Computer Supported Cooperative Software Engineering with Beyond-Sniff

Walter R. Bischofberger, Thomas Kofler,
Kai-Uwe Mätzel, Bruno Schäffer
UBILAB, Union Bank of Switzerland
Bahnhofstr. 45, CH-8021 Zurich
e-mail: {bischofberger, kofler, maetzel, schaeffer}@ubilab.ubs.ch

Abstract

Teamwork is a prerequisite for the development of large complex software systems. In conventional software engineering coordination of teamwork is achieved by exchanging formal documents and by providing support for keeping these documents consistent even while several developers are evolving them.

In order to support teamwork more effectively it is important to move the focus beyond coordination towards cooperation. The goal of the Beyond-Sniff project is to provide cooperation support in three ways. First, by providing a terminological and conceptional foundation for the field of cooperative software engineering. Second, by developing tools for computer supported cooperative software engineering. Third, by developing a platform for control and data integration, a technical prerequisite for computer supported cooperative software engineering.[1]

1 Motivation and Overview

Software systems tend to have a more and more complex and comprehensive view on the application domains they model. The complexity of these systems goes hand in hand with the difficulties (the amount of efforts to be invested for their development) arising during their development. This is an important reason why the implementation of even modest software systems can only be carried out effectively as an iterative and cooperative process.

Communication and coordination are prerequisites to successful cooperation. Their quality strongly affects team productivity and the resulting products. Unfortunately, the costs of satisfying communication and coordination needs quickly reach unacceptable dimensions. These costs naturally limit the size of closely cooperating teams and therefore, also the size and complexity of projects that can be carried out cooperatively.

Since we have (or want) to stretch the limits created by the negative side effects of cooperation, we need methods and tools that explicitly address communication and coordination. The provision of suitable methods and tools and their application are the core of an emerging discipline that is called cooperative software engineering. A sensible and promising approach to cooperative software engineering is the enrichment of conventional software engineering by concepts found in the field of computer supported cooperative work (CSCW).

1.1 Motivation for the Beyond-Sniff Project

Based on our experience in developing large object-oriented software systems we began in 1991 to develop Sniff, an innovative C++ programming environment [1][2]. We drew the motivation for this project from the facts that the characteristics of object-oriented software development strongly increases the requirements for development environments [3] and that, at that time, our personal needs were not fulfilled by any available environment.

Sniff proved to be very useful for browsing, editing and executing large object-oriented software systems. The longer we use it in our team, the more it becomes clear that its conventional way of supporting cooperation with a configuration management system is far from optimal. Configuration management helps to organize the development process of large projects but it addresses only few problems of cooperation. For instance, it ignores the exchange of arbitrary information related to projects. In other words: there are no facilities for computer supported know-how transfer or fine grained harmonizing of activities.

A useful cooperative development environment should provide such support. Therefore it has to comprise facilities for flexible communication between coworkers and for attaching arbitrary information to all kinds of project artifacts. Typical information to be exchanged or attached are rationales for structure and evolution of subsystem interfaces as well as myriads of detailed design decisions.

Neither current software engineering nor development environments and their underlying architectures are suited to provide comprehensive cooperation support. This insight provided us with the motivation to start the Beyond-Sniff project.

The goals of the Beyond-Sniff project are to develop a conceptual framework for cooperative software engineering, and to build the development environment needed for its enactment. This environment requires a platform that allows to integrate a set of existing and new tools into a cooperative software development environment.

---

1 Published in Procs. of the 7th Conference on Software Engineering Environments, Noorwijkerhout, Netherlands, 5-7 April 1995.

2 Sniff was commercialized under the name of SNiFF+ at the end of 1992. The product version is free for universities and can be downloaded by ftp from eunet.co.at (/pub/vendor/takefive) or from self.stanford.edu (/pub/sniff).

1.2 Contents

Section 2 outlines two forms of cooperation we identified and defines the term Cooperative Software Engineering (CSE). Based on these definitions it investigates currently employed approaches to cooperative software engineering. Section 3 describes the Beyond-Sniff approach to support CSE. Section 4 provides some background on the realization of Beyond-Sniff and outlines the relationship between our approach and related work. Section 5 draws conclusions.

2 Cooperative Software Engineering

The terms "cooperative software engineering" and "computer supported cooperative software engineering" are used in different contexts with different meaning. There is no generally accepted definition for them. For this reason this section defines the term Cooperative Software Engineering (CSE) and discusses state-of-the-art approaches to support it.

2.1 Forms of Cooperation

We assume an intuitive understanding of the term cooperation. Cooperation usually implies shared goals among different actors [17]. Coordination is managing dependencies between activities. Coordination is an important part of cooperation.

We identify two forms of cooperation, policy-driven and informal cooperation. *Policy-driven cooperation* is done by exchange and correct handling of well-structured documents and concurrency control regarding the access to artifacts. *Informal cooperation* is characterized by the unrestricted exchange of structured or unstructured information.

Policy-driven cooperation comprises all coordination activities specifically organized for a software development process.

- It is regimented by a formal agreement about how a project has to be carried out. For small projects such an agreement can be relatively simple. For large projects or families of projects it is usually described in a project management handbook.
- The cooperation model defined in the agreement is implemented by a well-defined exchange of well-structured documents.
- It is supported by tools for controlling the evolution and management of these documents, such as configuration management tools, process engines, and work flow applications.

Informal cooperation comprises all coordination activities that were not formally planned.

- It is inherently not regimented.
- It is implemented by spontaneous, flexible, pragmatic exchange of information between project members.
- It is supported by everything that eases communication. Besides nearness in time and space these are mechanisms and devices such as e-mail, video conferencing, textual annotations of artifacts, and tools for extracting information from source code.

No matter how a software process is organized and regimented it always comprises policy driven and informal cooperation activities. The challenge of project management is to work toward an optimal relation between them, depending on the specific characteristics of a project.

2.2 The Nature of Cooperative Software Engineering

Based on the term software engineering, as defined by Pomberger [20] and on the terms introduced above we can define Cooperative Software Engineering (CSE) as follows:

Cooperative software engineering comprises all software engineering methods, norms and tools that support teamwork flexibly and effectively.

Cooperative software engineering is therefore a subset of classical software engineering. Tools support CSE if they fulfill the following requirements: They provide mechanisms for policy-driven and informal cooperation as well as for access control, and the mechanisms are completely integrated into the tools.

The following examples illustrate this definition. A syntax driven editor supporting distributed simultaneous editing is no CSE tool because it does not support policy-driven cooperation. A documentation tool comprising browsers for information acquisition, supporting simultaneous editing and exchange of documents between team members (if the privileges permit it), is a CSE tool.

2.3 Approaches Related to Cooperative Software Engineering

There is a variety of approaches supporting cooperative work in general or cooperative software engineering in particular. This subsection discusses how conventional software engineering, process-centered software engineering, and CSCW address cooperation, and why neither approach fully meets the definition of CSE.

Conventional Software Engineering. For conventional software engineering coordination of concurrent development is one of the major task of configuration management. Pessimistic and optimistic approaches to coordination are distinguished today [24].

Pessimistic coordination means that all developers work on the same artifacts. The concurrent editing of the same files is prevented by locking. In practice, this approach fails as soon as the number of coworkers exceeds a project-dependent but very low boundary.

During optimistic coordination each developer works on his personal copy of the source code. From time to time, the copies can be merged into a new shared version. Conflicts between changes have to be resolved during the merge process. The advantage of optimistic coordination is that it makes it possible to decouple developers almost completely for some time. The price for decoupling is the need for merging. At the merging phase, part of the communication that has been postponed now takes place in a more concentrated manner. Decoupling pays off because the cost of postponing and merging are usually much smaller than the benefit obtained.

Pessimistic and optimistic coordination are supported by configuration management tools. Both are forms of policy-driven cooperation. Informal cooperation is not addressed–on purpose–by conventional software engineering, which is a consequence of the basic assumption that software development can be carried out top-down as a tailoristic, rigidly sequential process. Although this basic assumption is today generally considered wrong (e.g., [2, 4]), informal cooperation is still not addressed by software engineering.

Process-Centered Software Engineering. Process-centered software engineering tries to establish a comprehensive theoretical basis for understanding, describing, and enacting specific software processes [15, 19].

The basic idea is to describe a specific software process with all the activities and information flows it comprises. The resulting process model is represented as a set of rules that define in which sequence under which preconditions which documents may be modified with which tools. With the same mechanism invariants for the usage of tools are defined [14]. A process model is enacted by executing it with a process engine, which is the hub of every process-centered development environment. The process engine controls the application of all tools. Tool integration hence is a prerequisite for practical process-centered software engineering.

Process-centered software engineering is still in its infancy but receives great attention by researchers. This is manifested by the International Software Process Workshop and International Conference on the Software Process series. We believe that it will take considerable research efforts before process-centered software engineering comes into widespread use.

Process-centered software engineering is a consequent evolution of conventional software engineering approaches. Its purpose is to optimize policy-driven cooperation without addressing informal cooperation.

Computer Supported Cooperative Work (CSCW). The goal of CSCW [26, 12] is to assist groups in communicating, in collaborating, and in coordinating their activities. Ellis et al. propose a time and space taxonomy for CSCW tools [26] as depicted in figure 1. Meeting room technology would be within the upper left cell; a real-time document editor within the lower left cell; a bulletin board within the upper right cell; and an electronic mail system within the lower right cell.

Tools located in any cell of this taxonomy can be useful for CSE. This is true for all general purpose CSCW tools, although they do not fulfill our definition of CSE. Due to the relative immaturity of the field of CSCW there are not many systems in actual use for software engineering besides e-mail and bulletin boards.

There are also a few tools that were specifically developed for CSE. They mostly provide support in synchronous editing and debugging (e.g., [7, 13]). We would like to have this kind of tools available in daily work (as well as many general purpose CSCW tools) but we do not think that they address the most important problems developers face today in cooperatively developing large software systems.

	Same Time	**Different Times**
Same Place	face-to-face interaction	asynchronous interaction
Different Places	synchronous distributed interaction	asynchronous distributed interaction

Figure 1. Time space taxonomy according to [26].

There are two areas where we foresee considerable benefit in applying CSCW to software engineering. One is the support of synchronous activities during the analysis and design phases (e.g., [18]). The other is support of asynchronous informal cooperation. Practical tool support for both areas is missing today because specific tools for CSE can only be built on top of a comprehensive infrastructure that allows to integrate sets of tools used by more than one developer. Research on these problems is carried out in the field of tool integration [24].

3 CSE with Beyond-Sniff

Considerable research for methods and tools supporting cooperation is currently being carried out in the areas of process-centered software engineering and CSCW. We believe that results from both areas should be practically applied as soon as they are available. Unfortunately, the support of informal cooperation on software development is neglected in both software engineering and CSCW.

It is therefore important to do research in the area where CSCW and CSE intersect. We have taken first steps in this direction in developing Beyond-Sniff. This section describes the innovative aspects of Beyond-Sniff that support cooperation.

3.1 Informal Cooperation with Beyond-Sniff

Cooperative software development requires a lot of communication between developers. The use of object technology, which becomes increasingly popular, even tends to exacerbate the communication problems among developers. They often neglect to share small pieces of information anyway, because the conventional way of putting them into documents with fixed structure does either not make sense or is too expensive. The worst, but typical case is information that can not be formalized, such as ideas, short term plans, or a set of remarks about a class and its methods. It is difficult and tedious to organize this kind of information in conventional documents, to keep it up to date, and to make use of it.

It is therefore common that a developer is interrupted by requests for some specific information that nobody else can provide. Interrupts of that kind affect the concentration and are counter productive if they frequently occur. Facilities for asynchronous communication can decrease the number of productivity decreasing interruptions.

Developers working in the same building can satisfy their need for information to a certain degree by informally keeping each other up-to-date. This is no more feasible for teams that have many members, or that are separated by large distances. The communication problems then lead to a permanent information deficit or to a huge overhead, which both reduce overall productivity.

We experienced these problems first hand when Sniff was commercialized and certain parts were finished in Zurich while work was already going on in Salzburg. This was a strong motivation to develop an annotation mechanism as part of the Beyond-Sniff platform. This annotation mechanism makes it possible to connect structured information with any kind of artifacts, be it fine-grained artifacts such as classes and instance variables or coarse-grained artifacts such as projects and files.

A Beyond-Sniff annotation has a type that defines which information fields it comprises. This makes it possible to store different kinds of structured information. Frequently used annotation types are, for example, error, documentation, idea, and to-do annotations. Annotated artifacts are visually marked in all Beyond-Sniff tools. One mouse click suffices to display all annotations connected with an artifact. Annotation types can be extended by inheritance, and they are defined with a graphical schema editor.

Annotations are centrally stored for every project per site. A developer has either the possibility to access an annotation via artifacts, or he can formulate an OQL [6] query with a query tool to obtain all annotations matching certain conditions. For example, it is possible to obtain all *idea* or *to-do* annotations that have been connected to a certain project since a given date. Figure 2 shows the query tool with an evaluated query. Figure 3 shows the screen after selection of a matching annotation: The user sees that the class SymtabItem has annotations. One of them is shown in a separate window.

In many situations, a developer wants to be notified when an annotation is created. For that purpose, he can define OQL trigger queries that are executed whenever an annotation is created or modified. Upon a match the developer is notified either with the Beyond-Sniff notification tool or by e-mail. A developer might wish, for example, to immediately see all error annotations attached to the projects he is responsible for.

The central storage of annotations together with the query and trigger query mechanisms makes it possible to easily share information. For instance, there is no need to bother about who might be interested in some information. This reduces the communication overhead by decoupling developers the same way as the Smalltalk change propagation mechanism decouples cooperating objects [10].

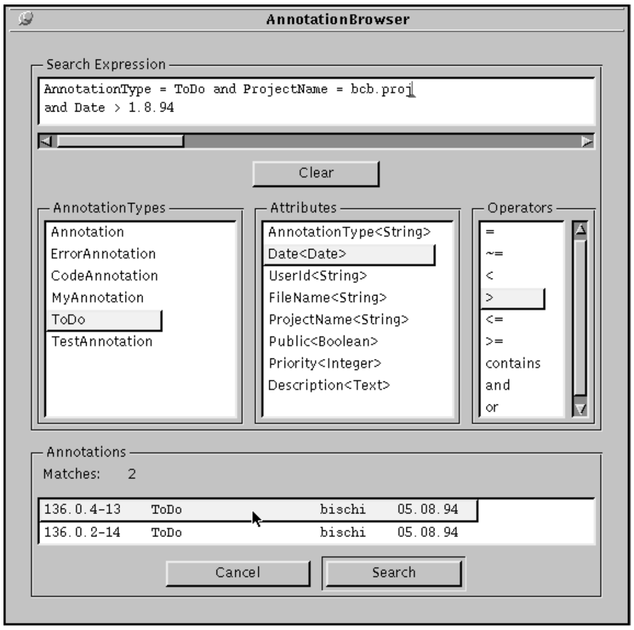

Figure 2. AnnotationBrowser.

Annotations are a mechanism for undirected communication. Sometimes it is useful to make sure that coworkers read a particular annotation. Beyond-Sniff has two features for that purpose. First, an annotation can be specifically addressed to developers. Second, the creator can specify that he wants to be automatically notified whenever an annotation is opened.

Annotations are not only created manually but also generated by tools. Annotations are generated, for example, on check-in of a file into the version control system, or when a project's structure is modified.

Beyond-Sniff's annotations are a hybrid approach to information management. On one hand, they can be used together with links to organize information as a hypertext. On the other hand they have a structure defined by a type, they are centrally stored and they can be retrieved with a query mechanism. This integration of hypertext and database approach makes it possible to easily store structured information, to connect it with any kind of artifact and to find them in different ways.

In this paper annotations and links were discussed in the context of cooperative software development only. Of course, these are very general devices that can solve a variety of information management and communication problems.

3.2 Policy-Driven Cooperation with Beyond-Sniff

Beyond-Sniff implements a conventional configuration management approach, i.e., optimistic and pessimistic coordination. There are two areas in which Beyond-Sniff goes beyond conventional configuration management. It supports optimistic coordination over long distances and low bandwidths, and it provides extensive support for merging parallel versions of entire projects. A detailed discussion of the first topic goes beyond the scope of this paper. The rest of this section gives a brief overview on Beyond-Sniff's merging support.

Projects define the level of granularity on which developers are cooperating with Beyond-Sniff. A project consists of all artifacts relevant to the development of a software system. Projects have explicit representations and can be structured in a tree of subprojects.

The merging of projects is a central task whenever optimistic cooperation is employed. Beyond-Sniff's TurboMixer provides support for comparing and merging projects on a high level of abstraction. It visualizes differences with colors and pictograms on project, symbolic and textual levels and there are several abstraction levels on which both symbolic and textual differences can be browsed.

Figure 4 shows the TurboMixer in comparing three consecutive versions of a project, ordered by age from left to right. The structure of every working project is visualized as a tree and the classes are listed above them. New, changed, and deleted elements are visualized the same way in the tree and in the list. The semantics of the pictograms and colors is described in the lower left corner.

3.3 Related Work

In the area of CSCW, a large number of synchronous approaches has been published, such as synchronous editing of documents and video conferencing. Except for the latter, none of them has achieved a relevant level of practical application until now. Dewan proposes in [7] to apply synchronous approaches such as synchronous editing and debugging to cooperative software engineering. This approach does not address, however, the relevant problems of cooperative software engineering as discussed in Section 2.

Some programming environments such as Cadillac [9] and Field [22] incorporate annotations, too. In contrast to our approach their annotation concepts are simplistic means for connecting some information with source code. They are tool specific and cover only a small part of the artifacts. Moreover, these environments are aimed at single developers.

We do not know about tools similar to the TurboMixer. Concepts building on the same basic idea are also presented in a paper by Grass [11].

4 Realization of Beyond-Sniff

Beyond-Sniff consists of a set of cooperating tools used by a set of developers running on a set of workstations. The biggest challenge in implementing such a distributed system is tool integration, i.e., the integration of different kinds of tools and services in a way that they work together as seamlessly as possible (from the user's point of view). The field of tool integration can be further divided into control, data, and user interface integration (e.g., [24, 9]). Control and data integration are discussed in this section.

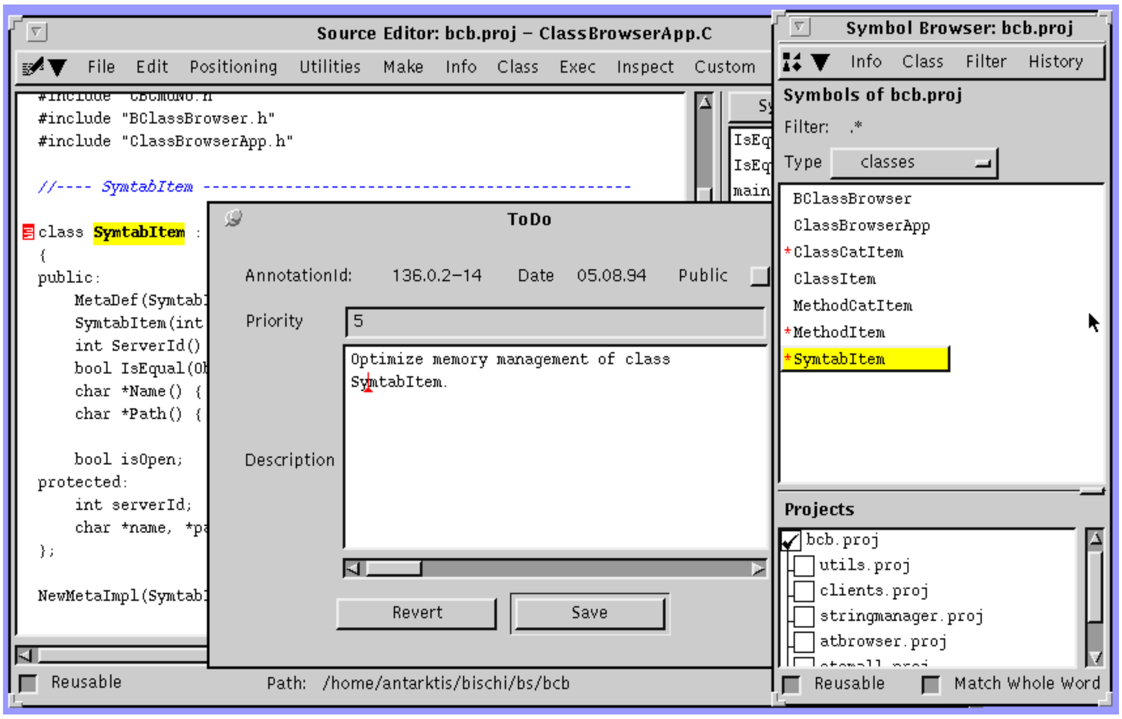

Figure 3. Annotation aware Editor and SymbolBrowser with an annotation.

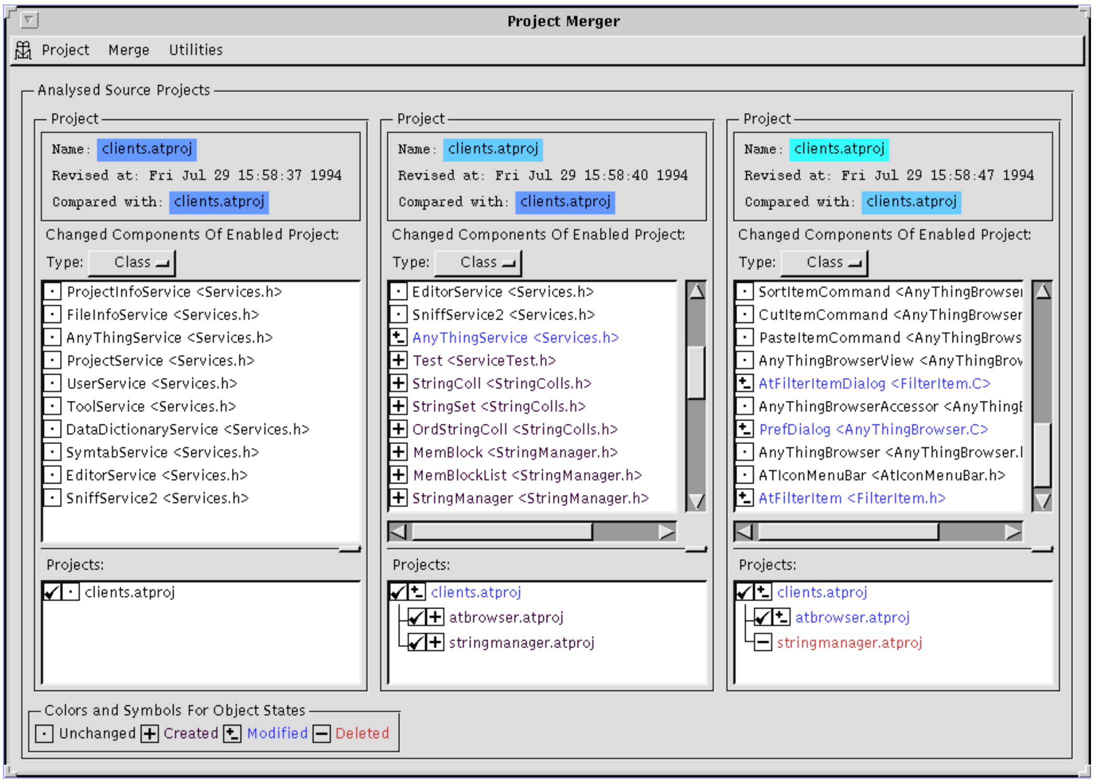

Figure 4. TurboMixer visualizing differences in structure and classes.

4.1 Overview of Services and Applications

Beyond-Sniff consists of an extensible number of services and tools. It is beyond the scope of this paper to give a comprehensive overview of them. Figure 5 provides an architecture overview, which also shows some of the important services and tools.

4.2 Control Integration

Scalability is a key property for the integration mechanism of a platform that runs a large number of services and tools. Beyond-Sniff achieves scalability by using point to point communication for request processing between clients and services, and multicasts between services and their clients for notifying updates. The efficiency of the communication between services and applications is therefore independent of the number of running clients and services.

Beyond-Sniff provides a configurable two-level hierarchy of service brokers to connect clients with services as depicted in Figure 6. The global service broker directly connects clients and global infrastructure services, which exist only once per installation (e.g., the UserService). Otherwise it forwards the request to a second-level service broker responsible for a certain type of service (e.g., ProjectServices). A second-level service broker knows all active services of a certain type. Upon the receipt of a request it checks whether the requested service is already running and the service can support a further client. Otherwise a new service is started and connected.

A further degree of indirection has been implemented to make it possible to replace some kind of services without modifying the clients. In ordering a

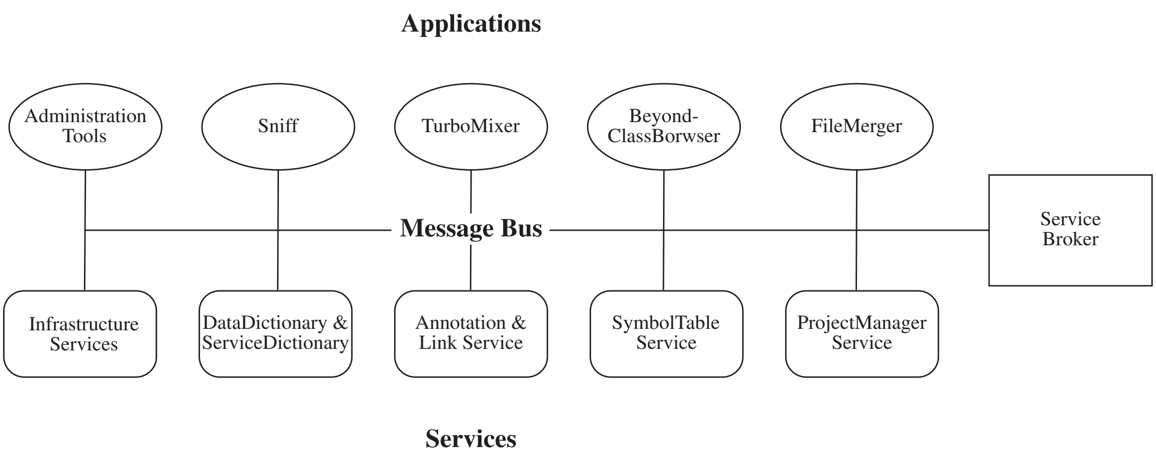

Figure 5. Architecture overview.

service from the global service broker a prospective client does not specify what kind of a service it needs but it states functionality requirements. The global service broker looks up which kind of service provides the requested functionality before negotiating a connection.

Service brokers are implemented based on a specific framework. To implement a new service broker it suffices to specify when a service can support a further client, and how it is started.

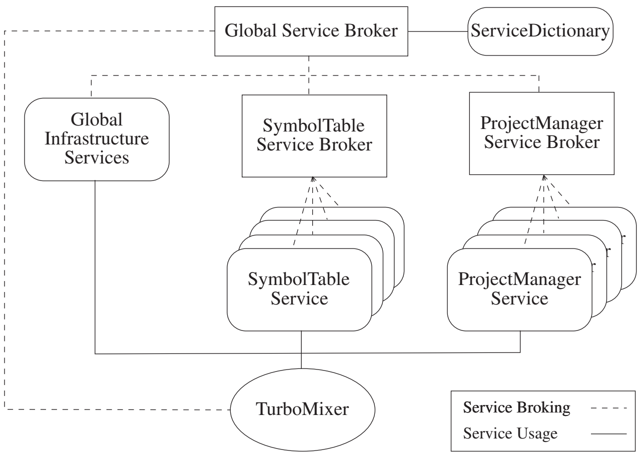

Figure 6. Service broker hierarchy.

4.3 Data Integration

Data integration between services and applications on the Beyond-Sniff platform is based on a federated approach. There is no global data model, and every service is responsible for the consistency of its own information. All services use generic object graphs as their external data representation. The data model of each service is described by an object graph and managed by the central DataDictionary. Beyond-Sniff object graphs are language independent. They can be represented simply and efficiently in different programming languages.

The generic way to obtain information from a service is to send it an OQL query [6]. The service evaluates the query against its database and returns an object graph as the result.

Every service defines it own data model, which can be extended by clients. This is an important requirement for generic services because it is usually impossible to design a complete data model right from the beginning. For example, it is impossible to know in advance all the different kinds of information that tools want to store about project artifacts. If this flexibility is missing replication of functionality and data will result. This is clearly undesirable.

Beyond-Sniff provides a framework which makes it possible to implement new standard services with a minimal effort. The framework comprises, among others, the management of object graphs, the evaluation of OQL queries, locking, recovery, and a standard update interface. To implement an information service using an object graph database it suffices to override the methods for loading and storing the data, and to define the data model.

4.4 Related Work

In comparing different approaches for the integration of applications the relevant differences are usually found in the way that control and data integration are organized.

Message dispatchers (e.g., [21]) are today the most widespread approach to control integration. The best known of them, the HP-Softbench message dispatcher [5], is almost a defacto industry standard. The basic idea underlying the message dispatcher approach is that services and applications communicate by sending strings to each other. These strings are not transmitted over a point-to-point connection but they are sent to the message dispatcher which forwards them to interested processes. Each process interested in a certain kind of messages installs a pattern with the message dispatcher. If a message matches a pattern it is forwarded to the corresponding process.

This approach works well if a small number of tools exchange a small number of messages from a manageable number of types. It does not scale for two reasons. The central message dispatcher becomes a bottle neck, and the lack of explicit protocols makes it difficult to determine the pattern to be installed to receive a certain kind of messages (and only these). Beyond-Sniff does not have scalability problems. Its clients and services communicate over point-to-point connections, and it provides a precise service broker mechanism, as discussed in Section 4.2.

Currently many approaches to data integration are under discussion. Certain research papers propose the construction of a global data model which is mapped to the local data models of the services (e.g., [23]). This approach is conceptually attractive but it is unlikely that it will be applicable on practical problems in the near future. Another approach is to standardize data integration. The most prominent standard is PCTE [25]. The basic idea underling PCTE is to store all relevant information in one database with a unified data model. This is only realistic for a relatively small percentage of the information about a real world software system. For this reason PCTE concentrates on storing information about artifacts. The artifacts themselves are stored in the file system and interpreted by the corresponding tools. Implementations of PCTE such as Emeraude [8] are commercially available and applied for real-world tool integration.

PCTE takes a heavy-weight approach to data integration and ensures data consistency to a large degree. It provides only a part of Beyond-Sniff's functionality. Services providing large amounts of information such as the SymtabService are beyond the scope of PCTE. PCTE and Beyond-Sniff therefore take different approaches to data integration. PCTE is a centralized approach focusing on consistency. Beyond-

Sniff is a federated system with emphasis on lightweightness, extendibility and flexibility.

4.5 State and Further Proceeding

We implemented the first Beyond-Sniff prototype in 1992 to validate the feasibility of our approach. Based on this experience we rewrote large parts of the infrastructure, which is now mature enough that Beyond-Sniff is used on one host by multiple developers cooperatively. It is now used for its own evolution.

The next step is to replace the message bus to make it possible to use Beyond-Sniff cooperatively on several hosts. In parallel, the tools running on Beyond-Sniff will be evolved.

While implementation of the support for cooperation over large distances and low bandwidths has not started yet, we implemented GTS [16], a generic transport-layer-independent group-communication mechanism.[3] GTS guarantees the reliable exchange of information between groups of sites. It is the technical foundation on which we intend to realize the said functionality.

5 Conclusions

Software systems are most often developed in teams. Teamwork implies cooperation and therefore also coordination needs. We identified two forms of coordination, policy-driven and informal coordination. Conventional configuration management addresses the more obvious need for policy-driven coordination. Pessimistic or optimistic approaches can be distinguished. Informal coordination is neither addressed in theory nor supported by tools in practice.

In a rather different area, CSCW investigates the problem of computer supported cooperative work. Combining ideas from both fields leads to a broadened view that we call Cooperative Software Engineering (CSE).

We are currently working on Beyond-Sniff, an environment that focuses on cooperative software engineering. Beyond-Sniff provides a number of tightly integrated services and tools. Sniff, one of the tools, is in widespread use today. The TurboMixer, another example, is a novel approach to ease the burden of comparing and merging large amounts of code. Moreover, Beyond-Sniff provides the necessary infrastructure to integrate services and tools on a large scale. Our work is not finished yet, but we made some important steps towards a future cooperative software engineering environment.

6 References

1 Bischofberger WR: Sniff - A Pragmatic Approach to a C++ Programming Environment. Procs. of the USENIX C++ Conference, Portland, Oregon, Aug. 1992

2 Bischofberger WR, Pomberger G: Prototyping-Oriented Software Development – Concepts and Tools. Springer-Verlag 1992

3 Bischofberger WR, Kofler T, Schäffer B: Object-Oriented Programming Environments: Requirements and Approaches. Software – Concepts and Tools , Vol. 15 No. 2, Springer-Verlag, 1994

4 Booch G: Object-Oriented Analysis and Design with Applications. Benjamin/Cunnings Publishing Company, 1994

5 Cagan MR: The HP Softbench Environment: An Architecture for a New Generation of Software Tools; Hewlett-Packard Journal, Vol. 41, No. 3, 1990

6 Cattell RGG (ed.): The Object Database Standard: ODMG-93; Morgan Kaufman Publishers, 1994

7 Dewan P, Riedl J: Toward Computer Supported Concurrent Software Engineering. IEEE Computer, January 1993

8 Emeraude: Emeraude V12 User Manual; 1991

9 Gabriel R. P. et al.: Foundation for a C++ Programming Environment. Procs. of C++ at Work-90, Secaucus, New Jersey, 1990

10 Goldberg A, Robson D: Smalltalk-80–The Language; Addison-Wesley 1989

11 Grass JE: Cdiff: a Syntax Directed Differencer for C++ Programs. Procs. of the USENIX C++ Conference, Portland, Oregon, Aug. 1992

12 Grudin J: CSCW: History and Focus; IEEE Computer, Vol. 27, No. 5, May 1994

13 Kaiser GE, Kaplan SM, Micaleff J: Multiuser, Distributed, Language-Based Environments. IEEE Software, Vol. 4, No. 6, Nov. 1987

14 Kaiser GE, Popovich SS, Ben-Shaul IZ: A Bi-Level Language for Software Process Modeling. Procs. of the 15th ICSE, 1993

15 Madhavji NH: The Process Cycle. Software Engineering Journal, September 1991

16 Maffeis S, Bischofberger WR, Maetzel KU: GTS: A Generic Multicast Transport Service to Support Disconnected Operation. Procs. of the 2nd USENIX Symposium on Mobile and Location-Independent Computing, Ann Arbor, MI, 1995

17 MaloneTW, Crowston K: The Interdisciplinary Study of Coordination. ACM Computing Surveys, Vol. 26, No. 1, March 1994

18 Olson GM, Olson JS: User-Centered Design of Collaboration Technology. Journal of Organizational Computing, Vol. 1, No. 1., 1991

19 Osterweil L: Software Processes are Software too. Procs. of the 9th ICSE, 19987

20 Pomberger G, Blaschek G: Software Engineering – Prototyping and Object Oriented Software Development. Carl Hanser Verlag, 1993

21 Reiss SP: Connection Tools Using Message Passing in the Filed Environment; IEEE Software, July 1990

22 Reiss SP: Interacting with the FIELD environment. Software – Practice and Experience, Vol. 20, S1, 1990

23 Sarkar M, Reiss SP: A Data Model for Object-Oriented Databases; Technical Report CS-92-56, Department of Computer Science, Brown University, 1992

24 Schefstöm D., van den Broek G.: Tool Integration–Environments and Frameworks; John Wiley & Sons, 1993

25 Wakeman L, Jowett J: PCTE–The Standard for Open Repositories. Prentice Hall, 1993

26 Ellis CA, Gibbs S, Rein GL: Groupware– Some Issues and Experiences. CACM, Vol. 34, No. 1, January 1991

[3] GTS was developed in a cooperation between University of Zurich, UBS/UBILAB, and Siemens Munich. The project was sponsored by the Swiss Federal Commission for the Advancement of Scientific Research (KWF).