BALL: Biochemical Algorithms Library

N.P. Boghossian, O. Kohlbacher,
H.-P. Lenhof

MPI–I–1999–1-002 January 1999

Author's Address

N.P. Boghossian, O. Kohlbacher, H.-P. Lenhof
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken
{nicom|oliver|len}@mpi-sb.mpg.de
http://www.mpi-sb.mpg.de/BALL

Acknowledgements

This work was is supported by the DFG research cluster "Informatikmethoden zur Analyse und Interpretation großer genomischer Datenmengen", grant LE952/2-1.

Abstract

In the next century, virtual laboratories will play a key role in biotechnology. Computer experiments will not only replace time-consuming and expensive real-world experiments, but they will also provide insights that cannot be obtained using "wet" experiments. The field that deals with the modeling of atoms, molecules, and their reactions is called Molecular Modeling. The advent of Life Sciences gave rise to numerous new developments in this area. However, the implementation of new simulation tools is extremely time-consuming. This is mainly due to the large amount of supporting code (*e.g.* for data import/export, visualization, and so on) that is required in addition to the code necessary to implement the new idea. The only way to reduce the development time is to reuse reliable code, preferably using object-oriented approaches. We have designed and implemented BALL, the first object-oriented application framework for rapid prototyping in Molecular Modeling. By the use of the composite design pattern and polymorphism we were able to model the multitude of complex biochemical concepts in a well-structured and comprehensible class hierarchy, the BALL kernel classes. The isomorphism between the biochemical structures and the kernel classes leads to an intuitive interface. Since BALL was designed for rapid software prototyping, ease of use and flexibility were our principal design goals. Besides the kernel classes, BALL provides fundamental components for import/export of data in various file formats, Molecular Mechanics simulations, three-dimensional visualization, and more complex ones like a numerical solver for the Poisson-Boltzmann equation. The usefulness of BALL was shown by the implementation of an algorithm that checks proteins for similarity. Instead of the five months that an earlier implementation took, we were able to implement it within a day using BALL.

Contents

1 Introduction

The fast development of new methods in Molecular Modeling (MM) becomes increasingly important as the field of Life Sciences evolves more and more rapidly. According to our experience, a researcher spends only a small fraction of his or her time on the creative process of developing a new idea, whereas the realization of the idea – the implementation and verification – is usually very time consuming. For this reason, many good ideas have not been implemented and tested yet. Although the need for shorter development times and a sound software design has been recognized and addressed in the software industry for a long time, these design goals have often been neglected while developing new software in the academic field.

In Molecular Modeling applications, the implementation from scratch is too time consuming, because the developer has to spend most of his time on the implementation of supporting code, only marginally related to the problem he is interested in. This supporting code usually handles file I/O, data import/export, standard data structures, or visualization and output of results. The only way to reduce the development time is to reuse reliable code. The main goals of software reuse are reduced development and maintenance times, as well as increased reliability, efficiency, and consistency of the resulting software. There are different approaches for code reuse. According to Coulange [7] and Meyer [16] the object-oriented (OO) approach is the most favorable one.

In the course of our protein docking project [14], we searched for a suitable software package that has the above-mentioned properties. First of all, we had a look at commercial MM software packages. Most of them come with either a scripting language [13], or even a dedicated software development kit (SDK) [4]. They are quite expensive and do not allow the development of free software. Additionally, these packages are solely based on the procedural programming paradigm. To our knowledge, there is no object-oriented package commercially available. Out of academia only a few class libraries have evolved for Molecular Modeling, although in chemical engineering object-oriented programming is quite common and class libraries exist in abundance. Hinsen [12] presented his *Molecular Modeling Tool Kit*, which embeds efficient (C-coded) routines for force-field-based simulations into the object oriented scripting language Python [19]. The other existing libraries or tool kits concentrate on very specialized areas (*e.g.* PDBLib [5], a library for handling PDB files) or are only remotely related to Molecular Modeling (*e.g.* SCL [18], a class library for object oriented sequence analysis).

None of the above packages satisfies the basic requirements for rapid prototyping in Molecular Modeling, because they are either not state-of-the-art concerning software engineering or lack the required functionality. Thus there seemed a great need for a well designed, efficient application for rapid prototyping in the area of Molecular Modeling. Our answer to this is BALL. The library is implemented in C++, chosen over Java and other OO languages because of efficiency and wide acceptance. Since BALL is intended for rapid software prototyping, our principal design goals were *extensibility*, *ease of use*, and *robustness*. We spent much effort on the object-oriented modeling of the fundamental concepts in biochemistry like atoms, residues, molecules, and proteins. By the use of the composite design pattern [9] and polymorphism, we were able to model the multitude of

complex biochemical concepts in a well-structured and comprehensible class hierarchy, our *kernel* classes. The isomorphism between the biochemical structures and the kernel classes leads to an intuitive interface. With the *processor* concept (see Section 4), we also introduce an efficient and comfortable way to implement operations on all these kernel data structures. The kernel makes use of some basic data structures included in BALL, the so-called *foundation classes*. They provide classes for mathematical objects (*e.g.* matrices), hash associative containers, an extended string class, advanced data structures like three-dimensional hash grids, and many more. The foundation classes are partially based upon the Standard Template Library (STL) that is part of the ANSI C++ Standard [1]. A careful in-depth analysis of many typical Molecular Modeling applications identified data import/export in various formats, Molecular Mechanics simulations, and visualization of molecules as their most important functional components.

BALL provides not only a fully implemented Molecular Mechanics force field (AMBER 95 [6]), but also support for generic force fields. This support includes automated reading and interpretation of the required parameter files, data structures for these parameters, and modular force field components. Using this infrastructure, it is possible to implement user-defined force fields with minimal effort. Visualization of the results is often an important part in Molecular Modeling. However, providing a portable and flexible three-dimensional visualization of molecules is quite challenging. We decided to use QT [2] and OpenGL for this purpose. This combination allows the construction of portable, GUI-based applications on all platforms. Since the construction of the GUI is usually quite a time consuming process, we also implemented a stand-alone visualization tool that can be easily integrated into BALL applications through a stream-based interface. Using the object persistence implemented for the kernel classes, the application simply streams the objects to be visualized to the viewer. The stream that connects the viewer and the application is usually a socket, thus the viewer may also run on a different machine. Apart from this basic functionality, we also included some more advanced features. For example, BALL also contains efficient code to solve the linear Poisson-Boltzmann equation, a differential equation describing the electrostatics of biomolecules in solution. Using this Poisson-Boltzmann solver, we are able to model the effects of water on biomolecules that have long been neglected by simpler models. The implementation of the Poisson-Boltzmann solver was the first test for BALL's rapid prototyping abilities. Making extensive use of the BALL foundation classes, its basic functionality was implemented within a week. We also included functionality to identify common structural motifs in proteins and search for these motifs in large data sets. The usefulness of BALL was shown by the implementation of an algorithm that maps two similar protein structures onto each other. Whereas the first implementation took about five months (implemented in the course of a diploma thesis [3]), we were able to re-implement the algorithm using BALL within a day. The BALL implementation even showed better performance than the original implementation.

In Section 2 we discuss the main design goals of BALL. After giving an overview in Section 3, we will describe the object-oriented modeling of the fundamental biochemical concepts in Section 4. The foundation classes are summarized in Section 5, as is the functionality of the remaining components in Section 6. Section 7 presents an example of a BALL application. The future development of BALL is then outlined in Section 8.

2 Design goals

There were two main goals we had in mind when we designed BALL: its ease of use and its extensibility. Of course, *ease of use* is a very important property when it comes to rapid software prototyping. The user should be able to use the basic functionality immediately with a minimum of knowledge of the whole library, but what he learns should be globally applicable. To achieve this intuitive use, we designed the class hierarchy of the molecular data structures in a way that directly reflects the biochemical concepts and their relationship (see Section 4). This requires a consistent, minimal, complete, and well documented interface. The call for minimal interfaces is partially a contradiction to the requirements of rapid software prototyping. We break this rule and add additional functionality to the interface, if this leads to a significant decrease in source code size of standard applications (Rule: standard operations should be performed in not more than three lines). We also aim for an easily comprehensible, natural syntax. The last aspect of the ease of use, but certainly not the least, is a complete and consistent documentation. This was achieved by integrating the documentation of each class into the header files. The documentation may then be extracted in LaTeX or HTML format using the tool `doc++` [22].

Extensibility means basically two things: seamless integration of new functionality into the framework and compatibility with other existing libraries or applications. Compatibility is achieved through the use of namespaces. This guarantees no name collisions with other libraries such as the Standard Template Library (STL, now a part of the ANSI C++ standard [1]) and libraries like CGAL [8] or LEDA [15]. Compatibility with these libraries is particularly important, because computational geometry as well as graph algorithms play an important role in Molecular Modeling. The *efficiency* of our implementations has sometimes been sacrificed in favor of flexibility and especially robustness. However, time critical sections like the numerical parts (*e.g.* Poisson-Boltzmann solver, Molecular Mechanics simulations) have been carefully optimized for space and time efficiency.

Verifying the *correctness* of an implementation is very difficult in general. In the case of BALL, this is partly due to the extensive use of templates and partly due to some of the algorithms implemented. For example, it is difficult to verify whether an algorithm for the numerical solution of differential equations works correctly for all possible inputs. We decided to provide at least some basic black box testing whether every member function compiles correctly and behaves as intended. A test program is provided for each class (at least this is intended; not every test is yet completely implemented) that tests all member functions of the class. The resulting test suite is automatically compiled and run after the compilation of the library thus identifying compile-time as well as link-time problems.

Since BALL uses some of the more advanced features of the ANSI C++ Standard, such as member templates, namespaces, and run-time type identification, at the moment only a limited set of platforms is supported. However, we expect most compilers to support these features soon. Up to now, we support the following platforms:

- i386-Linux using `egcs 1.1.x` or Kuck&Associate's KAI C++
- Solaris 2.6/7 (SPARC and Intel) using `egcs 1.1.x`

- IRIX 6.5 using MipsPro 7.2.x

We also intend to port BALL to other platforms, like AIX, HP-UX, and Microsoft Windows. A special problem concerning the portability is the visualization. We chose QT [2] to support three-dimensional graphics on all platforms. The underlying 3D rendering is performed by OpenGL, which can be substituted on non-OpenGL systems by MESA a free library that converts OpenGL calls to X windows calls.

3 Library Overview

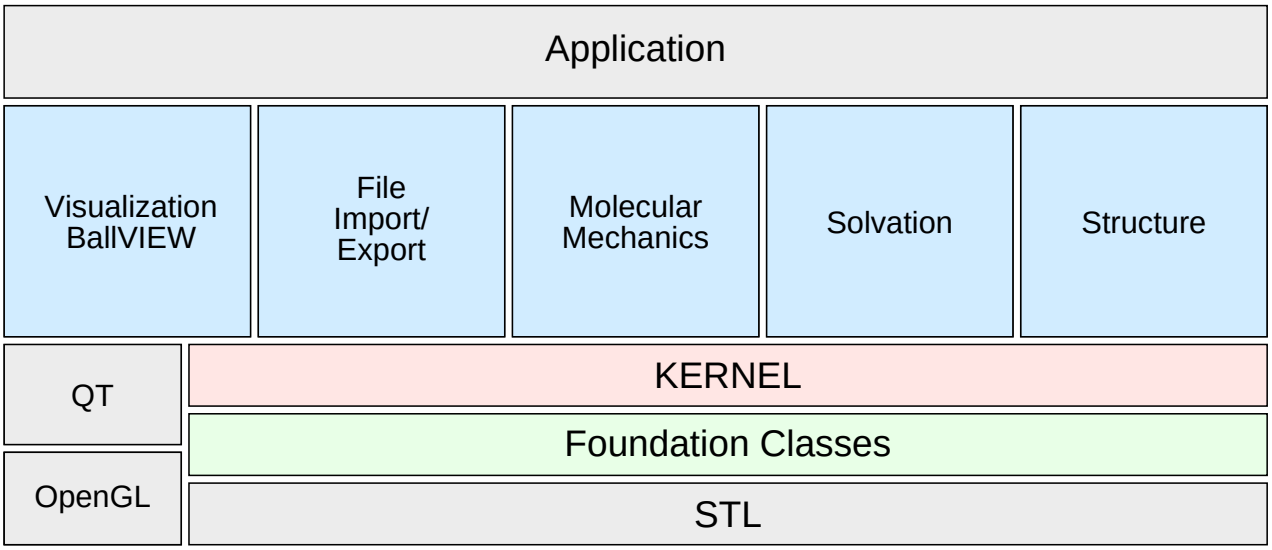

Figure 3.1: Overview of the components of BALL

BALL consists of several, inter-dependent components (Figure 3.1). The central part of BALL is the *kernel*, a set of data structures representing atoms, molecules, proteins, and so on. Design and functionality of the kernel classes will be described in Section 4. The kernel is implemented using the *foundation classes* that extend – and partially depend on – the classes provided by ANSI C++ (*e.g.* the string class). These three layers (STL/ANSI C++, foundation classes, and kernel) are used by the different *basic components* of the fourth layer. Each of these basic components provides functionality for a well-defined area: *file import/export* provides support for various file formats, primarily to read and write kernel data structures. The *visualization* component BALLVIEW provides portable visualization of the kernel data structures and general geometric primitives. The *Molecular Mechanics* component contains an implementation of the AMBER95 force field and support for user-defined force fields. The *structure* component provides functionality for the comparison of three-dimensional structures, mapping these structures onto each other and searching for structural motifs. The *solvation* component primarily contains a numerical solver for the Poisson-Boltzmann equation. The behavior and properties of solvated molecules, *i.e.* molecules in solution can be described using this equation.

A typical BALL application makes use of kernel data structures, the foundation classes, and one or more basic components, for example it uses the import/export component to read a molecule from a file, performs some simulation using the Molecular Mechanics component and visualizes the result using BALLVIEW.

4 Object-oriented modeling of biochemical concepts

As its name tells, Molecular Modeling software manipulates representations of molecular systems at an atomic level. Since these molecular systems may contain complex molecules of different kinds, one of the most difficult problems that we faced during the development of BALL was the design of a suitable class hierarchy to model all entities in these systems. Since most readers will not be very familiar with these biochemical models, we will outline them briefly. First of all, the whole "world" is built of *atoms* which are connected via *bonds*, thus forming *molecules*. Molecular Modeling usually deals with biomolecules (molecules of biological origin or relevance). Out of the many different classes of biomolecules proteins, nucleic acids (the carriers of hereditary information), and polysaccharides (sugars) are the most important ones. Common to these three classes is that their structures can be seen as a sequence (polymer) of smaller building blocks, so-called monomers. *Proteins*, for example, can be seen as chain molecules built from amino acids. We refer to the amino acids in such a *chain* as *residues*. Parts of a chain may form spatial periodic structures – like helices or strands – that are called *secondary structure* elements. *DNA* and *RNA* molecules are linear polymers built from monomers called *nucleotides*. We also introduce the term molecular *system* for a collection of molecules or biomolecules. The relationships between these entities is shown in Figure 4.1.

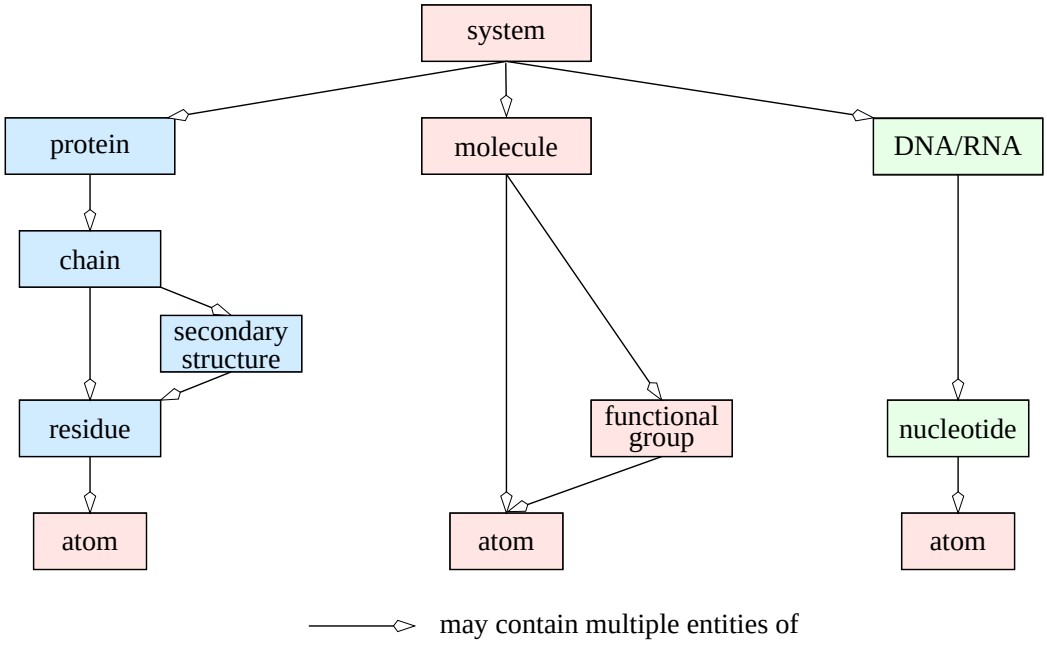

Figure 4.1: Hierarchical structure of molecules, proteins, and nucleic acids

Just from these brief definitions, we can deduce some basic properties of the data structures required. We recognize some `is-kind-of` relationships: a protein is a molecule, likewise a sugar. This is modeled by inheritance. Almost all of these "objects" printed in italics above represent containers: proteins contain chains, that contain secondary structures, that contain residues and so on. We also want to reach a certain uniformity between all these container classes. Most of our algorithms should operate on molecules as well as on residues or systems. We therefore model this relationship via the composite design pattern [9].

Without describing the different classes in depth, we give an overview of the molecular data structures in Figure 4.2, describing our solution domain. The base class of all our molecular classes is the `Composite` class. It implements a protocol (insertion, removal, and traversal of children) and may contain an arbitrary number of children, thus representing a tree structure. `BaseFragment` specializes the concept of a composite to an object representing molecular data structures. Also derived from `Composite` are `Atom`, `Bond`, and `System` (which represents a collection of molecular objects). Molecules are represented by `Molecule`, a specialization of `BaseFragment` that may contain either `Atoms`, or `Fragments`, which usually represent groups of atoms within a molecule. `Atom`, `Fragment`, and `Molecule` are used to represent general molecules; we call them the molecular framework. Biomacromolecules are represented by specializations of these classes. There are two frameworks already implemented besides the molecular framework: the protein framework and the nucleic acid framework, both representing a specialization and extension of the molecular framework.

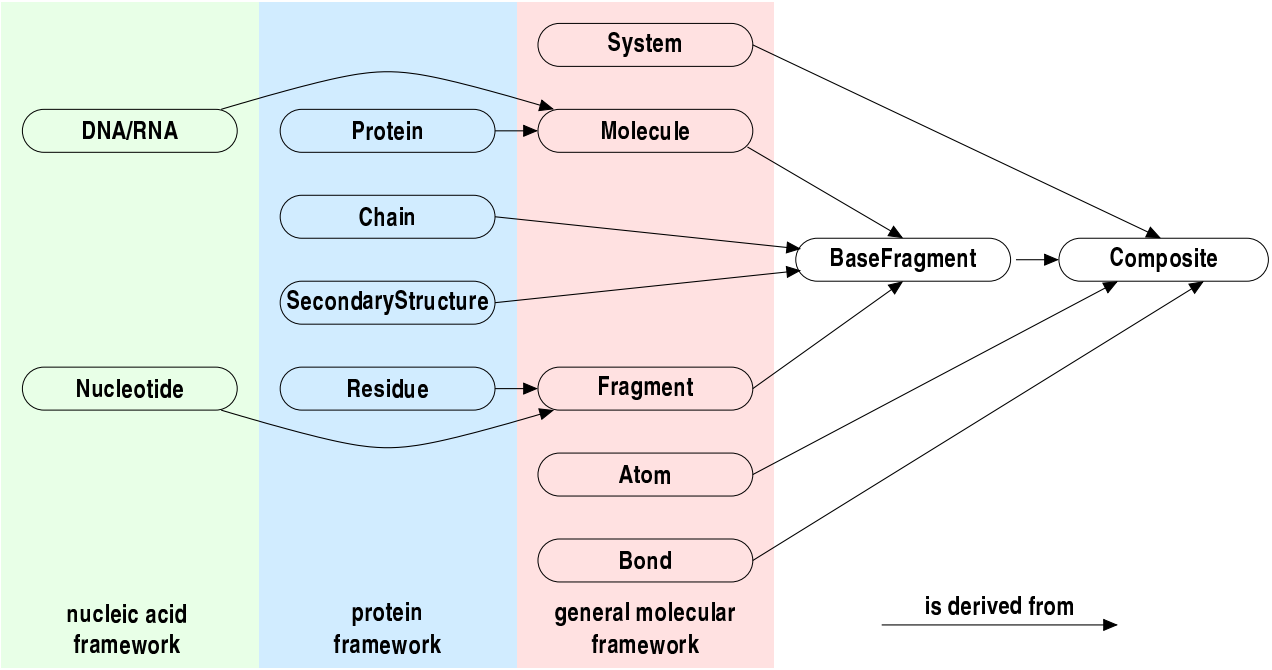

Figure 4.2: The hierarchy of the classes representing molecular data structures

In the protein framework, `Protein` replaces the `Molecule`. A Protein may contain `Chains` that may contain `Residues`. A `Residue` is a specialized variant of a `Fragment` possessing additional functionality, such as methods to decide whether it represents a terminal residue of a chain (which often must be treated differently from residues in the middle of a chain). `Chains` may also contain `SecondaryStructures`, if such information is available. In the nucleic acid framework, there exist two specialized classes derived from `Molecule`: `DNA` and `RNA`. Each `DNA` or `RNA` instance may contain `Nucleotides`.

Using this class hierarchy, we have the possibility to model the specific properties of biological macromolecules. Due to polymorphism we can design algorithms that may be applied to molecules or fragments of molecules in general. One might argue that the decomposition we chose to represent the

different biochemical terms and their relationships is the best. The difficulty with these biochemical terms stems from their lack of precision. For example, it is doubtful, whether a protein "is a" molecule. Every biochemist would confirm, that proteins are biomolecules. If asked for a definition of molecule, his or her answer would describe something like all atoms that are connected to each other via bonds. However, this definition does not apply to all proteins, because a protein might also be a non-bonded complex of multiple "molecules". After long discussions, we found the hierarchy we described before the most satisfying and the most intuitive one, but we do not expect it to fit all needs. Considering BALL to be a *framework* instead of a class library, this problem can easily be solved by a user-derived set of classes that satisfy these specialized needs. An example of this could be a framework representing polysaccharides.

We also introduce the concept of a `processor`. The processor performs an operation on objects and is parameterized with the class type of these objects. Each processor defines three basic methods: `start`, `operator ()` and `finish`. The typical usage of a processor is to `apply` it to a kernel object. `BaseFragment` defines a template member function `apply` that accepts arbitrary processors. First it calls the processor's `start` method, which is sometimes used to perform some initializations, but often left unimplemented. Every `Composite` can be seen as the root of a subtree. The `apply` method now iterates over all nodes in this subtree. Using run-time type identification, the actual class of each node is determined. If it matches the processor's template argument (*i.e.*, if the processor was parameterized with a base class of the node's type), the processor's `operator ()` is called for that node. After the end of the iteration, `finish` is called to perform some final action (*e.g.* clearing up temporary data structures). This concept is an elegant way to define operations that can be performed on any kernel class. It is very convenient to use, because the `apply` method usually takes the place of more complicated iteration loops. The implementation of the processor is trivial, because in most cases only three member functions have to be written.

5 Foundation Classes

Besides the kernel described in Section 4, BALL also provides more fundamental data structures. We will refer to these as the *foundation classes*. Since the total number of foundation classes (not counting exception classes and nested classes) is roughly 100, we can only present some selected classes here.

One of the most important classes is the `String` class that is derived from the ANSI C++ `string`. This class is especially useful if complex string handling is needed, *e.g.* in the parsing of line-based file formats. The `String` class allows case insensitive comparisons, efficient substring handling, among others. The foundation classes also provide more advanced data structures, such as hash associative containers, different types of trees, or three-dimensional hash grids (see the example in Section 7). Another part of the foundation classes provides mathematical and geometric objects like matrices, vectors, points, spheres, and so on. Object persistence was implemented for the kernel classes and for most of the foundation classes using some basic routines to simplify the serialization, to resolve interdependencies, and to translate pointers and references to objects to their new locations after reading a persistent object. In combination with the `SocketStream` classes, we are able to transport objects from one application to another via the network, a feature we use for the visualization of kernel objects (see Section 6).

6 The basic components

The Import/Export component

File import and export is an important basic functionality that every application in the field of Molecular Modeling uses. The molecular structures that an application has to handle come in a multitude of different file formats and sometimes even in different variants of these formats. Since tools are available to convert between most of these formats (*e.g.* BABEL [20]), we implemented only some of the important file formats: The Brookhaven Protein Databank (PDB) format, the MOL2 format from Tripos Associates and the HIN format used by HyperChem. We also added support for Windows INI files (used to store table-like data such as force field parameters) and a special format for hierarchically organized data that we use to store data of molecular fragments such as amino acids or nucleotides.

The Molecular Mechanics component

Molecular Mechanics is an important method to simulate the dynamic behavior of molecules and to predict their properties (*e.g.* geometry or relative energies). It is based on classical mechanics, describing atoms as mass points connected by bonds. The behavior of these atoms can be described in terms of a *force field*, *i.e.* a set of analytical descriptions of interactions between the atoms. The force field describes the total energy for a given molecular system. The forces acting upon each atom are obtained by calculating the gradient of the energy.

A *molecular dynamics simulation* is used to examine the dynamic behavior of molecules. The forces acting upon the atoms are evaluated after discrete time steps and the displacements of the atoms are calculated from these forces. Another application is *geometry optimization*, *i.e.* the search for an atom arrangement of the molecular system with the lowest possible energy. Molecular dynamics and geometry optimization are two very important, but time consuming applications in Molecular Modeling. Efficiency is a big topic here, because molecular dynamics simulations and geometry optimizations usually require a large number of iterative evaluations of the force field. An inefficient evaluation of the force field will always result in dramatic performance losses. To achieve optimal performance, we had to convert our kernel data structures to more compact data structures, containing just the information we needed to evaluate the force field. This conversion is done only once before the calculation starts. After the calculation, the kernel data structures are updated with their new values. As the time spent in the force field calculations is usually much larger than the time needed to set up these internal data structures, this overhead does not matter.

Since there are so many important force fields in use, we designed this component to be modular and easily extensible. We implemented a number of support classes that free the user of some of the annoying and difficult work. We call the classes the *generic force field*. The generic force field is able to automatically read and interpret force field parameter files and even handles multiple versions of parameter sets in one file. From the parameter files it constructs predefined data structures for most of the common force field terms. It also automatically constructs the fast internal data structures and assigns atom types according to a user-defined rule set. Leaving only the implementation of the force field terms to the user, it is possible to implement a simple force field within a day. Using the

generic force field, we implemented AMBER95, one of the most important force fields for modeling proteins. Further force fields are planned, but not yet implemented.

The visualization Component BALLVIEW

The three-dimensional visualization of molecules is a challenging subject. First of all, efficiency is a problem, because large biomolecules usually consist of several thousands of atoms. Visualization also requires a graphical user interface (GUI) that has to be portable. Considering these requirements, we decided to implement BALLVIEW using OpenGL. By compiling our kernel data structures into OpenGL display lists, we achieve optimal performance. We chose QT [2] as a platform-independent tool kit for the construction of graphical user interfaces. QT is free (at least on most platforms and for academic use), it is thoroughly tested, and provides a well designed class hierarchy.

Nevertheless, the development of a GUI is often too time consuming while testing new ideas. So we had to come up with a method to integrate a visualization component into BALL applications with just a few lines of code. We developed a stand-alone viewer that may be integrated via TCP sockets into a BALL application. If the application needs to visualize some kernel data structures (*e.g.* a protein), it sends these objects to the viewer via a stream connected to a socket, an operation requiring a single line of code. This feature uses the object persistence that we implemented for the kernel objects.

Further Components

Two other components that we will only briefly discuss are the solvation component and the structure component. The solvation component is used to describe the interaction of biomolecules with solvents, usually water. These interactions may be modeled using the so-called continuum model. The electrostatic part of these interactions is described by the Poisson-Boltzmann equation, a differential equation connecting the electrostatic potential and the charge distribution in the molecule. We implemented a modified version of the algorithms described by Nicholls and Honig [17] for the numerical solution of the Poisson-Boltzmann equation. It solves the linear Poisson-Boltzmann equation in its finite difference form on a grid using the method of successive over-relaxation.

The structure component allows the geometric comparison of molecules, the calculation of transformations that map one molecule onto the other, and provides functions for the search for common structural motifs in proteins.

7 Example: the ProteinMapper class

Using BALL, we have implemented several applications that stem from the area of chemical modeling and simulation. The implementation of these applications gave us the opportunity to test the usability of BALL. One of the tests that we carried out was the re-implementation of an algorithm for matching the 3D-structures of proteins, which had been implemented in the course of a diploma thesis [3]. The first implementation without BALL took more than five months; using BALL we were able to re-implement the algorithm within one day. Although implemented with a tool for rapid software prototyping the re-implementation has exhibited even better running times than the original implementation. In this section we describe the algorithm for matching the 3D-structures of proteins, present the BALL implementation of the algorithm as an example of a BALL program, and discuss some additional features of BALL.

In Section 4 we briefly introduced one of the most important classes of biological macromolecules, the proteins. Proteins are linear polymers (chain molecules) composed of twenty monomers, the so-called amino acids. The amino acid chain of every protein folds in a specific way. This folding process puts the protein into a well defined three-dimensional shape that has a strong influence on the chemical reactivity of the protein. The van-der-Waals model approximates the 3D-structure of a protein as a union of spheres, where each atom is represented by a sphere. The first column of Figure 7.1 shows the van-der-Waals models of two lysozymes (enzymes destroying cell walls of bacteria), the upper model shows rainbow trout lysozyme and the lower model hen lysozyme. The different colors indicate the different amino acids. The middle column of Figure 7.1 shows the so-called tube model of the two lysozymes. These models indicate in an abstract way how the amino acid chains of the two lysozymes are folded. Another simplified model represents the backbone of a protein by a discrete set of points. Every amino acid is represented by the coordinates of the atom center of its "central" C_α-atom that belongs to the backbone of the protein. The nature of the amino acid binding guarantees that all pairs of neighboring C_α-atoms have roughly the same Euclidean distance. Thus the C_α-atoms are in some sense "uniformly" distributed along the backbone. Because of the high packing density of proteins and certain chemical properties the C_α-atoms form also a good discrete representation of the volume of the protein. The last column of Figure 7.1 shows the sets of C_α-atoms for the two lysozymes. Since the 3D-structures of proteins determine their chemical reactivity, the comparison of the 3D-structures or even parts of the 3D-structures of proteins plays an important role in the functional analysis of proteins. An abstract version of the structure comparison problem can be formulated as follows: Given two proteins P_1 and P_2 and their 3D-structures, check if P_1 is similar to P_2 or to a part of it (or vice versa). In this abstract formulation the important term "similar" is not defined. The study of the literature about structure comparison algorithms shows that a large variety of similarity definitions and algorithms for these special problem instances have been published. One of these approaches formulates the structure comparison problem as a point set congruence problem: Given a distance bound $\epsilon > 0$ and two three-dimensional point sets, S_1, the atom centers of the C_α-atoms of P_1, and S_2, the atom centers of the C_α-atoms of P_2, compute the maximal number $k \in N$, such that S_1 and S_2 are (ϵ, k)-congruent. Two point sets S_1 and S_2 are (ϵ, k)-congruent if there exists a rigid transformation M (rotation + translation) and an injective function $f : S_1 \longrightarrow S_2$ such that at least k points $p \in S_1$ are moved by the transformation M into

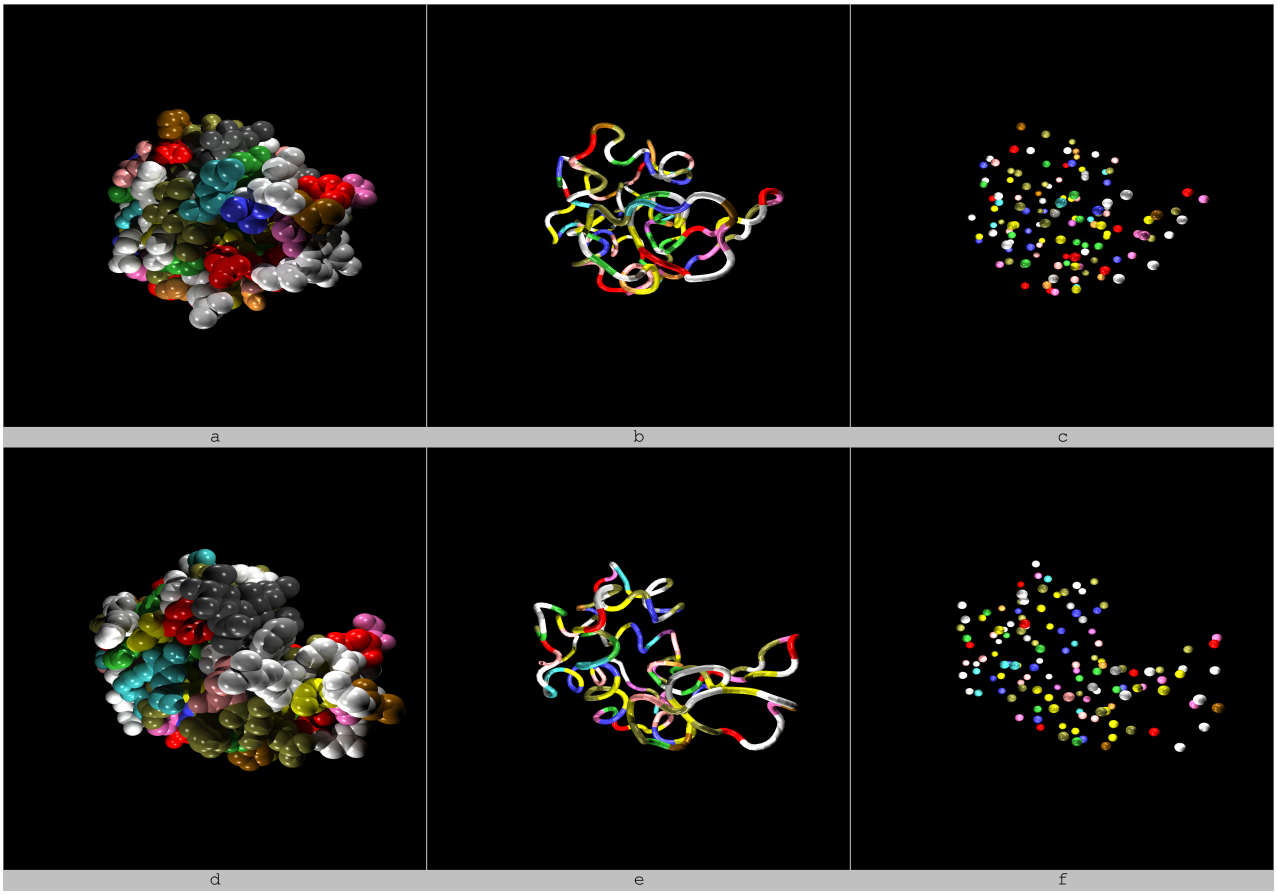

Figure 7.1: This figure shows different representations for the three-dimensional structure of two different lysozymes (upper row: rainbow trout lysozyme, lower row: hen lysozyme)

the ϵ neighborhood of the corresponding points $f(p) \in S_2$, *i.e.*, $d(M(p), f(p)) < \epsilon$, where $d(,)$ is the Euclidean distance. The definition of the (ϵ, k)-congruence is due to Heffernan [10], who published an algorithm for testing if point sets are (ϵ, k)-congruent. Tests with average-sized proteins showed that this algorithm has huge running times for values of ϵ suitable for protein similarity tests. In his Master's thesis [3], Becker developed and implemented a heuristic algorithm for testing (ϵ, k)-congruence extending techniques presented by Heffernan and Schirra [11]. We sketch this heuristic algorithm now.

- The algorithms determines a set T_1 of triangles between points in S_1 with edge lengths greater than l_{min} and smaller than l_{max}. The triangles of T_1 are stored in a three-dimensional hash grid G with box length ϵ. Each triangle (p_1, p_2, p_3) is stored in G according to the length $(d(p_1, p_2), d(p_2, p_3), d(p_3, p_1))$ of the "sorted" triangle edges in the grid box with indices $(\lfloor \frac{d(p_1,p_2)}{\epsilon} \rfloor, \lfloor \frac{d(p_2,p_3)}{\epsilon} \rfloor, \lfloor \frac{d(p_3,p_1)}{\epsilon} \rfloor)$. This technique is called geometric hashing [21].

- The algorithm computes a set T_2 of triangles between points in S_2. For each triangle t_2 in T_2 the algorithm searches for all triangles t_1 stored in the grid G that are ϵ-similar to t_2, *i.e.*, the absolute difference between the lengths of the ith edge of t_1 and the ith edge of t_2 is smaller than ϵ for every $i \in \{1, 2, 3\}$. For each ϵ-similar pair $(t_1 = (p_1^1, p_2^1, p_3^1), t_2 = (p_1^2, p_2^2, p_3^2))$ of triangles the algorithm computes a rigid transformation M that maps triangle t_1 onto triangle t_2. More precisely, the algorithm computes a rigid motion that maps point p_1^1 onto point p_1^2, p_2^1 onto the ray starting in p_1^2 that goes through the point p_2^2, and p_3^1 into the plane generated by p_1^2, p_2^2, p_3^2 (since we are considering only non-degenerate triangles, the point sets are not

collinear).

- For each rigid transformation M computed in the previous step the following operations will be carried out: The transformation M is applied to the point set S_1. For each point p_1^M in the transformed point set $M(S_1)$, the algorithm determines all points $p_2 \in S_2$ with $d(p_1^M, p_2) < \epsilon$. In order to determine the best injective function f for the given transformation M, the algorithm has to solve the maximum matching problem for the following bipartite graph formed from the points of S_1 and S_2. There is an edge between a vertex p_1 and a vertex p_2 if $d(M(p_1), p_2) < \epsilon$. If k is the number of edges in a maximum matching of this bipartite graph, then S_1 and S_2 are at least (ϵ, k)-congruent. The algorithm stores the transformation M_{best} that yields the maximal (ϵ, k)-congruence.

The following listing shows a short BALL program implementing the key parts of the algorithm. Several details (like include directives) have been omitted for brevity.

```
 1 int main(int argc, char** argv)
 2 {
 3   Protein P1, P2;
 4
 5   PDBFile infile(argv[1]);
 6   infile >> P1;
 7   infile.close();
 8
 9   infile.open(argv[2]);
10   infile >> P2;
11   infile.close();
12
13   vector<Vector3> S1, S2;
14
15   AtomIterator it;
16   for (it = P1.beginAtom(); +it; ++it)
17     if (it->getElement() == PSE[Element::C] && it->getName() == "CA")
18       S1.push_back(it->getPosition());
19
20   for (it = P2.beginAtom(); +it; ++it)
21     if (it->getElement() == PSE[Element::C] && it->getName() == "CA")
22       S2.push_back(it->getPosition());
23
24   int k = 0;
25   Matrix4x4 M = StructureMapper::calculateEpsilonKCongruence(S1, S2, k, 0.5, 4, 8);
26
27   cout << "Could match " << k << " Calpha atoms." << endl;
28   cout << "Best transformation is:" << endl;
29   cout << M << endl;
30
31   TransformationProcessor T(M);
32   P1.apply(T);
33   PDBFile outfile(argv[3]);
34   outfile << P1;
35   outfile.close()
36
37   return 0;
38 }
```

In lines 3 through 11 we read two PDB files into two proteins `P1` and `P2`. We then declare two STL `vectors` `S1` and `S2` parameterized with a three-dimensional vector (a BALL type: `Vector3`). The `for` loop in line 16 iterates over all atoms that are contained in `P1`. The overloaded `operator +` of `AtomIterator` is a shorthand for `AtomIterator::isValid()`. In the following line we extract all carbon atoms (carbon is defined as `Element::C` in the periodic system of elements `PSE`) with the name CA. The positions of these C_α atoms (coordinates of the atom centers) are then stored in the `vector` `S1`. Lines 20 through 22 contain the same loop for protein `P2`. After these preparatory steps, we use a static member function of `StructureMapper` to calculate the maximum k such that the two point sets `S1` and `S2` are (ϵ, k)-congruent. The resulting transformation `M` is then written to `cout`. We then initialize the `TransformationProcessor` `T` using the best transformation `M`. A `TransformationProcessor` multiplies the coordinates of all atoms contained in a kernel object with a matrix. The transformed protein `P1` is then written to a PDB file.

The listing of `StructureMapper::calculateEpsilonKCongurence` is shown in the following lines to illustrates some additional features of BALL.

```
 1 Matrix4x4 StructureMapper::calculateEpsilonKCongruence
 2   (vector<Vector3>& S1, vector<Vector3>& S2,
 3    Size& k_best, float epsilon,
 4    float l_min, float l_max)
 5 {
 6   typedef TVector3<int>                      IndexVector;
 7   typedef pair<IndexVector, Vector3>         Triangle;
 8
 9   vector<Triangle> T1 = computeTriangles(S1, l_min, l_max, epsilon);
10   vector<Triangle> T2 = computeTriangles(S2, l_min, l_max, epsilon);
11
12   THashGrid3<IndexVector>                    G;
13   Matrix4x4                                  M_best, M;
14   Size                                       k = 0;
15   vector<Vector3>                            transformed(S1.size());
16
17   k_best = 0;
18   G << T1;
19
20   vector<Triangle>::iterator                 t2;
21   THashGridBox3<IndexVector>::BoxIterator    box_it;
22   THashGridBox3<IndexVector>::DataIterator   t1;
23
24   for (t2 = T2.begin() ; t2 != T2.end(); ++t2) {
25     for (box_it = G.getBox(t2->second).beginBox() ; +box_it; ++box_it ) {
26       for (t1 = box_it->beginData() ; +t1; ++t1) {
27         M = matchPoints(S1[t1->x],       S1[t1->y],       S1[t1->z],
28                         S2[t2->first.x], S2[t2->first.y], S2[t2->first.z]);
29
30         for (Size i = 0; i < S1.size(); i++)
31           transformed[i] = M * S1[i];
32
33         k = calculateMaximumMatching(transformed, S2, 2 * epsilon).size();
34
35         if (k > k_best) { k_best = k; M_best = M; }
36       }
37     }
38   }
39
40   return M_best;
41 }
```

In lines 9 and 10 the triangle sets `T1` and `T2` are calculated by the function `StructureMapper::computeTriangles`. Using the overloaded `operator<<`, we insert the triangles of `T1` into the hash grid `G` (line 18). In lines 24–38 we use the iterator `t2` to traverse all triangles in `T2`. We then iterate over all triangles in the grid `G` (using `t1`) that are ϵ-similar to `t2` (for brevity, we removed the ϵ-similarity test, but the box size we chose for `G` guarantees (2ϵ)-similarity). For each of the similar triangle pairs we calculate the transformation M that maps `t1` onto `t2` (see description of the algorithm). In lines 30–31 we transform all points in `S1` using `M` and store them in `transformed`. We then calculate the maximum matching of `transformed` and `S2`. The function `StructureMapper::calculateMaximumMatching` returns a `vector` of point pairs. The number `k` of matched points equals the number of pairs in this `vector`. If `k` is larger than the best matching we found up to that point (`k_best`), we store it along with the corresponding transformation (line 33). Finally we return the best transformation in line 40. If the algorithm is applied to the two lysozymes from Figure 7.1, it successfully maps them onto each other thus revealing their similarity. The superposition of the two lysozymes is shown in Figure 7.2.

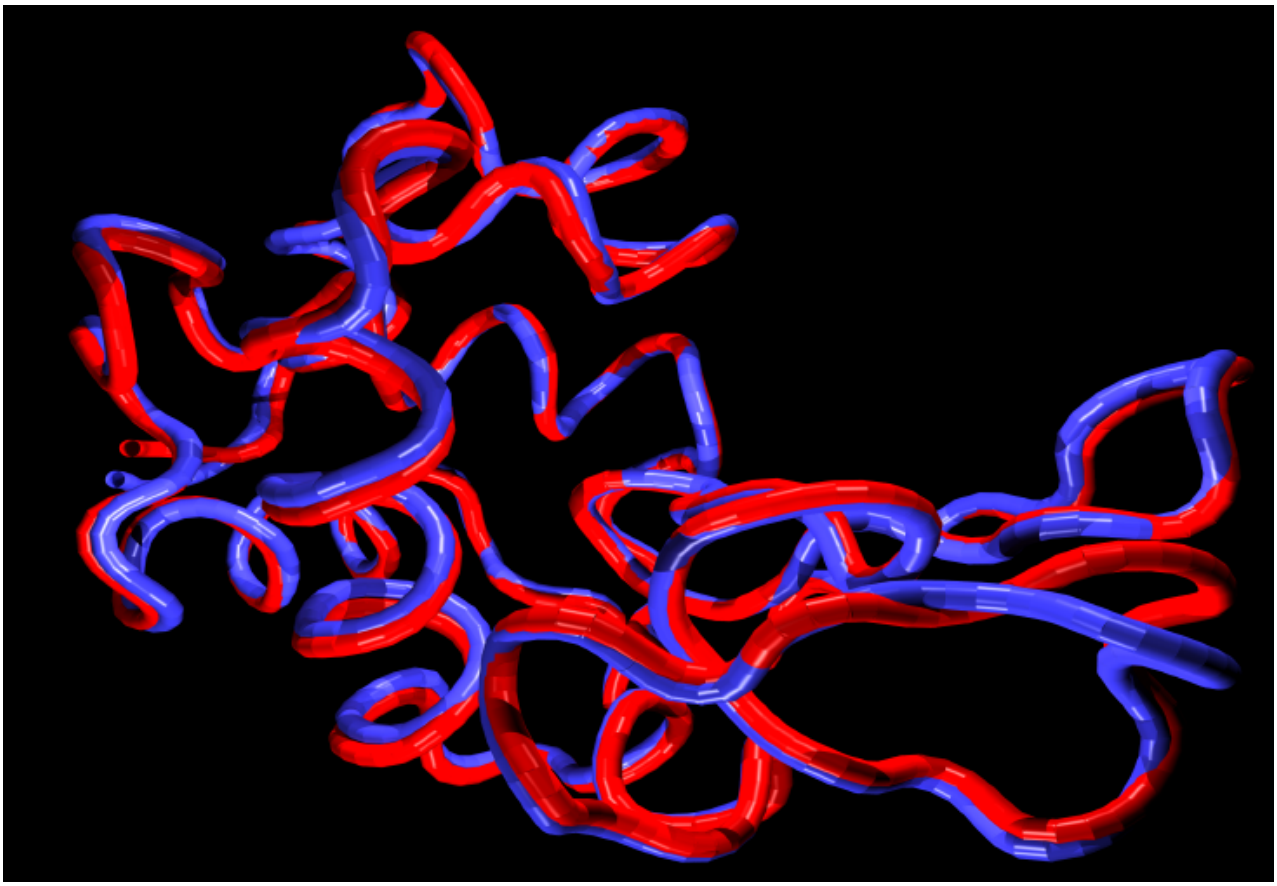

Figure 7.2: The tube representation of the two mapped lysozymes

8 Conclusion

BALL is the first object-oriented application framework for rapid software prototyping in the area of Molecular Modeling. It provides rich functionality and an intuitive interface. BALL has shown its usefulness in several example applications. In each case it decreased the development time tremendously without loss of performance.

However, it is still far from complete. We are currently implementing the missing parts of the test suite to improve the robustness of BALL. After the completion of these tests, we will release a first version of the library in summer 1999. After this release, additional functionality will be added. We are currently designing new components, for example a tool for the simulation of nuclear magnetic resonance spectra. A second version will also support additional platforms, including Windows NT.

Bibliography

[1] Programming Languages – C++. International Standard, American National Standards Institute, New York, July 1998. Ref. No. ISO/IEC 14882:1998(E).

[2] Troll Tech AS. QT release 1.42. `http://www.troll.no/products/qt.html`.

[3] Jörg Becker. Allgemeine approximative Kongruenz zweier Punktmengen im R. Master's thesis, Universität des Saarlandes, 1995.

[4] Cerius^2 modeling environment. Molecular Simulations Inc., San Diego, 1997.

[5] W. Chang, I.N. Shindyalov, C. Pu, and P.E. Bourne. Design and application of PDBLib, a C++ macromolecular class library. *CABIOS*, 10(6):575–586, 1994.

[6] Wendy D. Cornell, Piotr Cieplak, Christopher I. Bayly, Ian R. Gould, Kenneth M. Merz, Jr., David M. Ferguson, David C. Spellmeyer, Thomas Fox, James W. Caldwell, and Peter A Kollman. A second generation force field for the simulation of proteins, nucleic acids and organic molecules. *J. Am. Chem. Soc.*, 117:5179–5197, 1995.

[7] Bernard Coulange. *Software reuse*. Springer, London, 1997.

[8] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, the computational geometry algorithms library. Technical Report MPI-I-98-1-007, Max-Planck-Institut für Informatik, Saarbrücken, February 1998.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[10] Paul J. Heffernan. Generalized approximate algorithms for point set congruence. In *WADS93*, 1993.

[11] Paul J. Heffernan and Stefan Schirra. Approximate decision algorithms for point set congruence. *Computational Geometry : Theory and applications*, 4(3):137–156, 1994.

[12] Konrad Hinsen. The Molecular Modelling Toolkit: a case study of a large scientific application in python. In *Proceedings of the 6th International Python Conference*, pages 29–35, San Jose, Ca., October 1997.

[13] HyperChem release 4.5. Hypercube Inc., 1995.

[14] Hans-Peter Lenhof. New contact measures for the protein docking problem. In *Proc. of the First Annual International Conference on Computational Molecular Biology RECOMB 97*, pages 182–191, 1997.

[15] Kurt Mehlhorn, Stefan Näher, Michael Seel, and Christian Uhrig. *The LEDA user manual : version 3.6*. Max-Planck-Institut für Informatik, Saarbrücken, 1998.

[16] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, New Jersey, 2nd edition, 1997.

[17] Anthony Nicholls and Barry Honig. A rapid finite difference algorithm, utilizing successive over-relaxation to solve the poisson-boltzmann equation. *J. Comput. Chem.*, 12(4):435–445, 1991.

[18] Wolfgang Vahrson, Klaus Hermann, Jürgen Kleffe, and Burghardt Wittig. Object-oriented sequence analysis: SCL – a C++ class library. *CABIOS*, 12(2):119–127, 1996.

[19] Guido van Rossum. Python version 1.5.1. `http://www.python.org`.

[20] Pat Walters and Matt Stahl. BABEL version 1.6. University of Arizona.

[21] H. J. Wolfson. Model based object recognition by 'geometric hashing'. In *Proc. 1st European Conf. Comput. Vision*, pages 526–536, 1990.

[22] Malte Zöckler and Roland Wunderling. DOC++ version 3.2. `http://www.zib.de/Visual/software/doc++/`.

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Im Stadtwald
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2000-1-001	E. Althaus, O. Kohlbacher, H. Lenhof, P. Müller	A branch and cut algorithm for the optimal solution of the side-chain placement problem
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages
MPI-I-1999-3-002	J. Raskin, P. Schobbens	The Logic of Event Clocks
MPI-I-1999-3-001	S. Vorobyov	New Lower Bounds for the Expressiveness and the Higher-Order Matching Problem in the Simply Typed Lambda Calculus
MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments
MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA
MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?
MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable "open-world" desing in Java
MPI-I-1999-1-003	P. Sanders, S. Egner, J. Korst	Fast Concurrent Access to Parallel Disks
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing

MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid E-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-2-002	F. Jacquemard, C. Meyer, C. Weidenbach	Unification in Extensions of Shallow Equational Theories
MPI-I-98-1-031	G.W. Klau, P. Mutzel	Optimal Compaction of Orthogonal Grid Drawings
MPI-I-98-1-030	H. Brönniman, L. Kettner, S. Schirra, R. Veltkamp	Applications of the Generic Programming Paradigm in the Design of CGAL
MPI-I-98-1-029	P. Mutzel, R. Weiskircher	Optimizing Over All Combinatorial Embeddings of a Planar Graph
MPI-I-98-1-028	A. Crauser, K. Mehlhorn, E. Althaus, K. Brengel, T. Buchheit, J. Keller, H. Krone, O. Lambert, R. Schulte, S. Thiel, M. Westphal, R. Wirth	On the performance of LEDA-SM
MPI-I-98-1-027	C. Burnikel	Delaunay Graphs by Divide and Conquer
MPI-I-98-1-026	K. Jansen, L. Porkolab	Improved Approximation Schemes for Scheduling Unrelated Parallel Machines
MPI-I-98-1-025	K. Jansen, L. Porkolab	Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks
MPI-I-98-1-024	S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, M. Vingron	q-gram Based Database Searching Using a Suffix Array (QUASAR)
MPI-I-98-1-023	C. Burnikel	Rational Points on Circles
MPI-I-98-1-022	C. Burnikel, J. Ziegler	Fast Recursive Division
MPI-I-98-1-021	S. Albers, G. Schmidt	Scheduling with Unexpected Machine Breakdowns
MPI-I-98-1-020	C. Rüb	On Wallace's Method for the Generation of Normal Variates
MPI-I-98-1-019		2nd Workshop on Algorithm Engineering WAE '98 - Proceedings
MPI-I-98-1-018	D. Dubhashi, D. Ranjan	On Positive Influence and Negative Dependence
MPI-I-98-1-017	A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, E. Ramos	Randomized External-Memory Algorithms for Some Geometric Problems
MPI-I-98-1-016	P. Krysta, K. Loryś	New Approximation Algorithms for the Achromatic Number
MPI-I-98-1-015	M.R. Henzinger, S. Leonardi	Scheduling Multicasts on Unit-Capacity Trees and Meshes
MPI-I-98-1-014	U. Meyer, J.F. Sibeyn	Time-Independent Gossiping on Full-Port Tori
MPI-I-98-1-013	G.W. Klau, P. Mutzel	Quasi-Orthogonal Drawing of Planar Graphs
MPI-I-98-1-012	S. Mahajan, E.A. Ramos, K.V. Subrahmanyam	Solving some discrepancy problems in NC*