Core Technologies for System Renovation

Mark van den Brand[1], Paul Klint[2,1], Chris Verhoef[1] *

[1] Programming Research Group, University of Amsterdam, Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
[2] Department of Software Technology, Centre for Mathematics and Computer Science, P.O. Box 4079, NL-1009 AB Amsterdam, The Netherlands
e-mail: markvdb@wins.uva.nl, paulk@cwi.nl, chris@wins.uva.nl

Abstract. Renovation of business-critical software is becoming increasingly important. We identify fundamental notions and techniques to aid in system renovation and sketch some basic techniques: generic language technology to build analysis tools, a knowledge retrieval system to aid in program understanding, and a coordination architecture that is useful to restructure monolithic systems thus enabling their renovation. We argue that these techniques are not only essential for the renovation of old software but that they can also play an important role during the development and maintenance of new software systems.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.2.m [**Software Engineering**]: Miscellaneous—Rapid prototyping; D.3.2 [**Programming Languages**]: Language Classifications—Specialized application languages; E.2 [**Data**]: Data Storage Representations—Composite structures
Additional Key Words and Phrases: re-engineering, system renovation, intermediate data representation, coordination language, query algebra

1 Introduction

There is a constant need for updating and renovating business-critical software systems for many and diverse reasons: business requirements change, technological infrastructure is modernized, the government changes laws, or the third millennium approaches, to mention a few. So, in the area of software engineering the subjects of program understanding and system renovation become more and more important, see, for instance, [BKV96] for an annotated bibliography. The interest in such subjects originates from the difficulties that one encounters when

* The authors were all in part sponsored by bank ABN AMRO, software house DP-Finance, and the Dutch Ministry of Economical Affairs via the Senter Project #ITU95017 "SOS Resolver". Chris Verhoef was also supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), project *Interactive tools for program understanding*, 612-33-002.

attempting to maintain large, old, software systems. It is not hard to understand that it is very difficult—if not impossible—to renovate such *legacy* systems.

The purpose of this paper is to identify core technologies for system renovation and to incorporate reverse engineering techniques in forward engineering so that the maintenance problem may become more manageable in the future. We identify the following core technologies for system renovation:

- Generic language technology.
- Techniques for extracting information from existing source code.
- Visualization of extracted information.
- Decomposition of systems into smaller components.
- Coordination of (heterogeneous) components.
- Construction techniques for components.

Why are these technologies essential for system renovation? Before the actual renovation can start it will be necessary to make an inventory of the specification and the documentation of the system to be renovated. It is our experience that either there is no documentation at all, or the original programmers that could possibly explain the functionality of parts of the system have left, or both. The only documentation that is left is the source code itself. Thus, since the vital information of the software is solely accessible via the source code, techniques are needed for understanding this code. Note that legacy systems are usually polylingual, hence generic language technology is desirable to obtain analysis tools for all these languages. Given the results of program understanding (logic) components and their relationships in the system can be identified. The latter can be used to restructure the system via coordination techniques, this creates the possibility to redesign the extracted (logic) components into physical ones.

We will briefly review some of these technologies in this paper. The presentation is largely determined by our own current research and practice in re-engineering and does not aim at completeness.

First we introduce some basic terminology in re-engineering and generic language technology.

1.1 Re-engineering

We briefly recall some re-engineering terminology as proposed in [CC90]. The term reverse engineering finds its origins in hardware technology and denotes the process of obtaining the specification of complex hardware systems. Now the meaning of this notion has shifted to software. While *forward engineering* moves from a high-level design to a low-level implementation, *reverse engineering* can be seen as the inverse process. It can be characterized by analyzing a software system in order to, firstly, identify the system components and their interactions, and to, secondly, make representations of the system on a different, possible higher, level of abstraction. Reverse engineering restricts itself to *investigating* and *understanding* a system, and is also called *program understanding*. Adaptation of a system is beyond reverse engineering but within the scope of system renovation. The notion *restructuring* amounts to transforming a system from one representation to another one at the same level of abstraction. An essential aspect of restructuring is that the semantic behaviour of the original system and the new one should remain the same; no modifications of the functionality is involved. The purpose of *re-engineering* or *renovation* is to study the system, by making a specification at a higher abstraction level, adding new functionality to this specification and develop a completely new system on the basis of the original one.

1.2 Generic Language Technology

Another issue we will encounter is to what extent various tools depend on specific programming languages. We will classify language (in)dependence in the following categories:

- We call a system *language-independent* if it has *no* built-in knowledge of a specific language. An example is the UNIX[3] tool `grep(1)`, that can be used for simple textual searches in source files.
- We call a system *language-dependent* if the knowledge of a language is hard-wired in the system, e.g., a C-compiler. This knowledge can be implemented in the system by hand, via a generator, or via a combination of these approaches.
- We call a system *language parameterized* (or *generic*) if the language is a parameter of the system and upon instantiation with a language definition a language-specific system is obtained. Examples are the Synthesizer Generator [RT89] and the ASF+SDF Meta-Environment [Kli93, DHK96].

1.3 Related Work

There are not many scientific papers that actually discuss the re-engineering of large software systems, as we discovered during the compilation of an annotated bibliography [BKV96]. There are probably two reasons for this. First, academic software is easily put aside, because there is in general no real economic necessity to keep it running. Second, re-engineering projects carried out in industry are not frequently published for commercial reasons. This paper is based on our experiences in re-engineering a large academic software system as well as on the cooperation with a large commercial bank and a software house specialized in financial software.

We discuss a knowledge retrieval system based on a query algebra that is language parameterized and interprets queries directly. It supports multidirectional traversal of the syntax trees representing the code of legacy systems. Some related work on the topic of knowledge retrieval systems is [Ode93, PP94a, PP94b, REQ94, DRW96]. In [Ode93] a query mechanism is used for a completely different purpose than ours, namely, to define type checkers, the tree traversal

[3] UNIX is a registered trademark of UNIX System Laboratories

mechanism is, however, similar to ours. In [PP94a, PP94b] it is claimed that the described Source Code Algebra is language-independent, but the data base in which the source code is stored must be initialized with language specific entries. Adding a new language results in adapting the data base. The Source Code Algebra is conceptually language parameterized, but the implementation technique used required instantiation of language parameters, thus yielding a language-dependent system. All examples in [PP94b] only use top-down tree traversal. [DRW96] describes tools for analysing C/C++ programs for program understanding. These tools are generated and support a procedural mechanism to retrieve information from the C/C++ programs. So it is not based on a query algebra and is language-dependent. REQL [REQ94] is a source code query language. The syntactic notions in a language are translated to attributes in a data base and into queries to test for relationships and properties. These attributes are hard-wired in the system but can be used for more than one language. The visualization mechanism and querying mechanism are fully integrated. This is a disadvantage since it prevents the visualization of results obtained by other mechanisms. It also supports some procedural functionality to analyze programs. An implementation of the query language REQL is described that permits the connection of parsers for different languages.

We will also discuss a coordination architecture to implement the decomposition of legacy systems. A detailed overview of related work on software architectures and coordination languages can be found in [BK96b].

1.4 Organization of the Paper

In Section 2 we discuss a common representation format that is useful to exchange data both between components in a (legacy) system and between tools in a re-engineering environment. In Section 3 we discuss a knowledge retrieval system based on a query algebra which uses the common representation format of Section 2. In Section 4 we discuss a mechanism to decompose a legacy system into smaller components and show how the exchange of data between these components can be established via the common representation format of Section 2. In the final section we put the core technologies for re-engineering in the perspective of forward engineering.

2 Exchanging Data: the Annotated Term Format

How can re-engineering tools share and exchange information? How can components of a legacy system exchange information during incremental renovation?

We propose a data structure, called the *Annotated Term Format (ATF)* specially designed for the data exchange between (possibly) heterogeneous components in a software system. This data format describes terms extended with annotations and is able to accommodate the representation of *all* possible data that might be exchanged between components. The representation is such that individual components are, so to speak, immune for information that is added and manipulated by other components.

ATF can also be used as internal data structure of the components themselves. It is a powerful format in which it is possible to represent, for instance, parse trees that can be annotated with many and diverse sorts of information, such as textual coordinates, access paths, or the result of program analysis (e.g., data flow analysis).

2.1 Annotated Terms

Below we will give an inductive definition of terms in Annotated Term Format, we will discuss the annotation mechanism and we give examples of its use. A full formal specification of ATF is given in [BKOV] and is beyond the scope of this paper.

Definition 1. For each set S, let $S^* = \{s_1 \ldots s_n | s_i \in S, 1 \leq i \leq n\}$ be the set of words over S.
The set $S" = \{"s" | s \in S^*\}$ is called the set of quoted words[4]. The empty quoted word is represented as `""`.
Terms in Annotated Term Format are called *ATerms* and they are defined as follows:

- Let C be a set of constants, then $C^+ \subseteq$ ATerms.
- Let N be a set of numerals, then $N^+ \subseteq$ ATerms.
- Let L be a set of literals, then $L" \subseteq$ ATerms.
- `[]` is an ATerm, called the *empty list*. Let $T_1, \ldots, T_n$ be ATerms, then $[T_1, \ldots, T_n]$ is an ATerm called *list*.
- Let F be a set of function symbols together with their arity. Let $(f, n) \in F$, so f is a function symbol of arity n. Let $T_1, \ldots, T_n$ be ATerms, then $f(T_1, \ldots, T_n)$ is an ATerm called *application*.
- Let T be an ATerm and $T_1, \ldots, T_n$ be (distinct) ATerms with $n \geq 1$, then $T\{T_1, \ldots, T_n\}$ is an ATerm called *annotation*.

By instantiating C, N, L, and F we can obtain various forms of annotated terms.

Example: ATerms Choose $C = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, N = \{\mathtt{1}, \mathtt{2}, \mathtt{3}\}, L = C \cup N$, and $F = \{(\mathtt{f}, 1), (\mathtt{g}, 2), (\mathtt{h}, 3)\}$. Examples of ATerms are then:

- *constants*: `abc`.
- *numerals*: `123`.
- *literals*: `"abc"` or `"123"`.
- *lists*: `[]`, `[1, "abc", 3]`, or `[1, 2, [3, 2], 1]`.
- *functions*: `f("a")`, `g(1,[])`, or `h("1", f("2"), ["a","b"])`.
- *annotations*: `f("a"){g(2,["a","b"])}` or `"1"{[1,2,3],"abc"}`.

[4] For simplicity, we assume in this presentation that $" \notin S$.

2.2 Representing Parse Trees

The use of ATF can be demonstrated by showing how to represent parse trees in ATF, and by annotating them with path and visualization information, respectively. This can be done by instantiating C, N, L, and F. Note that this example will be used in Section 3.2 when we discuss a knowledge retrieval system.

Let C and N be the empty set, and let L be $\{a, \dots, Z, 0, \dots, 9\}$. Let the set of function symbols F consist of the following elements:

- $(\texttt{prod}, 1)$, e.g., $\texttt{prod}(T)$ represents production rule T.
- $(\texttt{appl}, 2)$, e.g., $\texttt{appl}(T_1, T_2)$ represents applying production rule T_1 to the arguments T_2.
- $(\texttt{l}, 1)$, e.g., $\texttt{l}(T)$ represents literal T.
- $(\texttt{sort}, 1)$, e.g., $\texttt{sort}(T)$ represents sort T.
- $(\texttt{lex}, 2)$, e.g., $\texttt{lex}(T_1, T_2)$ represents (lexical) token T_1 of sort T_2.
- $(\texttt{w}, 1)$, e.g., $\texttt{w}(T)$ represents white space T.

With the functions defined above we can represent simple parse trees in ATF. A complete specification of parse trees is given in [BKOV].

The following context-free syntax rules (in SDF [HHKR92]) are necessary to parse the input sentence `true and false`.

```
sort Bool
context-free syntax
  true          -> Bool
  false         -> Bool
  Bool and Bool -> Bool {left}
```

The parse tree below represents the input sentence `true and false` in ATF, in Figure 1 this parse tree is depicted.

```
appl(prod("Bool and Bool -> Bool"),
     [appl(prod("true -> Bool"),[l("true")]),
      w(" "),l("and"),w(" "),
      appl(prod("false -> Bool"),[l("false")])
     ])
```

Note that this parse tree is completely self-contained and does not depend on a separate grammar definition.

To demonstrate the annotation mechanism we extend the above definition to create pointers to trees in the form of paths. A path is a list of natural numbers $[n_0, \dots, n_k]$ where each number n_i stands for the n_ith son of the node n_{i-1} where $1 \leq i \leq k$. The first number n_0 represents the root. The syntax of paths in ATF can be obtained by extending N with $\{0, \dots, 9\}$ and extending the set of functions with $\{(\texttt{path}, 1)\}$.

The example parse tree can be annotated with path information thus:

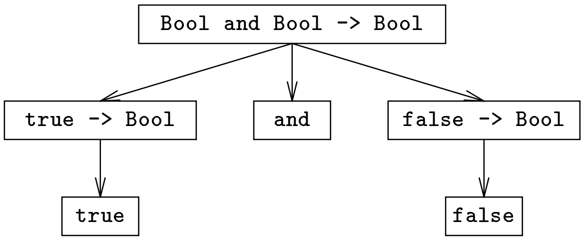

Fig. 1. The syntax tree for `true and false`.

```
appl(prod("Bool and Bool -> Bool"),
     [appl(prod("true -> Bool"),
           [l("true"){path([0,0,0])}]){path([0,0])},
      w(" "),l("and"){path([0,1])},w(" "),
      appl(prod("false -> Bool"),
           [l("false"){path([0,2,0])}]){path([0,2])}
     ]){path([0])}
```

Note that only the application and literal nodes are annotated with path information. The production rules are not annotated since they do not represent nodes in the parse tree for the original input sentence (`true and false`).

Another example is to use the annotations to store visual information, such as colours, for components that support them. We extend the set C with the constants `red`, `yellow`, and `blue` and the set F with the unary function `(colour,1)`. By adding the annotation `colour(red)` a tool for visualization knows that the annotated subterm should be printed in the colour red. Other tools will ignore this colour annotation. As an example we annotate in the example parse tree the node `l("true")`.

```
appl(prod("Bool and Bool -> Bool"),
     [appl(prod("true -> Bool"),
           [l("true"){path([0,0,0]),colour(red)}]){path([0,0])},
      w(" "),l("and"){path([0,1])},w(" "),
      appl(prod("false -> Bool"),
           [l("false"){path([0,2,0])}]){path([0,2])}
     ]){path([0])}
```

In a similar way we can define `font`, `font-type`, and `elision` annotations. Fonts can be obtained by extending the set C with the constants like `times` or `helvetica` and the set F with the unary function `(font,1)`. Font types are defined analogously (`bold` and `italic` are examples of font types). Elision can be obtained by extending C with the constant $\ldots$ (the common notation for elision of text) and the set F with `(elision,1)`. Elision can be used to suppress (large) parts of program texts.

2.3 Discussion

ATF resembles the intermediate tree representations [ASN87, Sno89, Aus90] usually found in compilers and programming environments. It has, however, a number of properties that make it particularly suited for our purpose:

- The term format is straightforward, enabling simple specification, implementation, and use.
- The term format is universal, i.e., all forms of data can be exchanged between components.
- The annotation mechanism permits different components to add their own auxiliary information to data exchanged with other components.
- The term format is language parameterized.

We will see that ATF can be used as a basis for knowledge retrieval (Section 3) and is also suited for exchange of data in heterogeneous, distributed, systems (Section 4.1).

3 Knowledge Retrieval

One of the crucial phases in re-engineering a (legacy) system is to obtain, collect, and store relevant knowledge about it. In this phase it is often desirable to first obtain an overall impression of the system and later on to dive into the details of relevant subsystems. The re-engineer can choose to store relevant (inferred) information to be used later on.

Given a specific re-engineering goal (e.g., year 2000 compliance, or language conversion), a re-engineer will have to zoom in and zoom out on the source code in order to increase her understanding and to identify program fragments that are relevant for the re-engineering goal at hand. Tools supporting such an interactive zooming mechanism are important, given the expectation that not all re-engineering tasks can be automated [Cor89] and will require human interaction and intelligence [You89].

Obtaining knowledge about a program is not language specific. Therefore, it is desirable that tools to obtain this knowledge are generic, i.e., can be parameterized with a desired language. An additional advantage of this approach is that a single tool is sufficient to deal with legacy systems implemented in more than one language, since even within one legacy system usually more than one language is used, viz. the combination of COBOL [ANS85] and Job Control Language JCL [Flo71].

3.1 Query Algebra: Motivation and Minimal Properties

A natural means for obtaining knowledge about a legacy system is to view it as a special type of data base that can be queried. We will use a parse tree with the possibility of adding annotations (as described in Section 2) as our data base model. Thus, the queries are functions that inspect the parse tree

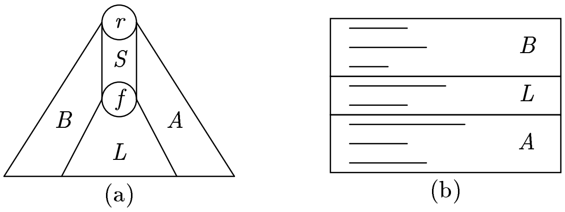

Fig. 2. Regions in the syntax tree and program text.

and modify its annotations when appropriate. Often combinations of extracted information yield important new information, therefore, it is natural to have the possibility of combining queries. This implies the existence of basic queries as well as operations to combine them, in other words an algebra of queries.

A query algebra should have at least the following properties:

- It must be generic (i.e., language parameterized).
- It must allow multidirectional propagation of queries through well-defined parts of the trees.
- It must provide random access to tree nodes.
- It must be adjustable.

A query algebra should be language parameterized since obtaining knowledge is not language specific. It can then also effortlessly deal with polylingual legacy systems. In [PP94b] this is identified as an extremely valuable feature.

Zooming in and out on the code is needed in order to locate the code fragments that cause problems or that are related to a specific re-engineering goal.

Random access on tree nodes can be obtained via the *focus*, an interactive pointing mechanism for the re-engineer to locate (sub)trees. A focus divides a tree in regions, such as:

- The region before the path from the root to the focussed tree.
- The region after the path from the root to the focussed tree.
- The region formed by the nodes on the path from the root to the focussed tree.
- The region formed by the focussed tree.

Arbitrary combinations of these four regions are also possible. A syntax tree with these regions is represented in Figure 2(a), where the B, A, L, and S stand for *before*, *after*, *local*, and *spine*, respectively. The regions before and after are called this way since in the program text they represent the text before and after the focus (or local), see 2(b). The nodes r and f in Figure 2(a) represent the root and the focus in the tree, respectively.

A query algebra system for re-engineering should be adjustable in the following ways:

- New parsers can be connected in order to support queries for the languages found in a specific legacy system.

– The results of semantic analysis (e.g., typechecking, data flow analysis) can be attached to the syntax trees constructed by these parsers and this information can be queried.
– New basic queries and query operators can be added.

3.2 The Query Algebra of IQAT

The Interactive Query Algebra Tool (IQAT) is a prototype we have developed to experiment with query algebras for system re-engineering to see whether the proposed properties are feasible. The query algebra should, therefore, at least support a multidirectional propagation mechanism of queries over tree regions; it should be language parameterized; and it should be adjustable. We will argue below that the IQAT system satisfies these properties.

The multidirectional propagation mechanism is explicitly available via a number of (basic) queries in IQAT. In Figure 2(a) we have seen that the syntax tree can be divided in a number of regions. An examples of a query that uses these regions is `trees(`*Region*`)`; selects all nodes in region *Region*, defined by one of the constants `before`, `after`, `local`, `spine`, discussed in the previous section. Note that this query has the actual tree that is being queried as implicit parameter (see below).

Our query algebra supports a language parameterized mechanism. More precisely, instead of parameterizing it with an entire language definition some queries can be parameterized with elements of a language definition, such as non-terminal names and production rules (we call these in the sequel *Syntactic-Categories*). Each language element in the query is translated to terms in ATF by the query evaluator. The resulting term is used when parse trees of the code of the legacy system are inspected to answer the query. For example, the query `has("MOVE" Corr Id-or-lit "TO" Id-or-lit+ -> Stat)` checks whether the root of a tree is a move-statement in a COBOL program. Another example is `contains(`*Region*`,`*SyntacticCategory*`)`; it selects all nodes in region *Region* which satisfy property *SyntacticCategory*.

The IQAT system supports a number of (basic) data types, for example booleans, integers, sets, and trees. It supports a number of operations on these types, for example the operations `and`, `or`, and `not` on the booleans and the operations `+` and `-` on the integers. We will not further discuss these basic types and their operations, instead, we will focus on the more sophisticated part of the query algebra: queries and operations on them. All queries work implicitly on a set of so-called tree addresses (`SetofTrees`) each consisting of a syntax tree and a path to a node in the syntax tree. Therefore, the first argument of all query definitions is shown between brackets to emphasize that it is an implicit argument of the query. Note that most (but not all) queries take an implicit `SetofTrees` argument and yield a `SetofTrees` as well.

We divide queries in the following conceptual categories and we give some typical examples of each category:

Topographical queries select nodes of given trees.

- `locate`: (`SetofTrees` ×) `Region` × `SyntacticCategory` → `SetofTrees`
  This query selects the nodes with the property *SyntacticCategory* in the region *Region* for all trees in *SetofTrees*.
- `trees`: (`SetofTrees` ×) `Region` → `SetofTrees`
  This query selects all nodes in the region *Region* of all trees in *SetofTrees*.
- `down`,`up`,`left`,`right`: (`SetofTrees` ×) `Int` → `SetofTrees`
  These queries return all nodes that can be reached by going *Int* steps `down`, `up`, `left`, `right`, respectively, in the trees in `SetofTrees`.

Existential queries check whether a node of a syntactic category exists in given syntax trees.
- `exists`: (`SetofTrees` ×) `Region` × `SyntacticCategory` → `Bool`
  This query checks whether one of the trees in *SetofTrees* contains a node with property *SyntacticCategory* and which is located in the region *Region*. In fact, `exists` is very similar to `locate` except that a boolean value is returned instead of a *SetofTrees*.
- `has`: (`Tree` ×) `SyntacticCategory` → `Bool`
  This query checks whether *Tree* has property *SyntacticCategory*.

Quantitative queries gather statistics or metrics about a set of trees.
- `size`: (`SetofTrees`) → `Int`
  This query counts the number of trees in *SetofTrees*.

Attributive queries manipulate the attributes in syntax trees. These queries are different from the previous ones since they may modify the data structure, whereas the others may only extract data (including the data in attributes).
- `add-attribute`: (`SetofTrees` ×) `Attribute` → `SetofTrees`
- `add-value`: (`SetofTrees` ×) `Attribute` × `Value` → `SetofTrees`
- `remove-attribute`: (`SetofTrees` ×) `Attribute` → `SetofTrees`
- `remove-value`: (`SetofTrees` ×) `Attribute` → `SetofTrees`
- `get-value`: (`Tree` ×) `Attribute` → `Value`

User-defined queries are queries to be defined by the user of the IQAT system. For example, a query for calculating the McCabe metric or data flow information.

Next we will discuss a number of composition operations on queries. Note that such operations generally are partial functions. For example, `size` and `exists` can not be sensibly composed. Below we list the most important operators.

Functional composition is used to apply a query to the results of a previous query and has the form
- `Query Query` → `Query`

Consider, e.g., the functional composition `trees(local) size`. Given an initial (implicit) set of trees S, `trees(local)` will compute a new set of trees S' consisting of all nodes of all subtrees of S. Next, `size` will compute the number of subtrees in S'.

Logical composition is used to combine queries which have a boolean as result. They have the syntax:

  - `Query and Query` → `Query`
  - `Query or Query` → `Query`
  - `not Query` → `Query`

Set composition is used to combine queries which have a set of trees as result. They have the syntax:
  - `Query intersection Query` → `Query`
  - `Query union Query` → `Query`

Hierarchical composition is used to impose further restrictions on previous query results. It can only eliminate parts of a previous query result but can not generate new ones. Hierarchical composition has the syntax
  - `Query provided Query` → `Query`

  An expression containing this operator returns all the nodes obtained by evaluating the left-hand side that satisfy the criterion of the right-hand side. For example, the `provided` combinator can be used to locate all `while` statements that contain `if` statements:

```
locate(local,"while" Exp "do" Series "od" -> Stat)
provided
contains(local,"if" Exp "then" Series "else" Series "fi" -> Stat)
```

3.3 Prototype Implementation

A prototype of the query evaluator has been developed using ASF+SDF [DHK96]. It takes a set of trees and applies the specified query on them. Before the actual evaluation starts the query expression is type-checked. The evaluation function has the following form:

- `eval: SetofTrees` × `Query` → `Result`

The output `Result` is either a set of trees, a boolean value, or an integer value. The output can be stored for the evaluation of other queries or for visualization.

Based on these experiences an implementation of the IQAT system is being made using the TOOLBUS [BK96a, BK96b], see Section 4.1. Due to the nature of the IQAT system it is crucial that it can be easily modified. This is achieved as follows:

- The TOOLBUS ensures adjustability of the architecture of the IQAT system, e.g., adding new parsers, adding data flow components, or adding graphical components.
- The ASF+SDF Meta-Environment [Kli93] (an interactive programming environment generator) ensures adjustability of those components of the IQAT system that have been developed with it, e.g., creating dialects of COBOL [ANS85] or adding new functionality to the query algebra.
- The annotated term format (ATF) ensures adjustability of data. Annotated terms are used to represent the code of the legacy system and the annotation mechanism is used both to store and visualize the results of queries.

3.4 Visualization

A visualization mechanism is necessary to show the results of queries. We separated this mechanism from the query algebra, so it can also be used for the viewing of results of other tools, like a data flow analysis tool. Although we recognize the importance of graphical views, we currently restrict ourselves to textual ones.

We have shown in Section 2 that parse trees in annotated term format can be annotated with `path`, `colour`, `font`, `font-type`, and `elision` information. This information can be used both to give a structured view on legacy code and to visualize the result of query evaluation. These evaluations are either booleans or integers, or sets of trees. We will now only discuss the visualization of trees.

A number of access functions can be defined on annotated parse trees to manipulate the various visual annotations. For example, we can manipulate the `colour` annotation with the functions:

- `set-colour`: `Tree` $\times$ `SetofTrees` $\times$ `Colour` $\rightarrow$ `Tree`
- `reset-colour`: `Tree` $\times$ `SetofTrees` $\rightarrow$ `Tree`

The second argument *SetofTrees* is the set of subtrees (tree addresses) of the *Tree* in the first argument that should be textually visualized with a given colour. For `font`, `font-type` and `elision` we have similar access functions.

The annotations `colour`, `font`, and `font-type` can be used to emphasize parts of the legacy code, whereas `elision` is useful to suppress parts of the legacy code.

The set of trees provided by queries, such as `trees`, serves as input for the access functions of the visualization mechanism. The access functions return a parse tree annotated with `colour`, `font`, `font-type`, and `elision` information. A viewing component traverses the resulting tree and interprets the visual annotations by providing the requested colours, fonts, font-types, and elisions.

3.5 Discussion

We have presented a generic approach for querying source code. Although we are currently concentrating on the functionality of the query system itself rather than already applying it in large case studies, we expect the following benefits from this approach:

- During forward engineering, queries can be used to pose *what-if* questions to study the effects of different implementation alternatives.
- During re-engineering queries can be used to explore the structure of a legacy system.

An obvious next step will be to combine queries with program transformations, in order to describe complete conversions of source code.

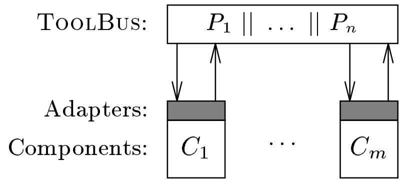

Fig. 3. General architecture of the ToolBus.

4 Structuring and Restructuring

How can a re-engineering tool invoke functionality provided by other tools? How can components of a legacy system call each other's services during incremental renovation?

We will first describe in Section 4.1 an implementation of a software bus, called the ToolBus [BK96a, BK96b], that facilitates the construction of software systems consisting of independent, cooperating, components. In Section 4.2 we focus on the creation and modification of systems by means of the ToolBus. In Section 4.3 we emphasize on using the ToolBus to re-engineer legacy systems, demonstrated by an example: the re-engineering of the Asf+Sdf Meta-Environment[Kli93].

4.1 The ToolBus Coordination Architecture

The ToolBus [BK96a, BK96b] is a component interconnection architecture resembling a hardware communication bus. To control the possible interactions between software components connected to the bus direct inter-component communication is forbidden.

The ToolBus serves the purpose of defining the cooperation of a number of *components* C_i $(i = 1, \ldots, m)$ that are to be combined into a complete system as is shown in Figure 3. The internal behaviour or implementation of each component is irrelevant: they may be implemented in different programming languages or be generated from specifications. Components may, or may not, maintain their own internal state. The *parallel process* $P_1 || \ldots || P_n$ in Figure 3 describes the initial behaviour of the interaction of the components $C_1, \ldots, C_m$ and the interaction between the *sequential processes* $P_1, \ldots, P_n$. Where a sequential process P_i can, for example, describe communication between components, communication between processes, creation of new components, or creation of new processes.

To give an idea of the expressive power of the sequential processes P_s we give a simplified BNF definition:

$$P_s ::= \texttt{send} \mid \texttt{receive} \mid \texttt{create} \mid \delta \mid P_s + P_s \mid P_s \cdot P_s \mid P_s * P_s$$

We only give the main operators and we have abstracted from the data part of the atomic actions `send`, `receive`, and `create`. The operators $+$, $\cdot$, and $*$ stand for choice, sequential composition, and iteration, respectively. The constant process δ stands for the deadlocked process, `send` and `receive` are actions for communication, and `create` is an action for the creation of processes and components. A ToolBus process consists of parallel sequential processes: $P_1 \mid\mid \ldots \mid\mid P_n$, where $\mid\mid$ stands for parallel composition. The operators in ToolBus processes stem from the algebra of communicating processes (ACP) [BW90, BV95]. In fact, ToolBus processes are expressions over a process algebra that also supports relative and absolute time aspects [BB96]. A detailed discussion of ToolBus processes is beyond the scope of this paper, and can be found in [BK96a].

The *adapters* in Figure 3 are small pieces of software needed for each component to adapt it to the common data representation (ATF, see Section 2) and the ToolBus process expression. A number of adapters are available via ToolBus libraries, but for specific components they have to be implemented separately.

4.2 Creating and Modifying New Systems

The ToolBus provides an architecture to compose systems from (basic) components. Within this architecture there is a mechanism to add and replace components without affecting the rest of the system. The components use a common data format (ATF) to exchange information. The common data format can also be used as internal data structure by the components themselves.

Systems developed with the ToolBus can more easily be modified than classical, monolithic, ones. Reasons for modifying an existing system are deleting obsolete components, replacing components by better ones, extending the system with new functionality, and system maintenance. Modifications of a ToolBus-based system can take place at various levels:

- Modifications on the level of the ToolBus. Examples are creation of a system and alterations of the system by adding or removing components.
- Modifications of the components themselves. Examples are optimizing a component or adding more functionality. Such modifications do not affect the shared data structure of components.
- Modifications of the data structure. An example is adding annotations to parse trees to store position information calculated by a parser, such information will be used by an editor and ignored by a compiler.

In general, the effects of modifications are better localized than in a monolithic system.

4.3 Re-engineering Old Systems

If we apply the approach just described in a top-down manner it can also be used to decompose an existing system into (smaller) components. By analyzing a monolithic system one can identify its logic components and their relationships, these are the arrows in Figure 4(a). Having identified these logic components and relationships it may be possible to decompose such a monolithic system. If the

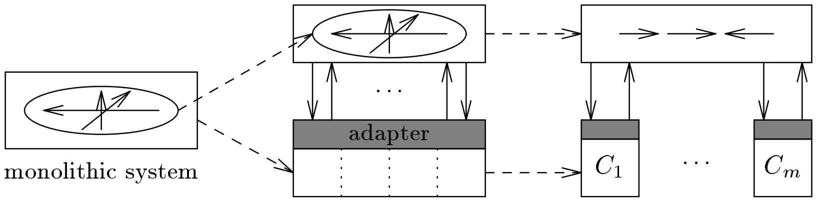

Fig. 4. General organization of restructering.

decomposition is feasible, we can implement these relationships in the software bus, see Figure 4(b) where the arrows in the monolithic system have been moved to the software bus. The monolithic system is connected to the software bus via an adapter that redirects the internal communication to the software bus, we call this *communication extraction*. Finally we can decompose the monolithic system by physically decomposing the logic components and connect them to the software bus, see Figure 4(c), we call this *component restructuring*. Furthermore, it is possible to reorganize the communication in the software bus, we call this *communication restructuring*, this is depicted by the reshuffling of the arrows in Figure 4(c). After having identified these components it has to be decided whether the functionality of the components is obsolete or re-usable. Obviously components with obsolete functionality can be thrown away. For components with re-usable functionality it must be decided whether it is possible to re-use existing code and adapt it, to reimplement the entire component, or to use existing third party software. We can now use the techniques described in Section 4.2 to renovate the system in an incremental way. A similar migration strategy for legacy systems is elaborately discussed in [BS95].

There is not necessarily a one-to-one correspondence between the processes and the components. The possibility of controlling a complex component by more than one sequential process frees us from the need to physically split such a component. This is an extremely valuable feature in the communication extraction phase of the decomposition of a legacy system. Namely, communication between the logic components in the legacy system is transferred to communication between the sequential processes in ToolBus. It is also possible to have more physical components than sequential processes, e.g., components that operate sequentially can be described by a single sequential process this is a valuable feature in the communication restructuring phase of the decomposition of a legacy system. Even if $n = m$ there is not necessarily a one-to-one correspondence between processes and components.

Example The above strategy is used at CWI/UvA to re-engineer the Asf+Sdf Meta-Environment, an interactive programming environment generator developed in the mid eighties.

The re-engineering goals of the Asf+Sdf Meta-Environment were manifold;

we will enumerate them below:

- Language conversion, from LeLisp [LeL87] to C [KR78].
- Improving maintainability by modularization.
- Using third party software, like Tcl/Tk [Ous94].
- Preparing it for adding external tools in the future without redesign.

Originally this monolithic system was implemented in LeLisp [LeL87], a Lisp dialect, and consisted of 200.000 lines of LeLisp code. No code of the old system could be re-used because of the first reverse engineering goal. Also automatic translation of old code was not feasible because of the costs involved and the desired improvements stated above as the second goal. Automatic conversion would also result in another monolithic and difficult to maintain system. After having identified the main components, it was decided that none of them were obsolete. It was decided which ones could be replaced by third party software and which ones had to be reimplemented.

A first attempt to restructure the ASF+SDF Meta-Environment in order to re-engineer it is described in [KB93]. The main goal of this restructuring was to use third party software for the user interface and the text editor. Those components were successfully separated from the main system and every part worked stand-alone. However, the interaction between those parts failed, since the interaction led to unexpected deadlocks. These problems were studied and described in [VVW94] and gave rise to the development of a more sophisticated coordination archictecture, the ToolBus [BK96a, BK96b].

At the time of writing this paper the re-engineering process is still in progress. The relationships between the components are implemented in the ToolBus. All components are being reimplemented in parallel by various people.

5 Perspective: Software Development using Re-engineering Technology

Interestingly enough, re-engineering techniques can play a useful role during forward engineering as well. Combined with the observation that most systems undergo many forward as well as reverse engineering activities during their life time we argue that a discipline of software engineering should be developed in which both forward and reverse engineering are seemlessly integrated. This approach will improve both the cost effectiveness of the software production and the quality of the resulting software.

One can distinguish the following phases in the life cycle of a software system:

Creation: in this phase the system is created using classical engineering, so only forward techniques are used. In Figure 5 this is depicted by a horizontal line: since $r = 0$ we have that $f - r/f + r = 1$, where r and f are a measure for the forward and reverse engineering effort, respectively.

Maintenance: in this phase initially minor maintenance is necessary to keep the system running. In Figure 5 this is depicted by small forward engineering

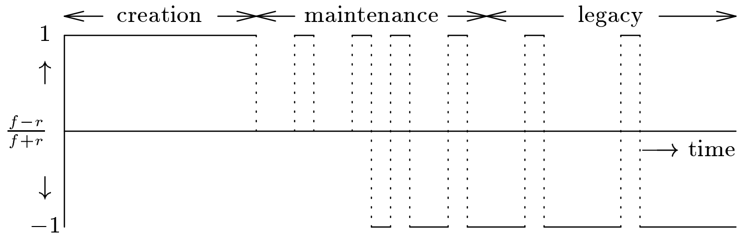

Fig. 5. Classical life cycle of a software system.

lines. However, later on it is becoming more difficult to maintain the system. Reverse engineering techniques are then needed. This is visualized by short forward engineering periods, interrupted by reverse engineering periods that are gradually taking longer time.

Legacy: in this phase it is hardly possible to maintain the system since maintainers are merely busy trying to understand the system.

The above approach yields a legacy system that can only be re-engineered at high costs. In order to avoid this we propose an alternative scenario that reduces the maintenance effort in the long run and prevents the creation of a legacy system. We distinguish the following two phases:

Creation: in this phase reverse engineering techniques are immediately used to influence the design and development of the system: for instance to study the impact of different implementation alternatives. This approach could also be characterized as reverse engineering driven software development. This is depicted in the curve that is presented in Figure 6. The harmonic character of this curve represents the interplay of forward and reverse engineering.

Maintenance: in this phase maintenance is necessary to keep the system running. We see a harmonic curve as well, reflecting the reverse and forward engineering activities while maintaining the system and keeping the knowledge about the system up-to-date.

The above approach could be called *harmonic* software engineering.

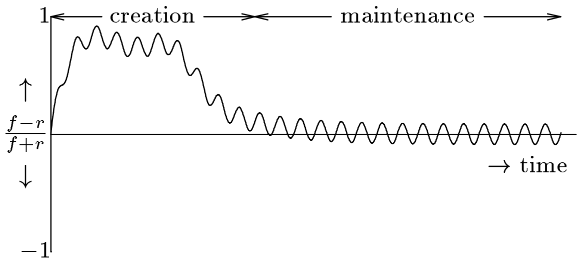

Fig. 6. Desired life cycle of a software system.

The core technologies for system renovation that we have briefly reviewed in this paper may thus also contribute to the discipline of software engineering as a whole.

Acknowledgements

We thank Pieter Olivier for useful comments related to Section 4.1. In addition, we thank all our colleagues in the Resolver project for general discussions on the topic of system renovation.

References

[ANS85] *COBOL, ANSI X3.23*. American National Standards Institute, 1985.

[ASN87] *Specification of Abstract Syntax Notation One (ASN-1)*. 1987. ISO 8824.

[Aus90] D. Austry. The VTP project: modular abstract syntax specification. Rapports de Recherche 1219, INRIA, Sophia-Antipolis, 1990.

[BB96] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8(2):188–208, 1996.

[BK96a] J. A. Bergstra and P. Klint. The discrete time toolbus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 288–305. Springer-Verlag, 1996.

[BK96b] J.A. Bergstra and P. Klint. The toolbus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 75–88. Springer-Verlag, 1996.

[BKOV] M.G.J. van den Brand, P. Klint, P. Olivier, and E. Visser. Aterms: representing structured data for exchange between heterogeneous tools. Unpublished manuscript.

[BKV96] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation – an annotated bibliography. Technical Report P9603, University of Amsterdam, Programming Research Group, 1996.

[BS95] M.L. Brodie and M. Stonebraker. *Migrating Legacy Systems — Gateways, Interfaces & The Incremental Approach*. Morgan Kaufmann Publishers, Inc., 1995.

[BV95] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume IV, Syntactical Methods*, pages 149–268. Oxford University Press, 1995.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[CC90] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[Cor89] T.A. Corbi. Program understanding: challenge for the 1990s. *IBM System Journal*, 28(2):294–306, 1989.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[DRW96] P.T. Devanbu, D.R. Rosenblum, and A.L. Wolf. Generating testing and analysis tools. *ACM Transactions on Software Engineering and Methodology*, 5(1):42–62, 1996.

[Flo71] I. Flores. *Job Control Language and File Definition*. Prentice-Hall, Englewood Cliffs, N.J., 1971.

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual*, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989.

[KB93] J.W.C. Koorn and H.C.N. Bakker. Building an editor from existing components: an exercise in software re-use. Report P9312, Programming Research Group, University of Amsterdam, 1993.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[LeL87] INRIA, Rocquencourt. *LeLisp, Version 15.21, le manuel de référence*, 1987.

[Ode93] M. Odersky. Defining context-dependent syntax without using contexts. *ACM Transactions on Programming Languages and Systems*, 15(3):535–562, 1993.

[Ous94] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[PP94a] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[PP94b] S. Paul and A. Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.

[REQ94] *REQL Source Code Query Language*. Raleigh, USA, 1994. User Guide and Language Reference — Version 2.0 for Windows.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[Sno89] R. Snodgrass. *The Interface Description Language*. Computer Science Press, 1989.

[VVW94] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. Control and data transfer in the distributed editor of the ASF+SDF meta-environment. Report P9415, Programming Research Group, University of Amsterdam, 1994.

[You89] E. Yourdon. RE-3. *American Programmer*, 2(4):3–10, 1989.