#####################

  PROCEDURE isFuncSubT(rd s, rd p);
  -- purpose: check subtype relations of -> types
  -- according to the contravariance rule for type substitution
  --
  -- input s, p type names, e.g. "person", "adr", ...
  VAR
    input_s := getFuncType("~I", getOperandList(s,g)),
        -- the input typename e.g. ~I1-xpT,
    input_p := getFuncType("~I", getOperandList(p,g)),
    output_s := getFuncType("~O", getOperandList(s,g)),
    output_p := getFuncType("~O", getOperandList(p,g));

  -- BEGIN

    RETURN isSubT(input_p, input_s) AND
            -- input must be contravariantly defined
       isSubT(output_s, output_p);
            -- output must be covariantly defined

  END isFuncSubT;


  -- #########################PROC#####################

PROCEDURE isCrossSubT(rd s, rd p);
-- purpose: check subtype relations of X-Types
-- input: s, p type names, e.g. "person", "adr", ...
VAR
  op_p := [], -- a element of the subtype operand list
  op_s := []; -- a element of the subtype operand list
          -- [typename%typeoperand, label]

-- BEGIN
    RETURN FORALL op_p IN getOperandList(p,g)
      | EXISTS op_s IN getOperandList(s,g)
        | op_s(1) = op_p(1) AND -- labels must be identical
          isSubT(break(op_s(2), TypeGraph.opsign),
                 break(op_p(2), TypeGraph.opsign)) ;
END isCrossSubT;

-- #########################PROC#####################

PROCEDURE isAprioriSubT(rd s, rd p);
```

```
  VAR td;
  -- BEGIN
  -- check if a defined subtype relation in ssb  for s and p exist
  -- ssb entered the scope via the procedure isSubType(...)
    RETURN EXISTS td IN ssb | td(1) = s AND td(2) = p;
  END isAprioriSubT;

END isSubType;

  -- ########################PROC####################

  PROCEDURE getOperandList(rd typename, rd g);
  -- returns the operand list of a type {[label,typename],,...}
  VAR oplist := {},
    graph := g; -- the global type graph

    FOR e in graph | break(e(1), TypeGraph.opsign) = typename LOOP
      oplist with:= [e(3), e(2)];
      -- operands with names e(3)/e(2) label/type
    END LOOP;
    RETURN oplist;
  END getOperandList;

  -- ########################PROC####################

  PROCEDURE getFuncType(rd side, rd operandset);
    -- input: which side-flag ("˜I", "˜O"),
    --       operandset of a function type e.g.
    --      '{["˜I", "˜I1-xpT%X"], ["˜O""˜O2-xpT%X"]}'
    --        which is a set of {[which-side-flag, typename%typeop]}
     RETURN break(operandset(side),TypeGraph.opsign);
      -- output: the type name
  END getFuncType;

    -- ########################PROC####################
  PROCEDURE delete(rw g, dellist);
  -- deletes all edges e given in the set dellist from graph g
  -- if source or target vertice (e(1), e(2))
  -- exists in the deletion list dellist
  VAR e := [];
    FOR e in g | e(1) IN dellist OR e(2) IN dellist  LOOP
        g less:= e;
    END LOOP;
  END delete;

  -- ########################PROC####################
  PROCEDURE rename(rw g, rd oldnode, rd newnode, addlabel);
  -- rename in graph g the old node with the name of newnode
  -- due to the set nature  of g this is a merging
  -- g is a set of edges of the form  [FROM, TO, LABEL]
  -- every substituted edge gets an unique name str(cnt+1) to avoid
  -- it from vanishing due to the set datastructure of graph g
  -- addlabel is the head of the new label

    VAR
      h := {}; -- auxiliary set structure
    -- BEGIN
```

```
  FOR e IN g | (e(1) /= newnode OR e(2) /= oldnode) LOOP
    -- [e(1), e(2), e(3)] =  [FROM, TO, LABEL]
    -- first check if the current edge is not the entered one,
    -- if yes, do not add this edge to the result
    -- because it will be renamed later on
    -- this avoids superfluous edges in the graph
    -- otherwise, substitute any occurence of oldnode by newnode
    IF e(1) = oldnode THEN
      e(1) := newnode; -- redirect edge
      e(3) := Util.squeeze(e(3))+">"+addlabel; -- new label
        -- squeeze eliminates white spaces in strings
    END IF;
    IF e(2) = oldnode THEN
      e(2) := newnode;
      e(3) := addlabel+"<"+Util.squeeze(e(3)); -- new label
    END IF;
    h with:= e; -- build intermediate result
  END LOOP;
  g := h; -- make intermediate result permanent
END rename;

-- #########################PROC#####################

PROCEDURE TypeOpOf(rd tn, rd types); -- returns the typeoperator
-- input: tn.. typename
--        types... set of types, injective
-- output: type operator of tn
  IF types = {} THEN -- only base types are defined in types
    RETURN TypeGraph.BaseTypeDesignator;
  ELSE
    RETURN types(tn);
  END IF;
END TypeOpOf;

-- #########################PROC#####################
PROCEDURE getOperator(rd typenode);
-- returns the current type operator of a vertice
  RETURN rbreak(typenode, TypeGraph.opSign);
END getOperator;

-- #########################PROC#####################
PROCEDURE getTypeName(rd typenode);
-- returns the type name of a vertice e.g. "real%BASETYPE"
  RETURN break(typenode, TypeGraph.opSign);
END getTypeName;

-- #########################PROC#####################
PROCEDURE areConstitutiveEqual(rd s, rd p, rd g);
  -- purpose: Check if s and p are type equal (without name check)
  -- input:   types s, p, type graph g
  -- output:  TRUE, if s and p base type equal, FALSE otherwise
  VAR
    types := TypeGraph.getNodeSet(g), -- set of type nodes
    alreadyChecked := {};

  -- BEGIN
    RETURN areCE(s, p);
```

```
PROCEDURE areCE(rd s, rd p);
  VAR
    t_p, t_s,   -- the type representation of s and p
          -- (name%Op)
    oplist_s := getOperandList(s,g),
      -- e.g.  oplist_s={["b [2]", "real%BASETYPE"],
                  --["a [1]", "a1-xpT%X"]}
    oplist_p :=  getOperandList(p,g);
-- BEGIN
  IF [s, p] IN alreadyChecked THEN
      -- avoid infinitely checking
      -- in case of recursive structures
    RETURN TRUE;
  ELSE
    alreadyChecked with:= [s,p];
    -- Definition "Constitutive equality (=_c)
    IF s = p THEN -- identical types
      RETURN TRUE;
    ELSEIF
      typeOpOf(s,types)=typeOpOf(p,types) THEN
      -- 1. s and p operand lists have the same length
      -- 2. Both types must have the identical operator
      IF  typeOpOf(s,types) =
        TypeGraph.BaseTypeDesignator THEN
      -- distinguish between func type and others
        RETURN FALSE;
        -- e.g. int is not constitutive equal to real
      ELSEIF typeOpOf(s,types) = "->" THEN
        RETURN
          areCE(
            getTypeName(oplist_s("~I")),
            getTypeName(oplist_p("~I")))
          AND
          areCE(
            getTypeName(oplist_s("~O")),
            getTypeName(oplist_p("~O")));
      ELSE  -- X or U check
        IF #oplist_s = #oplist_p THEN
          FOR t_s IN oplist_s LOOP
            IF EXISTS t_p IN oplist_p |
              areCE(
                getTypeName(t_s(2)),
                getTypeName(t_p(2))) THEN
              -- already validated operands should
              -- not be checked twice
              oplist_p less:= t_p;
            ELSE
              RETURN FALSE;
              -- no mapping between s and p
            END IF;
          END LOOP;
          RETURN TRUE;
          -- there exits a mapping from s to p
        ELSE
          RETURN FALSE; -- #oplist_s = #oplist_p
        END IF;
```

```
              END IF; -- typeOpOf(s,types) = "BASETYPE"
            ELSE
              RETURN FALSE; -- different type operators
            END IF; -- s=p
          END IF;-- [s, p] IN alreadyChecked
    END areCE;
  END areConstitutiveEqual;

END TypeRelation;
```

C

MSEQ implementation

The program `mseq` adapts the standard SEQUITUR algorithm. It is called within the SETL2 environment with `stlx mseq -i <sequencefile>] [-o <grammarfile>] [-n <length>] [-v] [-a] [-m [-b]] [-r <rep>]`. The synopsis of parameters is given below.

`-i <sequencefile>` The *i*nput file containing the sequences of events.

`-o <grammarfile>` specifies the name of the *o*utput file, where the grammar is put. Any existing file of the same name is overwritten. If this parameter is not specified, the default value is `sequencefile.grm`.

`-n <length>` Specifies the length *n* of the matching pattern. The default value is 2 (*di*gram).

`-v` Enables *v*erbosity. During analysis many more message are displayed.

`-a` Shows *a*ll generated rules. Does not employ rule utility, hence, underused rules are not deleted from the resulting grammar. This is useful for comprehending the overall process.

`-m` If multiple sequences are provided in the input file, the resulting grammars are *m*erged into one grammar.

- **`-b`** If this parameter is set, during mergin already existing *b*ase rules (terminal symbols on the right hand side of a rule only) from the individual grammars are reused as building blocks.

-r <rep> Specifies the *r*epetition heuristics. If there exists at least `<rep>` patterns repeated in the sequence, they are abstracted to one iteration.

The program `mseq` is not modularized into packages, since there are no functional parts within which are reusable in another context.

C.1 The `mseq` program

```
-- implementation of a sequitur variants

PROGRAM mseq;
-- definition part

CONST
  lIt := "(", rIt := ")+",
  altSym := "|", -- alternative rule symbol
  ruleSym := "$", -- symbol to denote rules
  StartSym := rulesym+"$",
  epsilon := "?";

VAR
  arguments := {}, -- the map of all command line arguments
  m, -- the domain knowledge about the minimal occurence of patterns
    -- to be considerds as iterations
  ruleGen := 0, -- used for generating unique rule designators
  grammar := {}, -- a map {["$S", ["$A","c","$A","d"]],
    --                          ["$A", ["b","c"]],...}
  mergedGrammar := {}, -- the grammar generated from all sequences
  grammars := [], -- sequence of all grammars generated from words
  nGram := [], -- holds the current ngram
  word := [], -- the current input
  token := "", -- a symbol from the input
  fHandle, -- a system file handle
  -- global variable
  verbose := FALSE, -- global verbose flag, used in "say"
  r := 0, -- repetition check indicator
  n := 2, -- the default length n for the n-gram
  seqFile, -- the file with the input sequence
  outFile;

-- BEGIN

  IF NOT CommandLineParse(arguments) THEN
    Usage();
  ELSEIF (n := arguments("-n")?n) < 2 THEN
    print("n-grams must be larger than '1', current value='",n,"'!");
  ELSE
    -- should the program be verbose
    verbose := arguments("-v")?FALSE;
    seqFile := arguments("-i");
    IF seqFile /= om -- input file provided?
      AND fexists(seqFile) THEN -- input file exists?
```

```
outfile := arguments("-o");

IF outfile = om THEN outfile := seqfile+".grm"; END IF;

-- open the sequence file
fHandle := open(seqfile, "text-in");

WHILE  (word := readWord(fhandle)) /= [] LOOP
-- while there are words in the file
  say("Word: "+ str(word));
  grammar := {[StartSym, []]};

  UNTIL word = [] LOOP

    IF (token := getToken(word)) /= om THEN
      nGramUniqueness(token, grammar, n);
    END IF; -- invalid token

  END LOOP; -- until eof

  IF NOT(arguments("-a")?FALSE) THEN
  -- should all rules be presented?
    RuleUtilityCheck(grammar);
  ELSE
    say("No Rule Utility Check...");
  END IF;

  IF (r := arguments("-r")?0) > 0 THEN -- repetition check
    RepetitionCheck(grammar, r);
    -- additional RUC, because RC changes grammar!
    IF NOT(arguments("-a")?FALSE) THEN
    -- should all rules be presented?
      RuleUtilityCheck(grammar);
    END IF;

  END IF;

  IF arguments("-m")?FALSE THEN -- multiple grammar merge
    -- merge current grammar with the general grammar
    MultipleGrammarMerge(mergedGrammar, grammar);
  END IF;

  -- add the grammar to the set of yet built grammars
  grammars with:= grammar;
END LOOP; -- while words in file

close(fHandle);
PrintGrammars(grammars, n, outfile);

IF arguments("-m")?FALSE THEN -- merging grammars?

  IF arguments("-b")?FALSE THEN -- base rule reuse required
    say("Reusing base rules");
    ReUseAtoms(mergedGrammar);
  END IF;
```

```
        say("-------GRAMMAR-MERGING-----------\n"+
        "Naive merged grammar\n"+ printgrammar(mergedGrammar));
        -- check for repeated patterns in the merged grammar
        GrammarNGramUniqueness(mergedGrammar,n);

        IF NOT(arguments("-a")?FALSE) THEN

          -- should all rules be presented?
          say("Merged Grammar: Rule Utility Check");
          RuleUtilityCheck(mergedGrammar);
        END IF;
        print("================\nMerged Grammar:\n",
            PrintGrammar(mergedGrammar));
      END IF; -- merging grammars?
    ELSE -- is there an input file?
      Usage();
      print("Non-Existing sequence input file '", seqfile,
          "'! \n... Aborting");
    END IF;
  END IF; -- Commandlineparse

--######### THE    P R O C E D U R E    S E C T I O N ###########

  --#########################PROC####################
  --### RuleUtilityCheck                           ###
  --#########################PROC####################

  -- checks if a nont-terminal symbol is used more than once
  -- if it is used only once, it is substiuted by its body.

  PROCEDURE RuleUtilityCheck(rw grammar);
  VAR
    rbody :=[], -- auxiliary variable for rhs of rule
    useCount := {}; -- map stores the number of used rules
  -- BEGIN

  FOR rhs IN range grammar LOOP

    FOR e IN rhs | e(1)=RuleSym LOOP
      IF NOT (e IN DOMAIN useCount) THEN
        useCount with:= [e, 0];
      END IF;
      useCount(e) := useCount(e)+1;
    END LOOP;
  END LOOP;

  -- now substitute every underused rule by its rhs
  FOR lhs IN domain useCount| useCount(lhs) <= 1 LOOP
    rbody := grammar(lhs);
    Substitute([lhs], rbody, grammar);
    -- delete underused rule from grammar
    grammar less:= [lhs, rbody];
    say("deleting underused rule: "+str(lhs)+" -> "+str(rbody));
  END LOOP;
  END RuleUtilityCheck;

  --#########################PROC####################
```

```
--### nGramUniqueness                            ###
--######################PROC####################
PROCEDURE nGramUniqueness --(token, grammar, n)
  (   rd tk, -- the just read token
      rw grammar, -- the current grammar
      rd n -- the length of the pattern n-gram
   );
  VAR
    newRule := [], -- a new rule
    i := 1, -- position counter for rhs
    ngramFound := FALSE,  -- counter for the occurence of ngrams
    hrrhs := "", -- the highest rule right hand side
    ngram := []; -- the current n-gram variable
-- BEGIN
  -- determine the current n-gram, which is the curren token
  -- on the last place an the n-1 tokens from the end of $$

  -- get the rhs of the highest rule ($$)
  hrrhs := grammar(startSym);

  IF (#hrrhs >= n) THEN
    -- get the tail of highest rhs
    -- and build ngram with n-1 tail

    ngram := hrrhs(#hrrhs-n+2..)+[tk];

    -- ensure digram uniqueness for the whole grammar
    -- r is e.g. ["$S", ["a", "b"]]
    -- 1. case ngram 2nd time in $S
    IF SubTuple(ngram, hrrhs) > 0 THEN -- pattern already existing
      -- generate new rule for multiple ngram patterns
      newRule := [NextRule(), ngram];
      Insert(tk, grammar); -- append token to start rule
      -- substitute each occurence of ngram
      -- in grammar by newRule-designator
      Substitute(ngram, newRule(1), grammar);
      -- now add new rule to grammar
      grammar with:= newRule;
    ELSE -- check, if ngram is not already rhs of some rule

      FOR r IN grammar | r(1) /= StartSym LOOP
      -- check every rule in grammar
        IF SubTuple(ngram, r(2)) > 0 THEN
        -- ngram found in some rhs-body
          ngramFound := TRUE;
          -- 2. case: rule body = ngram =>
          --          substitute ngram by r(1)
          IF r(2) = ngram THEN
            Insert(tk, grammar);

            Substitute(ngram, r(1), grammar);
            -- restore the original rule,
            -- which was destroyed by Substitue
            grammar(r(1)) := ngram;
            -- check recursively, if there are
            -- newly generated patterns there
            hrrhs := grammar(StartSym);
```

```
            grammar(startSym) := hrrhs(1..#hrrhs-1);
            -- now grammar has changed, the ngram uniqueness
            -- must be checked again
            nGramUniqueness(hrrhs(#hrrhs), grammar, n);

          END IF; -- body = ngram
        END IF; --  ngram found

      END LOOP; -- every rhs check
      IF NOT(ngramFound) THEN -- simple action: add the new token
        Insert(tk, grammar);
      END IF;
    END IF; -- SubTuple(ngram, hrrhs) > 0
  ELSE --   input sequence smaller than n-gramm
    Insert(tk, grammar);
  END IF; -- (#hrrhs >= n)
END nGramUniqueness;

--########################PROC#####################
--### GrammarNGramUniqueness                     ###
--########################PROC#####################
-- checks the grammar, if there are nPatterns
-- (including rule-symbols)
-- which occur multiple times in the whole grammar. If there are
-- repetitions, the nPattern is substituted by a new rule symbol
-- (or a rule already existing in the grammar)

PROCEDURE GrammarNGramUniqueness(rw generalGr,n);
VAR
  auxGr := generalGr,
  newrule := [],
  r := [], --iterator over ruleset
  npr, -- npattern-rule,  -> npattern
  rhs := [],
  npattern := [];

-- BEGIN

  -- now apply thr g-sequitur (grammar sequitur)
  -- get the n-pattern built from rhs
  WHILE auxGr /= {} LOOP
    r := arb auxGr; -- get arbitrary element from set
    auxGr less:= r;
    WHILE (npattern := BuildNPattern(r(2), n)) /= [] LOOP
    -- now check for 2nd (or more) occurence of npattern
      IF (SubTuple(npattern, r(2)) > 0)
      -- 2nd occurence of npattern in current rhs
      OR
      -- does npattern appear somewhere else on rhs?
        (EXISTS rhs IN range generalGr less generalGr(r(1)) |
          SubTuple(npattern, rhs) > 0)
      THEN
      -- checking for ev. superflous rules,
      -- which could be introduced by substituting rules
      -- of the form X -> ngram
        IF (EXISTS npr IN generalGr | npr(2) = npattern) THEN
```

```
          -- a rule for that npattern exists already
          -- reuse it
          generalGr less:= npr; -- backuo existing rule
          -- substitute pattern by existing rule name
          Substitute(npattern, npr(1), generalGr);
          generalGr with:= npr; -- restore rule
          say("Multiple rhs, reusing rule "+str(npr));
        ELSE
          -- pattern occurs multiply, making a rule for it
          -- new rule introduction
          newrule := [NextRule(), npattern];

          -- substitute pattern by new rule name
          Substitute(npattern, newrule(1), generalGr);
          generalGr with:= newrule;
          say("Multiple rhs occurence, introducing rule "+
              str(newrule));
        END IF;
        -- new situation, npattern match starts from beginning
        auxGr := generalGr;
      END IF; -- npattern subtuple in rhs?

    END LOOP;
  END LOOP;

END GrammarNGramUniqueness;

--########################PROC####################
--### ReUseAtoms                                  ###
--########################PROC####################
PROCEDURE ReUseAtoms(rw g);
VAR
  i,
  atomRules := {};
  -- get all rules with no rulesyms on rhs
  FOR r IN g LOOP
    i := 1;
    WHILE i <= #r(2) AND r(2)(i)(1) /= ruleSym LOOP
      i +:= 1;
    END LOOP;
    IF i > #r(2) THEN
      atomRules with:= r;
    END IF;
  END LOOP;
  -- substitute all other occurences of atoms rhs
  FOR ar IN atomRules | ar IN g LOOP
    g less:= ar; --  current atom should not be substituted
    Substitute (ar(2), ar(1), g);
    g with:= ar; -- restore atom
  END LOOP;
END ReUseAtoms;

--########################PROC####################
--### RepetitionCheck                           ###
--########################PROC####################
PROCEDURE RepetitionCheck(rw g, rd r) ;
-- detects iteration of at least r patterns in grammar g
```

```
VAR
  mxs, -- sequence with max length
  pos := 0,
  ps; -- set of all subsequences (powersequence)

-- BEGIN
  FOR rule IN g LOOP

    ps := PowerSeq(rule(2));
    -- get seq with maximal length from set of powerseq ps
    -- this maxseq (mxs) is deleted from ps
    WHILE (mxs := MaxSeq(ps)) /= om LOOP

      -- check, if maxseq appears sequentially r-times in rule
      IF SubTuple(mxs*r, rule(2)) > 0  THEN
        -- Repetition handling
        say("Found repetition '"+str(mxs)+
            "' in rule "+str(rule));
        SeqToIteration(mxs, r, g);
      END IF;
    END LOOP;

  END LOOP; -- for all rules in grammar
END RepetitionCheck;

--########################PROC####################
--### MaxSeq                                     ###
--########################PROC####################
PROCEDURE SeqToIteration (tkSeq, minRep, rw g);
-- replace every occurence of minrep*tkSeq by tkSeq
-- in the grammar g and introduces repetition meta symbols (, )+
VAR
  rep, -- current repetition count
  s, -- subtuple
  pos; --position of first sequence match
-- BEGIN
  FOR r IN g LOOP
    -- search for tkSeq
    WHILE (pos := SubTuple(tkSeq*minRep, r(2))) > 0 LOOP
      rep := minRep;
      -- check after first sequence occurence
      s := r(2)(pos+(minRep*#tkSeq)..);

      -- check if more repetitions exist
      WHILE #s >= #tkseq AND s(1..#tkseq) = tkseq LOOP
      -- immediate neighbour is also tkSeq
        rep +:= 1;
        s := s(#tkSeq+1..);
      END LOOP;

      -- replace pattern sequence by newpattern
      r(2) := r(2)(1..pos-1)+[lIt]+tkSeq+[rIt]
                  +r(2)(pos+#tkSeq*rep..);
    END LOOP;
    -- update the grammar
    g(r(1)) := r(2);
  END LOOP;
```

```
END SeqToIteration;

--########################PROC####################
--### MaxSeq                                    ###
--########################PROC####################
PROCEDURE MaxSeq(rw seq);
-- returns the longest sequence m in seq-set.
-- if nothing is found, om is returned.
-- side effect: m is deleted from seq-set
VAR
  m,s;

  IF (exists m in seq |
      (forall s IN seq | #m >= #s))
  THEN
    seq less:= m;
  END IF;
  RETURN m;
END MaxSeq;

--########################PROC####################
--### PowerSeq                                  ###
--########################PROC####################
PROCEDURE PowerSeq(rd t);
-- generates all sequences from a tuple t
VAR
  pss := {};  -- the PowerSequence set

-- BEGIN
  FOR i IN [1..#t] LOOP
    FOR j IN [i..#t] LOOP
      pss with:= t(i..j);
    END LOOP;
  END LOOP;
  RETURN pss;
END PowerSeq;


--########################PROC####################
--### BuildNPattern                             ###
--########################PROC####################
PROCEDURE BuildNPattern(rw rhs, rd n);
-- builds pattern of length n from tuple rhs
VAR
   t := rhs,
  i := 1,
  p := [];

-- BEGIN
  WHILE t(i) /= om AND i <= n LOOP
    IF t(i)  /= altSym AND -- no new word
      NOT (t(i) IN {lIt, rIt})
      -- pattern must not span iterations
    THEN
      p with:= t(i);
      i +:= 1;
```

```
    ELSE  -- next rule body or iteration body
      t := t(i+1..);
      p := [];
      i := 1;
    END IF;
  END LOOP;
  IF i <= n THEN -- invalid nPattern
    p := [];
    rhs := t;
  ELSE
    rhs := rhs(2..);
  END IF;
  RETURN p;
END BuildNPattern;

--########################PROC####################


--### MultipleGrammarMerge                      ###
--########################PROC####################
PROCEDURE MultipleGrammarMerge (rw mg, rd g);
-- merged grammar mg represents all yet built grammars
-- if a rule's rhs of g matches with a rhs of mg, in mg
-- this rule is substituted by g's one.
-- if g's rule does not match any rhs, it is an alternative
-- and its concatenated to the start rule of mg.
VAR
  newrule := [], -- if a startrule must be substituted
  gr := []; -- a general rule from mg

-- BEGIN
  IF mg = {} THEN -- first grammar encountered
    mg := g;
  ELSE  -- Generate a unified, merged Grammar from all words
    FOR r IN g | r NOTIN mg LOOP
    -- check every rule in grammar g, when it is not already in mg

      IF EXISTS gr IN  mg | Eq(gr(2), mg, r(2), g) THEN
      -- is there a rhs-match
      -- replace every symbol of general rule
      -- by new symbol from grammar
        IF r(1) = StartSym THEN
          -- rhs already in general merged grammar
          IF gr(1) /= StartSym AND
            NOT RHSMatch(gr(1), mg(StartSym))
            -- not yet in startrule
          THEN
            Insert(altSym, mg); -- alternative encountered
            Insert(gr(1), mg);
          END IF;
        ELSEIF gr(1) = StartSym THEN
          mg with:= r;
        ELSE
          -- no start symbol involved, replace all
          -- old references in mg with rule from g
          replace(gr(1), r(1), mg);
        END IF;
```

```
      ELSE
        -- no rhs-match
        IF r(1) = StartSym THEN
          Insert(altSym, mg); -- alternative encountered
          newrule := [NextRule(), r(2)]; -- generate new rule
          Insert(newrule(1), mg);
        ELSE -- add not yet merged rule to merged grammar mg
          newrule := r;
        END IF;
        mg with:= newrule; -- add the new rule to merged grammar
      END IF;
    END LOOP;
  END IF;
END MultipleGrammarMerge;


--########################PROC####################
--### Eq                                         ###
--########################PROC####################

-- Eq checks for structural equality of two tuples
-- Eq considers following requirements:
  -- Rulesymbols are always equal
  -- A rhs with "rhs1 | rhs2" is checked with s=rhs1 and s=rhs2

PROCEDURE Eq(rd s1, rd g1, rd s2, rd g2);
VAR
  alt1, alt2,
  found := FALSE,
  i := 1;

-- BEGIN
  LOOP
    -- empty rules are always equal
    IF s1(i) = om AND s2(i) = om THEN
      found := TRUE;
    -- alternative symbol marks end of rhs as well
    ELSEIF s1(i) = om AND s2(i) = AltSym THEN
      found := TRUE;
    ELSEIF s2(i) = om AND s1(i) = AltSym THEN
      found := TRUE;
    -- only one rhs is empty, the other one not
    ELSEIF s1(i) = om OR s2(i) = om THEN
      EXIT;
    -- symbols match
    ELSEIF s1(i) = s2(i) THEN   -- matching
      i +:= 1; -- go in with next symbol
    ELSEIF -- rule symbols must be treated separately
      (s1(i)(1) = ruleSym AND s2(i)(1) = RuleSym)
    THEN
      IF Eq(g1(s1(i)), g1, g2(s2(i)), g2) THEN
      -- check depth equality (without considering symbol names)
        i +:= 1; -- go on with next symbol
      ELSE -- no deep equality
        EXIT;
      END IF;
    ELSEIF s1(i) /= s2(i) THEN  -- no matching
```

```
      -- start with next alternative word
      IF (alt1 := SubTuple([altSym], s1)) > 0 THEN
        s1 := s1(alt1+1..);
        i := 1;
      END IF;
      IF (alt2 := SubTuple([altSym], s2)) > 0 THEN
        s2 := s2(alt2+1..);
        i := 1;
      END IF;
    END IF;

    -- Loop exit conditions, setl2 has no pass-through loop
    IF alt1=0 AND alt2=0 THEN
      -- there were no alteratives in both rhs
      EXIT; -- nothing to find
    ELSEIF found THEN
      EXIT;
    END IF;

  END LOOP;
  RETURN found;
END Eq;
--########################PROC####################
--### RHSMatch                                  ###
--########################PROC####################
PROCEDURE RHSMatch(s, t);
-- gets two tuples as input, returns when s is a rhs in
-- t, such that alternatives ("|") are considered,
-- e.g.  [A, B] is in [A, C | C, D | A, B]
VAR pos := 1;
IF NOT is_tuple(s) THEN s := [s]; END IF;
IF (pos := SubTuple(s,t)) > 0 THEN
  IF pos  = 1 OR t(pos-1) = altSym THEN -- match possible
    IF pos+#s  > #t OR t(pos+#s) = altSym THEN
      RETURN TRUE;
    END IF;
  END IF;
END IF;
RETURN FALSE;
END RHSMatch;
--########################PROC####################
--### Insert                                    ###
--########################PROC####################
PROCEDURE Insert(rd tk, rw grmr);
VAR rhs := grmr(StartSym);
-- BEGIN
  grmr(startSym) := rhs with tk;
END Insert;

--########################PROC####################
--### NextRule                                  ###
--########################PROC####################
PROCEDURE NextRule();
-- Generates a new unique rule namr
CONST
  ruleSyms :=  [65..90]+[97..122] ;
VAR
```

```
  rn := ruleSym,
  r := 0,
  z := ruleGen,
  b := #ruleSyms;
-- BEGIN
  -- RETURN rn +:= str(ruleGen +:= 1);
  FOR i IN [0..(ruleGen / b)] LOOP
    r := z MOD b;
    z := z / b;
    rn +:= char(ruleSyms(r+1));
  END LOOP;
  ruleGen +:= 1;
  RETURN rn;
END NextRule;

--########################PROC#####################
--### readWord                                  ###
--########################PROC#####################
PROCEDURE readWord(rd fh  -- filehandle
            );
  VAR
    invalid := {"%", rulesym},
    str := "",
    tk := "",
    wrd := [];

--BEGIN
  geta(fh, str);
  WHILE NOT(eof()) AND str /= "" LOOP
    -- strip leading spaces
    WHILE str /= "" AND str(1) IN " \t\n" LOOP
      str := str(2..);
    END LOOP;
    -- be aware of comments in input
    WHILE str /= "" AND NOT (str(1) IN invalid)  LOOP
      IF str(1) IN " \t\n" THEN
        wrd with:= tk;
        tk := "";
        -- only one white space divides tokens, the rest not
        WHILE str /= "" AND str(1) IN " \t" LOOP
          str := str(2..);
        END LOOP;
      ELSE
        tk +:= str(1);
        str := str(2..);
      END IF;
    END LOOP;

    IF tk /= "" THEN wrd with:= tk; tk := ""; END IF;
    geta(fh, str);
  END LOOP;
  RETURN wrd;
END readWord;

--########################PROC#####################
--### getToken                                  ###
--########################PROC#####################
```

```
PROCEDURE getToken(rw word -- tuple of tokens
            );
VAR
  token;
--BEGIN
  token := word(1);
  word := word(2..);
  RETURN token;

END getToken;

--########################PROC####################
--### SubTuple                                  ###
--########################PROC####################

-- checks, if u is in v and returns the index of the first element
-- in v witch starts match
-- if nothing was found, 0 is returned
PROCEDURE SubTuple(rd u, rd v);
VAR
  i := 1;
-- BEGIN
  IF #u <= #v THEN -- matching possible
    WHILE i+#u-1 <= #v LOOP
      IF u = v(i..i+#u-1) THEN -- match
        RETURN i;
      ELSE    -- no match
        i +:= 1;
      END IF;
    END LOOP;
  END IF;
  RETURN 0;
END SubTuple;

--########################PROC####################
--### Substitute                                  ###
--########################PROC####################

PROCEDURE Substitute(rd oldtuple, rd newtuple, rw map);
-- substitute occurence of oldtuple in map with newtuple
VAR pos := 0;
-- BEGIN
IF NOT(IS_TUPLE(newtuple)) THEN
  newtuple := [newtuple];
END IF;
IF NOT(IS_TUPLE(oldtuple)) THEN
  oldtuple := [oldtuple];
END IF;
FOR r IN map LOOP -- e.g. r= ["$D", ["$C", "$A"]]
  WHILE (pos := SubTuple(oldtuple, r(2))) > 0 LOOP
  -- there was a matching
    r(2) := r(2)(1..pos-1)+newtuple+r(2)(pos+#oldtuple..);
  END LOOP;
  map(r(1)) := r(2);
END LOOP;

END Substitute;
```

```
  --########################PROC####################
  --### REPLACE                                  ###
  --########################PROC####################
  PROCEDURE Replace(rd oldsym, rd newsym, rw grammar);
  -- replaces oldsymbol by new symbol in grammar
  VAR h, pos;
  -- BEGIN
  FOR r IN grammar LOOP
    h := r(1) ;
    pos := 1;
    IF r(1) = oldsym THEN r(1) := newsym; END IF;

    WHILE pos <= #r(2) LOOP
      IF r(2)(pos) = oldsym THEN
        r(2)(pos) := newsym;
      END IF;
      pos +:= 1;
    END LOOP;
    grammar lessf:= h;
    grammar with:= r;
  END LOOP;
  END Replace;

  --########################PROC####################
  --### PrintGrammars                            ###
  --########################PROC####################
  PROCEDURE PrintGrammars(rd grms, n, file);
  VAR
    i := 1,
    t := "Grammar(s): "+str(#grms)+"\tLength of ngram-pattern: "
        +str(n)+"\n",
    filehdl  := open(file, "text-out");

-- BEGIN
    FOR g IN grms LOOP
      t +:= "\n"+str(i)+". Grammar ("+str(#g)+" rule(s)):\n";
      i +:= 1;
      t +:= PrintGrammar(g);
    END LOOP;
    print(t);
    printa(filehdl, t);
    close(filehdl);

  END PrintGrammars;
  --########################PROC####################
  --### PrintGrammar                             ###
  --########################PROC####################
  PROCEDURE PrintGrammar(rd grmr);
  CONST separator := " ";
  VAR
    text := "",
    toprint := {startsym}, -- the symbols to print
    printed := {},
    s := startsym;

-- BEGIN
```

```
  -- startsymbol
  LOOP
    s from toprint;
    IF NOT(s IN printed) THEN
      text +:= s +"\t->\t";

      FOR r IN grmr(s) LOOP

        IF r(1) = rulesym THEN
          toprint with:= r;
        END IF;
        IF r = altSym THEN
          text +:= r+"\n\t\t";
        ELSE
          text +:= str(r)+separator;
        END IF;
      END LOOP;
      text +:= "\n";
      printed with:= s;
    END IF;
    IF toprint = {} THEN EXIT; END IF;
  END LOOP;
  RETURN text;
END PrintGrammar;

--########################PROC####################
--### say                                       ###
--########################PROC####################
PROCEDURE say(rd m);
  IF verbose THEN
    print(">   ",m);
  END IF;
END say;

--########################PROC####################
--### CommandLineParse                          ###
--########################PROC####################

PROCEDURE CommandLineParse(rw argv);
-- input: a map of commandline-Options and current values
VAR i := 1;
  WHILE command_line(i) /= om LOOP
    CASE command_line(i)
      WHEN "-i" =>
        argv with:= ["-i", command_line(i+1)]; -- inputfile
        i +:= 2;
      WHEN "-o" =>
        argv with:= ["-o", command_line(i+1)]; -- output file
        i +:= 2;
      WHEN "-n" => -- n-gram length
        argv with:= ["-n", unstr(command_line(i+1))];
        i +:= 2;
      WHEN "-v" =>  -- verbose
        argv with:= ["-v", TRUE];
        i +:= 1;
      WHEN "-a" =>  -- no simplify, full set of rules
        argv with:= ["-a", TRUE];
```

```
        i +:= 1;
      WHEN "-m" => -- merging multiple grammars
        argv with:= ["-m", TRUE];
        i +:= 1;
      WHEN "-b" => -- reuse base rules
        argv with:= [command_line(i), TRUE];
        i +:= 1;
      WHEN "-r" => -- repetition check
        argv with:= [command_line(i), unstr(command_line(i+1))];
        i +:= 2;
      OTHERWISE =>
        print("unknown command line parameter '",
            command_line(i),"'!");
        RETURN FALSE;
      END CASE;
    END LOOP;
    RETURN TRUE;
  END CommandLineParse;

  --########################PROC####################
  --### Usage                                    ###
  --########################PROC####################

  PROCEDURE Usage();
    print(
      "\nUSAGE: stlx mseq -i <sequencefile> [-o <grammarfile>]",
      " [-n <length>] [-v] [-a] [-m [-b]] [-r <rep>]\n",
      "Synopsis:",
      "\n\t-n ... length of matching pattern (default = 2)",
      "\n\t-v ... Verbose",
      "\n\t-a ... All rules, underused rules will not be deleted",
      "\n\t-m ... Merging multiple grammars into general grammar",
      "\n\t-b ... reuse Base rules of base grammars"+
      " (during  merging)",
      "\n\t-r ... find Repetitions of at least <rep> times"
      );
  END Usage;

END mseq;
```

Lebenslauf

Dipl.-Ing. Heinz Pozewaunig
Neckheimgasse 26/17
A-9020 Klagenfurt

Heinz Pozewaunig wurde am 17. Oktober 1967 in Klagenfurt geboren. Nach der Pflichtschule begann er eine Lehre als industrieller Schuhfertiger, bevor er ab 1984 die Handelsakademie Feldkirchen besuchte. Nach der Matura, die er 1989 mit aus gezeichnetem Erfolg ablegte, und dem Wehrdienst, arbeitete er bis zum Studienbeginn im Herbst 1990 als Sachbearbeiter bei einer Versicherungsgesellschaft. Das Studium der Angewandten Informatik an der Universität Klagenfurt schloss er im November 1996 ab. Die Diplomarbeit behandelte den Aspekt der Modellierung und Integration von Zeitrestriktionen für Workflow-Managementsysteme.

Bevor er im Februar 1997 als Vertragsassist am Institut für Informatik zu arbeiten begann, war Heinz Pozewaunig bei einer Klagenfurter Firma an der Entwicklung eines internet-basiertes Projektmanagement-Werkzeugs beteiligt. Ab Jänner 1998 war er als Universitätsassistent am Institut für Informatik-Systeme in der Forschungsgruppe “Informatik unter Berücksichtigung der betrieblichen Anwendung” von Professor Mittermeir tätig.