VALIS: A Multi-language System for Rapid Prototyping in Computational Biology

By

Salvatore Paxia

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2004

____

Bhubaneswar Mishra (Dissertation Advisor)

Acknowledgements

I would like to thank my advisor Bud Mishra, for his support and guidance over the last few years.

During my stay at NYU I have had the privilege of working with Jack Schwartz on many interesting projects. I am very grateful to him.

I would also like to thank my family, friends and colleagues. In particular Laura Cerami, David Bacon, Marco Antoniotti, Nadia Ugel, Joey Zhou, Raoul Daruwala, Archi Rudra, Arash Baratloo and Giuseppe Di Mauro.

In my first year at NYU I was supported by a generous grant from the Bonino-Pulejo foundation of Messina, Italy.

This research was sponsored in part by grants from the Department of Energy and the New York State Office of Science, Technology & Academic Research (NYSTAR).

Abstract

Bioinformatics is a challenging area for computer science, since the underlying computational formalisms span database systems, numerical methods, geometric modeling and visualization, imaging and image analysis, combinatorial algorithms, data analysis and mining, statistical approaches, and reasoning under uncertainty.

This thesis describes the VALIS environment for rapid application prototyping in bioinformatics. The core components of the VALIS system are the underlying database structure and the algorithmic development platform.

This thesis presents a novel set of data structures that have marked advantages when dealing with unstructured and unbounded data that are common in scientific fields and bioinformatics.

Bioinformatics problems rarely have a one-language, one-platform solution. The VALIS environment allows seamless integration between scripts written in different programming languages and includes tools to rapidly prototype graphical user interfaces.

To date the speed of computation of most whole genome analysis tools have stood in the way of developing fast interactive programs that may be used as exploratory tools. This thesis presents the basic algorithms and widgets that permit rapid prototyping of whole genomic scale real-time applications within VALIS.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software Requirements for Bioinformatics

During the last decade the various genome sequencing projects and related research have led to an unprecedented increase in the size, complexity and diversity of databases of biological data. One of the few pieces of equipment that is now ubiquitous in modern biology is the computer.

According to a National Institutes of Health definition, bioinformatics is “research, development, or application of computational tools and approaches for expanding the use of biological, medical, behavioral or health data, including those to acquire, store, organize, analyze, or visualize such data”, and is one of biology's fastest-growing technologies.

Biologists need easy-to-use software tools to form hypotheses, design experiments, analyze their experimental results, integrate their results into the growing body of biological knowledge, and help them to plan future experiments. They need easy interfaces that allow them to search databases, and compare their data with those of others.

Researchers analyzing whole genomes, and drug-discovery companies mining the genome for drug targets need high-throughput analysis tools to accelerate genome annotation and extract information from databases in more efficient and sophisticated ways.

Computer scientists play a crucial role in the emerging new discipline of bioinformatics. The underlying computational formalisms span a broad spectrum of computer science, including database systems, numerical methods, geometric modeling and visualization, imaging and image analysis, combinatorial algorithms, data analysis and mining, statistical approaches, and reasoning under uncertainty.

While the common “academic” approach for designing an environment for application prototyping would begin by surveying all the systems available, analyzing their weak points, and then proposing a novel synthesis and a perfect new programming language that would magically solve all the problems the biologists have, in this thesis we have taken a different road.

Too much effort has already been invested in developing with the tools currently available, and we cannot expect to displace them overnight. The approach we have taken is to come up with novel solutions whenever the current systems have proven deficiencies, like data storage and graphical user interfaces, but at the same time we have tried to reuse and integrate as much code as possible, when satisfactory solutions existed.

1.2 The importance of Scripting Languages

It is not surprising that many of the most successful systems and tools in bioinformatics have been written using scripting languages instead of system programming languages.

Scripting languages such as Perl [18], Python [19], Rexx [21], Tcl [20], Setl

[22], Visual Basic [91], and the Unix shells represent a very different style of programming from system programming languages like C and C++. Scripting languages are not intended for writing applications from scratch; they are intended primarily for plugging components together. For example, Tcl and Visual Basic can be used to arrange collections of user interface controls on the screen, and Unix shell scripts are used to assemble filter programs into pipelines. Scripting languages are often used to extend the features of components, but they are rarely used to implement complex algorithms and data structures; such features are usually provided by components written in lower-level languages. Scripting languages are sometimes referred to as *glue languages* or *system integration languages*.

In bioinformatics, scripting languages are very effective tools for the construction of customized analysis pipelines or workflows from existing building blocks and public domain database services. A characteristic of software applications in bioinformatics is that sometimes a program will be used only a few times to test a hypothesis and there will be no additional development after a prototype has been built. It is advantageous to develop such throw-away applications using a scripting language instead of a full-fledged system programming language, as industrial strength robustness of the application is not required.

1.3 Factors affecting the success of Scripting Languages

Ousterhout [8] is correct in suggesting that the success of scripting languages stems from the availability of faster computers, the increasing importance of graphical user interfaces, the advantages of weak typing when prototyping and the design of better high level languages. But he fails to point out the reason why certain scripting languages succeed while others are not widely adopted.

It can be argued that the success of a scripting language depends less on its design and its speed than: 1) the set of components it makes available; 2) the ease of development of new components; 3) the ability to interface easily and seamlessly with lower level languages.

These features are important for two reasons: 1) they make it possible to reuse the vast amount of public domain code and libraries available on the Internet; 2) particularly critical routines can be brought to high efficiency by coding them in native code. Once a set of useful components is made available to the users of a scripting language, higher-level code can be developed quickly. With a bit of luck, the language can become a de-facto standard for an application domain.

One can certainly claim that PERL and TCL are inelegant languages, but nevertheless, they are widely used. Why? PERL provides a good set of regular expression libraries combined with a syntax slightly less cryptic than AWK. TCL makes powerful graphical functionalities available. TCL became successful when the only alternative to develop a portable graphical user interface for a UNIX program was to use X directly, and when there were no cross-platform (Windows, UNIX and Mac) GUI toolkits for use. Python quickly became one of the most successful scripting languages because new libraries and types can easily be added to the core system, and the language itself can readily be integrated in other systems. Examination of the Python 2.3.2 sources shows that the core language consists of about 95,000 lines of “C” code, while the external native packages are more than 140,000 lines. Finally (and we will comment on this point later) there are about 270,000 lines of Python source code that are part of the Python Standard Library.

1.4 Integrating external code libraries

Currently it can be difficult to gain access to high performance native libraries from scripting languages. This would be easier to accomplish if the libraries were written with a specific language in mind. Even systems like SWIG (Simplified Wrapper and Interface Generator)[24], that can generate the glue code required to link a library to many different scripting languages, cannot deal well with libraries that operate with objects other than simple data types, as we will see later. Hence reuse of important native code is often surrounded by difficulties, which require much glue code to be written before a scripting language can be used effectively.

1.5 Efficiency and quality problems which arise when high level libraries are used

High-level code libraries developed for bioinformatics applications are sometimes of uneven quality, and can be plagued by efficiency problems.

To illustrate this point, let us look at the Bioperl [2][3] library for bioinformatics applications. This system is regarded as one of the most successful open source, community developed efforts in bioinformatics. As of February 2003 Bioperl was being developed by 45 major contributors, had 551 modules and about 80K lines of code. But, even one [25] of the "major contributors” complained that Bioperl was far from perfect. He noted that the average module length was only 144 lines of code, that many modules were untested or dead, and that there were multiple representations of a sequence, the basic data type for this kind of systems. His observations suggest a loose integration between the modules. Moreover the performance of the system was sometimes very poor, since a simple sequence creation and manipulation test, using the eight (!) different sequence classes available in Bioperl (which range from native Perl strings to LargeSeq objects), yielded results that vary in performance by <u>three orders of magnitude</u>.

This example suggests that for serious bioinformatics use the scripting languages must assume that there already exists a collection of useful end efficient components written in a lower level language.

Note that in some bioinformatics applications efficiency considerations can be paramount: dealing with and visualizing sequences of hundreds of megabytes and working with gene expression data for thousands of genes in hundreds of experiments are tasks requiring a completely different level of code efficiency than putting together a quick and dirty script to manage the files and directories in a filesystem (as in PERL or TCL/Tk).

1.6 The importance of inter-language communication

Another important but neglected area is reuse of existing higher-level code already available in a variety of scripting languages. Such reuse can be critical to rapid prototyping of bioinformatics applications. The Bioperl developers recognized this and still claim in their position paper that “Interoperation with modules written in Python and Java is supported through the evolving BioCORBA bridge". (However the BioCorba bridge has been dead since the beginning of 2003, and has never been productively used by the Bioperl developers.)

1.7 A summary of the thesis

This thesis addresses the problems outlined above by describing a coherent, language-independent framework for rapid and efficient application prototyping in bioinformatics. We call our system VALIS, (Vast Active Living Intelligent System), inspired by Philip Dick's 1981 science fiction novel. The thesis is organized as follows:

We first present an architecture that supports the sharing of both native and higher level code between multiple scripting languages. Then, descending to a lower level of detail, we describe the way in which various scripting languages support native code classes and we describe the most successful efforts to design native classes for use by multiple scripting languages.

Once this basic material has been presented we move to consider the important problem of bioinformatics data storage. Though relational databases have been very successful, they have clear shortcomings. This was made apparent a decade ago, when object database designers claimed to be presenting an important new paradigm. Handling the diverse data types encountered in bioinformatics requires powerful means for efficiently storing and retrieving general kinds of programming language objects without having to translate them into the limited set of formats supported by relational databases. We will describe a system, originally developed for the SETL language, but then implemented in C and made available in VALIS, that can cope efficiently with unstructured data and very large objects. This system offers facilities which go beyond those familiar to users of standard relational databases.

Chapter 4 then considers Graphical User Interfaces. The TCL Advocacy Page [26] suggests that TCL and Visual Basic are currently the only presently available scripting languages that directly support the rapid development of GUIs, since Python supports them by embedding TCL. In Chapter 4 we present a novel system which allows GUIs to be put together graphically, as in Visual Basic, but allows customization code and events for an application to be written in multiple scripting languages.

Having described these fundamental components of our environment in our initial chapters, we move on to show VALIS in action. We show how a few high performance components can be used to build a genome browser and a whole-genome comparison tool. Our system allows such tasks to be accomplished in any scripting language using only a few hundred lines of code. As a second example we show how we reimplemented Simpathica [27][82] inside VALIS. Simpathica is a set of tools for simulating and analyzing biological pathways, and was originally written in four different programming languages (Java, Lisp, Octave and C++). Here we show that we can maximize code reuse by selecting the best language for each required subsystem, and that we can link everything together seamlessly using the VALIS multi-language scripting interface.

Our last chapter compares the present effort with competing systems, e.g., Microsoft .NET, and describes some of the shortcomings of these other programming environments.

The VALIS system has been developed largely under Microsoft Windows, but we also show here how it can be ported to other platforms.

The VALIS system and examples described in this thesis are available for download at the VALIS website [1].

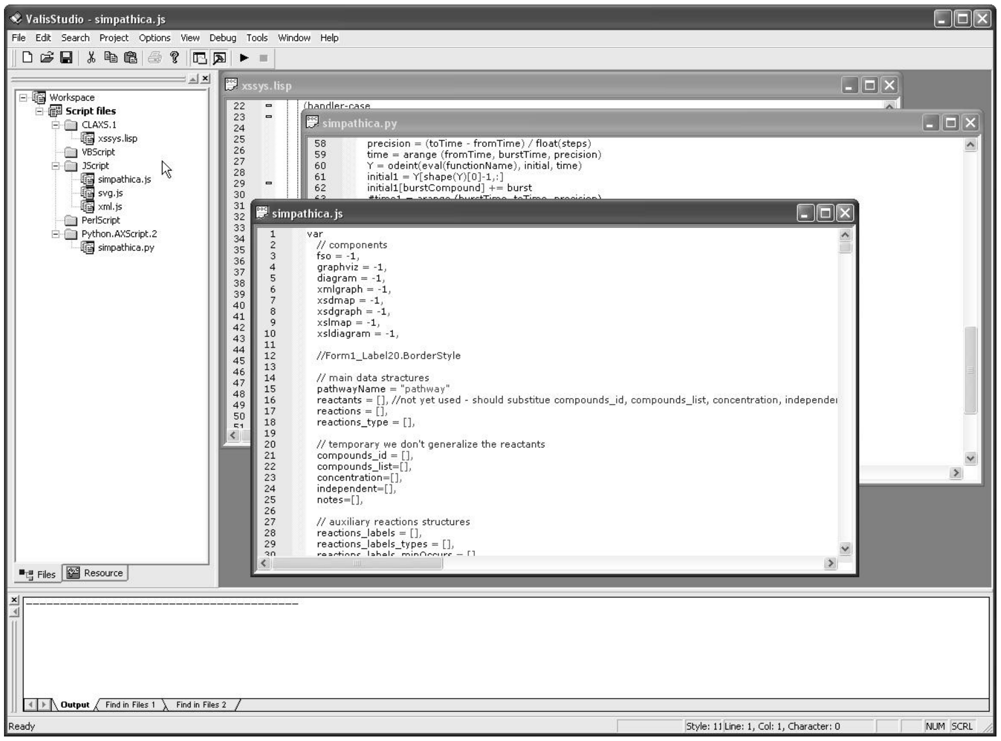
Chapter 2

Multi-language Scripting Capability

2.1 A VALIS mini manual

Users prototype applications in the VALIS Studio development environment pictured below.

2.1.1. The programming environment



```
(handler-case
```

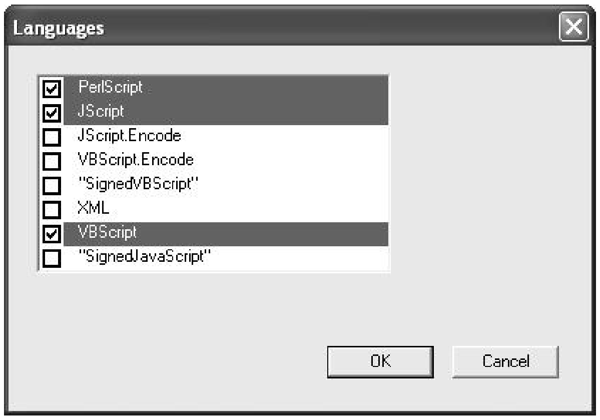
```
precision = (toTime - fromTime) / float(steps)
time = arange (fromTime, burstTime, precision)
Y = odeint(eval(functionName), initial, time)
initial1 = Y[shape(Y)[0]-1,:]
initial1[burstCompound] += burst
```

```
var
  // components
  fso = -1,
  graphviz = -1,
  diagram = -1,
  xmlgraph = -1,
  xsdmap = -1,
  xsdgraph = -1,
  xslmap = -1,
  xsldiagram = -1,

  //Form1_Label20.BorderStyle

  // main data structures
  pathwayName = "pathway"
  reactants = [], //not yet used - should substitue compounds_id, compounds_list, concentration, independe
  reactions = [],
  reactions_type = [],

  // temporary we don't generalize the reactants
  compounds_id = [],
  compounds_list=[],
  concentration=[],
  independent=[],
  notes=[],

  // auxiliary reactions structures
  reactions_labels = [],
  reactions_labels_types = [],
```

Figure 1. The VALIS Studio environment

VALIS organizes applications as ‘Projects’. A VALIS Project consists of a collection of *forms* and *script files*. Each script file can be written in a different language, but all can communicate within a common environment. VALIS can work with any scripting engine supporting the ActiveX scripting interfaces. Note that PERL, Python and Ruby, all support these interfaces.

As soon as a new scripting engine is installed, it will appear in the list of available languages in the *Options* menu available within VALIS:



- [✓] PerlScript
- [✓] JScript
- [ ] JScript.Encode
- [ ] VBScript.Encode
- [ ] "SignedVBScript"
- [ ] XML
- [✓] VBScript
- [ ] "SignedJavaScript"

Figure 2. The VALIS 'Language options' dialog

Once a set of languages has been selected for use in a VALIS project, the project workspace will show a folder for each language selected, as seen at the upper left of Figure 1. Script files can then be created by editing either in the VALIS Studio environment, or using another editor.

The VALIS Studio editing facilities offer language-specific syntax highlighting, auto completion (typing the initial part of a function recalls all the available functions with that prefix) and are integrated with the VALIS Form system (for auto completion of Form objects and method names).

Once scripts are saved to disk (as text files), they can be added to the VALIS workspace by dropping their file icons into the appropriate language folder. In the following picture, a file named ‘3.js’ has been added to the Jscript language folder. Other files have been added to the VBScript and PerlScript folders.

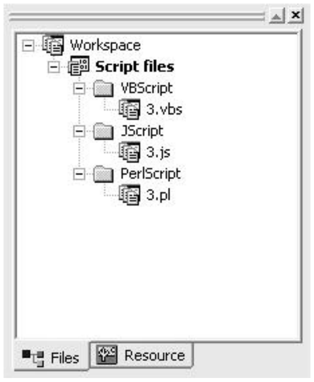


Figure 3. The language folders

VALIS then knows that ‘3.js’ is a Javascript script file. (These associations between script files and languages are necessary for VALIS to select the correct interpreter for parsing and execution.) Any number of script files can be added to each language folder, and any number of language folders can be used within a single workspace.

2.1.2. The VALIS namespace; application startup and termination

In VALIS all the scripts that are part of the same project workspace share a common inter-language communication namespace. Any function defined as ‘public’ in a script file will be accessible not only from other script files within the same programming language, but also from scripts written in other programming languages. These function names must be unique within the project.

When the VALIS application is started, execution ordinarily commences from the function whose name is ‘valismain’, which must be supplied within the project. (This name can be changed from the VALIS Studio Preferences menu.) Since in VALIS all scripting languages share a common namespace, this function can be defined in any programming language.

A VALIS run ends when the ‘valismain’ function returns, provided that no forms have been opened. This convention allows us to write two different kinds of VALIS applications: simple scripts, and scripts that manage a Graphical User Interface. Simple scripts do all their processing in the special main function ‘valismain’ (but can call other functions in any other of the programming languages available). VALIS graphical applications are also executed starting with the ‘valismain’ function, but in such an application this function will prepare and display at least one VALIS graphical form.

VALIS keeps track of the number of forms being displayed, and the application stops when the last form is closed. VALIS GUI applications, like other such applications, are driven by graphical events originating from the forms and their widgets.

2.1.3. Built-in VALIS library functions

VALIS provides all the scripting engines available in it with mechanisms for invoking and manipulating external objects. In particular all the scripts that are part of a VALIS application can communicate with the VALIS environment. This communication is realized using certain special predefined objects, described below.

When the execution of a VALIS application begins, VALIS creates and initializes all the scripting engines made available to it. Then the basic VALIS communication objects are linked to the engine environments. The most important such external object is the “Valis” object.

This object offers the following methods:

CreateObject(string program_id)	Locates the registry entry for and creates an external Microsoft COM object whose name is ‘program_id’ and returns a reference to it
Clear()	Clears the VALIS Studio output window
RGB(int R, int G, int B)	Macro returning the Color object with the specified Red, Green and Blue components
Alert(string message)	Displays the specified message and waits for user input
Print(string message)	Prints a string onto the VALIS Studio output window

Table 1. Methods of the built-in 'Valis' object

This single ‘Valis’ object is all we need to support the manipulation of components from all supported engines.

The most important method of the ‘Valis’ object is its CreateObject method.

Since in the Microsoft environment, on which we focus, scripting engines were originally designed for Web development inside Internet Explorer, some of the scripting languages, in which we are interested (notably Jscript and VBScript), do not support creation of new components owing to security considerations. Hence we must create new components outside the scripting engines environments, and we do it in VALIS using the CreateObject method. CreateObject returns a Microsoft COM IDispatch object reference. This standard communication interface is central to the way in which we share objects between different scripting engines and is described in some detail in section 2.4.1 below.

Any ActiveX component or library available on the machine executing VALIS can be used within an application being prototyped by calling the CreateObject function. Objects created by using this method can also be one of a number of built-in VALIS object types. Among the high performance internal native components which VALIS installs and registers are:

Valis.DB	The free format database described in a later chapter of the present thesis
Valis.Table	A flexible spreadsheet-like multi dimensional array data structure
Valis.DNASeq	A general object used for DNA sequence I/O and manipulation
Valis.Sarray	An efficient library for building and manipulating suffix arrays
Valis.Merclient	A library for high performance exact and inexact matching using suffix

	arrays
Valis.Seqdb	A specialized sequence database based on Berkeley DB
Valis.Sqlpack	A library to gain access to local and remote relational databases

Table 2. VALIS built-in objects

This library of built-in objects can be extended by integrating additional native code packages.

Other built-in VALIS controls, described in our next chapters, are available to aid in the construction of graphical user interfaces.

The following example (written in JavaScript) shows how CreateObject is typically used inside VALIS applications:

```
seqdb=CreateObject("Valis.Seqdb");
seq=CreateObject("Valis.DNASeq");
seq.Input("spam.fasta");
seq.SelectSequence(1);
l=seq.Length();
seqdb.ImportSeq("spam.db", "seq1");
Print("Fasta sequence imported");
```

Figure 4. Creating a BerkeleyDB style database object within VALIS

This example uses a Sequence object to read in and parse a FASTA file and store it in the local BerkeleyDB based database ‘spam.db’ with a ‘seq1’ key.

This small example illustrates a few important points: the VALIS environment provides various efficient components, like *Valis.Seqdb* and *Valis.DNASeq* in the example. As remarked above, the library of native components is extensible. The scripting engines of course know nothing about these objects, but can create the new components using the CreateObject method of the predefined “Valis” object.

2.1.4. Parameter passing and Object sharing

As seen above a user prototyping applications in VALIS can read in a sequence from a FASTA file in just one statement knowing only that a sequence object is named *Valis.DNASeq* and that it has a method called *Input* that will open, read and parse a standard sequence file. The user is otherwise free to use the programming language of his choice to build the application.

Once components like the *Valis.DNASeq* sequence object have been created, they can be controlled by multiple scripting engines. The only requirement is that the engine can call the methods of an IDispatch object reference (and this capability is supported by all the engines available to us at this time).

Since functions defined in any programming languages can be freely called from any other language, objects can be passed between programming languages for processing. While simple data types (integers, longs, strings, etc.) are passed by value, objects are passed by reference. Since some scripting engines may have no type information available at runtime (as these engine are disparate), and complex objects cannot be automatically converted (for the reasons outlined at the beginning of this chapter), the object references passed between languages must be considered “opaque” pointers. When a parameter passed across programming languages contains an object reference, the programming language called can use that reference and call any method that the object provides. This applies to both complex objects created by the VALIS libraries (as tables, databases, sequences) and to objects defined by any of the scripting engines used in an application.

2.1.5. What VALIS achieves

The VALIS approach improves on what can be achieved by the simple native package use supported by many scripting languages. Of course the VALIS components are standard ActiveX components, and they have to be written according to certain standards and include an IDL description file, but this is easier than having to generate the glue code for multiple languages manually (or automatically with a tool like SWIG). We will discuss this in more detail in sections 2.2 and 2.3 below.

Moreover, when a component is improved or updated, it only needs to be reinstalled and registered once to become available to all the scripting engines supported in VALIS; with standard native packages, at minimum the SWIG generated glue code must be updated and all the packages must be recompiled.

In addition, all high-level code manipulating VALIS native libraries can be called directly from any programming language through this multi-language scripting interface.

2.1.6. Project storage and VALIS Executables

VALIS Studio uses various internal data structures to maintain the association between script files and the correct engines for the Application we are prototyping. Projects can be saved to disk and read back into the VALIS environment.

The VALIS project descriptors are stored as simple XML files:

```
<?xml version="1.0" encoding="UTF-8"?>
<XMLConfigSettings>
  <files>
    <fileinfo>
      <filename>simpathica.vfr</filename>
      <name>Form1</name>
      <type>FORM</type>
    </fileinfo>
    <fileinfo>
      <filename>simpathica.js</filename>
      <name></name>
      <type>JScript</type>
    </fileinfo>
    <fileinfo>
      <filename>xssys.lisp</filename>
      <name></name>
      <type>CLAXS.1</type>
    </fileinfo>
    <fileinfo>
      <filename>generate.js</filename>
      <name></name>
      <type>JScript</type>
    </fileinfo>
    <fileinfo>
      <filename>simpathica.py</filename>
      <name></name>
      <type>Python.AXScript.2</type>
    </fileinfo>
  </files>
  <references>
    <refinfo>
      <typename></typename>
      <clsid>{CDCDCDCD-CDCD-CDCD-CDCD-CDCDCDCDCDCD}</clsid>
    </refinfo>
  </references>
</XMLConfigSettings>
```

Figure 5. A VALIS project descriptor

The project file contains a list of all the script and form files that have been added to the project, and a list of the project type libraries used (in the <references> section); see below.

VALIS applications can be distributed without the VALIS Studio environment being needed by converting them to self-standing executable programs using the “make executable” function, which VALIS supplies. This function reads all the scripts and form files of a project and produces an extended version of the project file, which it embeds into a VALIS run-time executable. The result is a stand-alone program which does not require the VALIS Studio environment. However, all the interpreters for the scripting languages used in the project must be present on the machine on which the application will be executed. Moreover, all the graphical user interface components that are used by the project must be registered with the operating system. The environment setup required for this can readily be handled by building a custom installation program (which includes all the components needed) for the VALIS application.

2.1.7. Project type libraries

One of the problems with native code libraries is that numerous constants and special values are usually needed to control the operation of the code. As it is infeasible to add the definitions of these constants manually to each scripting language, special "type libraries" can be made part of the environment of each VALIS application. The definitions available in these libraries are typically values such as color codes, special values and enumerations. The libraries can also contain function signatures that are used to improve the "auto complete" facility of the VALIS editing system.

2.2 Technical issues and design choices in extending scripting languages

We now turn to discuss some of the many technical issues which arise in constructing a system of the kind described in section 2.1 above.

The author's experience with the problem of integrating external, especially native software packages into a scripting language began with such an integration effort for the SETL language.

During this development, which involved the addition of several native packages to this high-level language, a few important lessons about extending scripting languages were learned:

i) Calling an external procedure written in a lower level language from a scripting language requires development of a substantial amount of interface code that depends on the scripting language used.

ii) The “C” language header files generally available with major C or C++ libraries do not convey enough information to generate the necessary glue code automatically.

iii) Often a considerable amount of high level code must be added to make the library usable, even after its functions are made callable.

As an example of the issues that arise in interfacing scripting languages to code libraries written in C or C++, let us consider the problem of implementing the following simple extension to the Python interpreter. Suppose that, in a UNIX environment, we want to call the “system” function, to execute a UNIX command from Python.

The C-written native functions available in the Python extensions usually have a standard signature that is designed to permit passing of a variable number of arguments. To be called from Python, arguments of such functions must be type checked, extracted, and then converted to standard “C” data types. The library function corresponding to the native function is then called, and its result packaged into a format suitable to the Python interpreter. We show a schematic version of the ‘glue’ code needed to accomplish just this much.

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

Figure 6. A typical 'glue code' fragment

This new Python native function, complete with glue code like that seen above, must be somehow registered with the interpreter. The conventions that apply to this depend on the scripting language used. However, although precise implementation details vary widely between different languages, the requirement for some kind of ‘registration’ is common to most of the interpreters.

2.3 Existing tools for integrating native packages

The interface code required for native library linkage can be generated either by custom code generators, or using more generic systems like SWIG[24]. The SWIG tool parses “C” header files and generates the ‘glue’ code described above automatically, also generating the makefiles necessary to build loadable modules. Packages generated by SWIG can therefore be adapted to many different scripting languages. For each scripting language, one simply needs to write a SWIG driver that will generate the proper glue code.

Although SWIG is a very useful tool, and the author was aware of it when he developed a large but otherwise typical native code interface (namely the SETL native package for the "Numerical Recipes in C" library[32]), SWIG could not used because of a simple but typical interfacing problem: there is not enough information in a C declaration to properly map C pointers to all the higher level constructs that such pointers can represent. For example, a "C" pointer declared as `int *` could be an array of integers, but it could also be an output parameter of the routine in whose header it appears. How do we distinguish between the two cases? And there is also an efficiency issue: Even if we could identify the C value pointed to correctly as an array, this could be so large that mapping it to some default scripting language structure, for example a list, would be a bad idea.

In the SETL native code interface described above, this problem was solved by writing a SETL code generator that read a custom designed template file containing an extended "C" signature language and output the necessary glue code (complete with type and size checking) and the higher level (SETL) source code needed to deal with the lower level code.

Of course this process can be made more general by providing all native libraries with an extended interface written in a specialized Interface Definition Language, which is exactly what CORBA [34] and Microsoft's COM [33] do. But even if such extended interface description were systematically available, there is another major problem with SWIG's approach. To understand this issue, consider the problems that need to be faced if one wants to embed the TCL programming language from another scripting language (both Python and SETL have packages that allow them to call TCL freely and use all of TCL's many interactive graphical widgets).

In this important case, since TCL is itself a string interpreter, one can create a complete communication interface simply by writing glue code for seven functions to pass strings back and forth between TCL and another scripting language. This is a simple enough task. But if one just provides these seven routines, without adding higher level glue code, the users of the language will be compelled to write TCL code inside the scripting language which hosts it.

To solve this problem, both Python and SETL provide higher level interfaces that generate all needed TCL code automatically via scripting language widget classes. Developing such an interface library for your favorite scripting language can be a substantial task, with which code generators like SWIG help hardly at all.

Another example is furnished by SETL's interface to the Numerical Recipes library. This provides several distinct classes for matrices, for example diagonal, tridiagonal, sparse matrices etc. A good high level interface should hide the complexity of its low level library from the user, and automatically select the best low level representation for the data. That is why SWIG or similar systems, even systems based on interface definition languages, will never be able to automatically generate fully acceptable glue code.

The VALIS solution to this problem is to allow one high level language to call directly procedures written in another such language. Then once the high level glue code has been developed for one scripting language, other languages can call the resulting high level procedures directly. One can thereby ignore whatever ugliness there are in the lower-level libraries or languages that ultimately provide the functionalities made available. This is potentially a very advantageous approach for bioinformatics applications. For example, one only needs a few functions to implement a socket interface library for a scripting language, but a “service” looking up genes in Genbank requires a fair amount of scripting code. One can hope to achieve near-optimal low level representations for biological data and select the most appropriate algorithm for processing it, all using a fairly high level API. If this can be achieved, even biologists confounded by computer details could develop high performance code.

Another reason for which high-level code reuse is desirable is that there is a very substantial amount of native code available via widely used scripting languages. We do not need to rewrite all such native packages as VALIS components at once; this can take place gradually, as required.

2.4 Combining scripting engines; Technical issues and design choices

The germ of our VALIS design is implicit in an often-overlooked capability of Microsoft’s Internet Explorer browser. Internet Explorer supports use of multiple languages inside a web page and allows the developers to choose a language using the HTML <SCRIPT> tag. By using the windows ‘registry’ and some standard COM primitives, Internet Explorer can identify the correct scripting engine component from the name appearing in the above-described tag. Many developers have overlooked the significance of this feature, since there are no clear advantages in using multiple languages in a simple web page. Since, to incorporate this feature, one must also make sure that the scripting engines are correctly installed into the user’s machine, this apparently esoteric feature has not seemed to be worth the work needed to exploit it. Also, Microsoft only guarantees that Jscript and VBScript are installed with Internet Explorer, there seems to be little reason for a user to download a potentially very large scripting engine like PERL or Python to view a web page.

The situation is different if we think not of Web pages, but of rapid prototyping in bioinformatics. In this context we can exploit the ActiveX scripting technology to allow multiple scripting engines to share high level code in a rapid prototyping environment. In this usage, the requirement that a scripting engine be installed is minor. Nevertheless, even though the extension of scripting languages using external libraries has attracted the interest of many developers, leading to the diffusion of tools like SWIG, little work has been done on welding multiple scripting engines together. Note that many public domain and commercial applications support an integrated scripting language, for example this is done in many terminal emulation programs, also in AutoCad, GNU Emacs and GNU Gimp. But there has been insufficient effort, especially by developers of public domain software, to agree on standard interfaces for embedding scripting languages in applications. Such interfaces would of course be very advantageous, as one could simply replace the application’s predefined macro language with a language of one’s choice.

The only system with the integration capability considered above that we could find, apart from Microsoft’s ActiveX scripting technology (which, since it is basic to VALIS, we will look into in detail below), is the Minotaur [35] system. But this is just a proof of concept system, which requires its author to deal with numerous different interfaces to embed the supported scripting languages. Moreover some of the languages that Minotaur tries to support were not written to be called by a “C” main program.

A few years ago the author encountered this same difficulty while trying to embed a SETL interpreter inside Netscape Navigator, the leading browser at the time, as a way to develop GUIs for SETL programs quickly, and to replace Javascript with SETL as scripting language for HTML pages. The project eventually succeeded, and we were able to run SETL code inside the browser, call Java and Javascript from SETL and vice-versa. But a sizable amount of code was needed to link together the compiler and interpreter, which were designed originally as stand-alone programs.

The situation is dramatically different if one uses the ActiveX Scripting technology of Microsoft's Windows operating system. The development of this technology started at the end of 1995 and the technology was first released in July 1996 with Internet Explorer 3.0. The idea behind ActiveX scripting is very simple: the interface between applications (called by Microsoft Scripting Hosts) and Scripting Engines (in Microsoft jargon) or Macro languages is carefully standardized. Then by implementing only the appropriate Host interfaces in applications we are able to support a wide variety of different languages within a single application.

These powerful interfacing conventions allow even a version of Internet Explorer, such as 3.0, to handle scripts written in languages that did not even exist in 1996. Moreover Microsoft has used Scripting Engines written following these conventions in many host applications. For example, the same engines are used in Microsoft's Excel, Word, Powerpoint, Active Server Pages and in its browsers. Future improvements in these Engines will automatically benefit all these Hosts.

ActiveX Scripting is based on Microsoft's Component Object Model (COM), and relies heavily on the OLE Automation Interface (IDispatch) to manipulate Host objects. It defines two "interfaces" that the Host and Engine must use to be able to communicate with each other. In the following paragraphs we summarize the COM conventions relevant to Scripting engine integration in broad outline.

2.4.1. COM and The IDispatch interface

COM is a software architecture that allows applications to be built from binary components. In COM, software components communicate by means of *interfaces*. An interface is actually a pointer to a table of function addresses, organized following a layout very much like that used by a C++ compiler to store VTABLEs in physical memory. Interfaces are identified by UUIDs (Unique identifiers).

Since COM is a binary (bit-level) convention, client and server objects using COM must agree exactly on the physical layout of the interfaces. This requirement yields efficiency advantages if the services offered by an object are known in advance and a custom interface can be designed. In the case of scripting languages, however, a more flexible mechanism is required when the communication of data and commands between components must be handled more dynamically. This is the basic functionality supported by the IDispatch interface.

IDispatch provides a ‘GetIDsOfNames’ function that maps method or property names to integer DISPID (Dispatch Identifiers). Once such a method identifier is obtained, the Invoke function of IDispatch can be used to call the method associated with it. The arguments of such calls are passed in so-called VARIANTS. These structures are basically “C” unions of different data types, each identified by a type tag. By examining this tag, interpreters (and the COM infrastructure) can check types and perform simple type conversions.

2.4.2. *Windows Scripting Architecture*

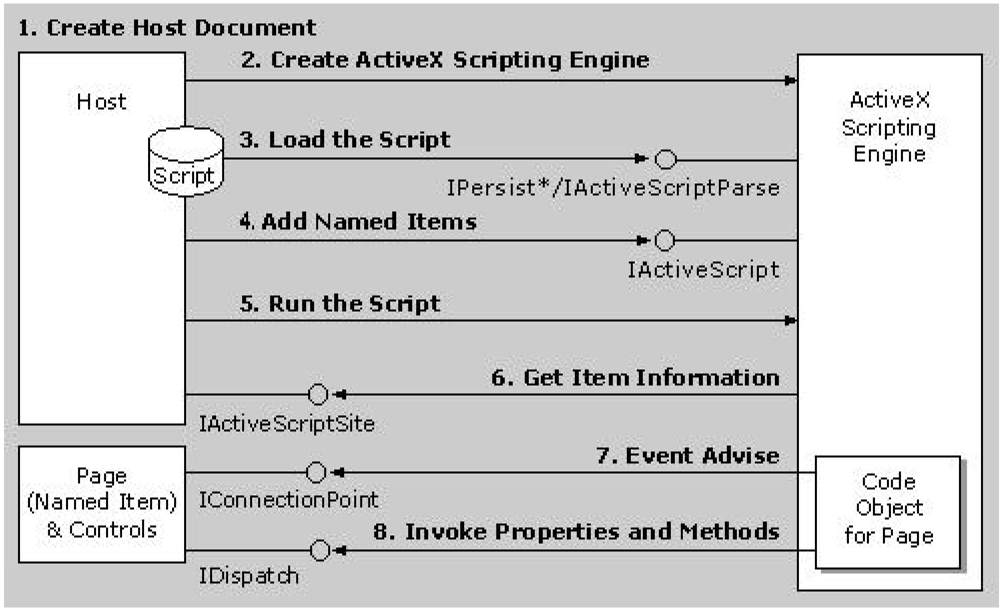


Figure 7. Windows Scripting Architecture

Many applications can be improved by allowing a savvy user (and not the original programmer) to customize and control them by using a scripting or macro language. The ActiveX Scripting technology carefully defines a few key COM interfaces (IactiveScriptParse and IactiveScriptSite) that can be used for the interaction between many Host applications and each of the scripting engines which it supports.

ActiveX Scripting does not specify how the desired scripting engine is selected by the Host, but it assumes that the Host has an unambiguous way to determine the correct engine to create (step 2 in Figure 7 above).

Script 'Loading' is then handled by IactiveScriptParse (step 3). 'Loading' is a slightly more general operation than simple parsing. Scripting Engines will often have the capability to save their state using the standard 'IPersist' interfaces defined by COM. This allows the host to call the script engine's IPersist*::Load method to feed it a script state that was saved at an earlier time (If no such saved state is provided, the host uses either the IPersist*::InitNew or IActiveScriptParse::InitNew method to create a null script). A host that maintains a script in text form can use IActiveScriptParse::ParseScriptText to supply a scripting engine with the text of the script to parse, after calling IActiveScriptParse::InitNew to initialize the environment required for parsing.

Applications are seen by scripting engines as ‘Documents’. Each such Document describes a collection of objects that the Engine can customize or control. For example, the Windows Scripting Host tool describes a local filesystem in such a Document, thereby allowing the Scripting Engines in communication with the host to operate on its files and directories.

‘Named Items’ are the top-level items of such an object hierarchy. To add a new Named Item, the host calls the IActiveScript::AddNamedItem method (step 4). (This step is not necessary if top-level named items are already part of the persistent state of the script loaded in step 3 (see below).) A host does not use IActiveScript::AddNamedItem to add sublevel named items (such as controls on an HTML page); rather, each engine obtains sublevel items from top-level items indirectly by using other ActiveX interfaces, such as the host's ITypeInfo and IDispatch (step 6).

Whenever a scripting engine needs to associate a symbol it finds in a script with a top-level item (which was previously added to the engine environment), it calls the IActiveScriptSite::GetItemInfo method, which returns information about the given item. This operation can be used repeatedly by the scripting engines as new items defined externally are encountered during parsing. As the script runs, properties and methods of named objects can be accessed and/or invoked through IDispatch::Invoke or other standard OLE binding mechanisms (step 8 in the figure above).

Finally, mechanisms are defined in IactiveScript to run scripts and to allow Host events to trigger script callbacks.

1. The host causes the appropriate scripting engine to start executing a script by setting the SCRIPTSTATE_CONNECTED flag via the IActiveScript::SetScriptState method (see below).

2. Before starting the actual script, the scripting engine connects to the events of all the relevant objects through the IConnectionPoint interface (steps 7 and 8 in the figure above).

At any given time, a Windows Scripting engine can be in one of several states.

- *UNITIALIZED*. The scripting engine has just been created and has not been loaded with a script yet.

- *INITIALIZED*. The scripting engine state has been initialized (using IActiveScriptParse::InitNew) or a prior state has been restored (scripting engine state is restored using either the IPersist interface or IPersist*::InitNew). The script code passed using the IActiveScriptParse::ParseScriptText method is queued by the engine but not executed yet.

- *STARTED*. The transition from the initialized state to the started state causes the engine to execute any code that was queued in the initialized state. The engine can execute code while in the started state, but it is not connected to any events added through the IActiveScript::AddNamedItem method. The host can execute code by calling the IDispatch interface obtained from the IActiveScript::GetScriptDispatch method, or by calling IActiveScriptParse::ParseScriptText.

- *CONNECTED*. The script is loaded and connected to events emanating from host objects. If this is a transition from the initialized state, the scripting engine should pass through the started state, performing the necessary actions, before entering the connected state and connecting to events.

- *DISCONNECTED*. A script has been loaded and has a run-time state, but is temporarily disconnected from sinking events from host objects.

- *CLOSED*. The script has been closed. The scripting engine halts and returns errors for most methods.

In fact the Microsoft Engines are slightly more complicated, as they also support an extended version of the IactiveScript Interface which supports debugging. One can use this interface for single step execution, variable inspection and so on. But the VALIS system described in this thesis does not use the Debugging interfaces yet. This interesting Microsoft feature indicates that even the interaction between a debugging tool and a scripting language can be standardized.

2.5 VALIS Scripting Architecture

The VALIS CScript class is the main class for creating a VALIS application as an ActiveX scripting host. It implements the IActiveScriptSite, IActiveScriptSiteWindow and IDispatch interfaces. The first two are needed to behave as a scripting host; the third is a custom IDispatch interface.

When we discussed the “Valis” object above, we noted that the scripting engines do not need to know a priori about the methods available in this object. When an unknown method of the “Valis” object is encountered, the scripting engines will query the IDispatch interface of the ‘Valis’ object requesting information about the method. By implementing a custom IDispatch interface, we take advantage of this to implement inter-language calls.

When the GetIDsOfNames method of this custom interface is called to request information about a method, it must look up the name and generate the corresponding DISPID. (If the method is not found, an error will be generated.) This function first checks to see if the method is part of the CValis predefined IDispatch interface. For example, this object exports the predefined methods CreateObject, Print, etc., as listed in Table 1 above. A map between method names and method location is accessed and a DISPID is generated. If the method is not predefined, the function will instead search in the namespaces of all the scripting engines, using their IDispatch interfaces.

A function whose name is ‘x’ defined in Jscript could have DISPID ‘i’ in the Jscript engine. When Valis.x is called from another engine, for example Perl, a new DISPID ‘j’ will be generated by the custom IDispatch in VALIS and the mapping will associate DISPID ‘j’ for PERL with DISPID ‘i’ in Jscript.

The lookup is only done once because our custom IDispatch caches them. Finally, when a scripting engine calls the method, using the IDispatch::Invoke function call, our custom IDispatch::Invoke is called, and this will just route the call to the correct IDispatch::Invoke method of the scripting engine where the name had been found.

2.6 Summary

In this chapter we examined how different scripting languages deal with native methods. A fair amount of glue code is required to integrate simple libraries in different scripting languages. Using tools like SWIG is not always a satisfactory solution. Embedding scripting languages in applications is even more problematic, and it sometimes requires major rewriting of their interpreters. Use of the carefully crafted interfaces provided in Microsoft’s ActiveX scripting is potentially a better way to interface scripting languages and their environments and has major advantages compared with an ad-hoc solution. We have exploited these interfaces in VALIS to implement language-neutral components, inter-language calls, and hence high level code reuse.

Chapter 3

Free Format Databases

Powerful and flexible data storage systems are another basic building block needed to prototype bioinformatics applications effectively.

In bioinformatics, as in many other scientific computing fields, one must deal with large amounts of unstructured or semistructured data. The common approach to this has been to store most of the data in flat files, and metadata in relational database systems. Sometimes the bulk of the data is stored in BLOBs (Binary Large Objects) within relational databases. But flat files and BLOBs cannot be searched effectively using the facilities available in typical relational databases.

During our revision of the SETL programming language [31] we added a powerful database facility, which can deal effectively with unstructured data and potentially very large objects, like DNA sequences, microarray data and annotations, and which promises to be very effective especially at the prototyping stage.

SETL now includes a powerful facility for database design and implementation. (Associated with this is an extended facility for prototyping and implementing complex commercial applications, based on the notion of hibernating transactions, which can be resident for long periods of time in the database itself.) These facilities are fully transactional, in the sense explained below, and also facilitate prototyping of and implementation of large distributed databases. We decided to adopt SETL's database system as the preferred data storage facility in VALIS.

SETL databases provide an alternative to standard SQL databases. Each SETL database object is a potentially very large (up to hundreds of gigabytes) abstract object stored on disk. Each such database object resembles a map (function) from a set of system generated 4- or 5-byte record identifiers to the records which the database stores. Each of such records is a fully general object of the SETL language, which is itself not excessively large (e.g. no more than a few dozen megabytes.) The database records are systematically indexed for efficient access, using the technique described below in abstract terms. Once a record of a SETL database has been accessed, all the operations of the SETL language can be applied to it. Conversely, any standard SETL object can be stored as a database record.

The operations specific to SETL databases (other than standard SETL operations applicable to individual records) are as follows:

x := db(id);	Find a database record given its identifier
db(id) := x;	Store x as a database record associating it with the specified identifier
db with x	Create a new record and put x into it. A pair of the form [modified_db,id_of_new_record] is returned.

db.to_string;	Database-associated mapping of records to reduced string variants ('summary strings') used for indexing; see below
db.contains(wd)	Creates and returns a database iterator object, which iterates over all records whose summary string (see section 3.2) contains the given word

Table 3. Free format database operations

Iterations 'x in iter_obj' over the iterator objects return the ordered sequence of keys of all records containing the specified word (in a standard key-collating order). These iteration objects support the following operations:

#iter_obj	Number of keys in an iteration group
arb(iter_obj)	First item in the iteration group, or OM (the SETL NULL value) if none
iter_obj + iter_obj	Iterate over union.
iter_obj * iter_obj	Iterate over intersection
iter_obj - iter_obj	Iterate over difference

Table 4. Iterator object operations

Internally, the database 'records' are represented by long string sections, separated by special marks representing each record's identifier. The detailed layout of the 'id-and-record' file used to store this information is described later in this chapter.

3.1 Comparison of SETL databases with standard relational databases

The database approach described in this chapter may have significant advantages over the currently pervasive SQL view of databases. To begin to see how these two approaches compare, note that the SQL database records are ordinarily shown with multiple fields, as in the following example.

Name	Age	Department	Salary	Employed	Position.
John Peter Doe	43	063	21000	6/7/79	Clerk
Mary M. Smith	25	063	45000	9/1/91	Mgr.

In the SETL version of simple databases of this kind, indices associated with the SQL records are handled as special ‘index’ strings (these are used only for searching in the SETL version of the same database). These index strings can have forms like

"John Peter Doe 43 D:063 $21000 6/7/79 Clerk"

"Mary Smith 25 D:063 $45000 9/1/91 Mgr."

Note that these strings can contain words not present in the original records.

For example, the department column is indexed by the string ‘D:’ followed by the department number. The database iteration operations listed above then allow the crucial core of database searches, e.g. a search for the manager of the employee John Doe, to be written in SETL as follows:

```
{name(y): x in contains("Doe"), y in contains(department(x))

 | position(y) = "Mgr." & "John" in words_of(string_of(x))}.
```

This expression returns all the managers in any department containing an employee called John Doe. (Along with whatever 'accidental' data may occasionally result from unexpected combinations of John and Doe, as in a conceivable department that manufactured Doe Skin gloves and had an employee named Wallace John, or John Johnson. But this sort of thing should be quite rare, and can in any case be detected by forming and filtering or displaying the set of pairs

```
{[x,y]: x in contains("Doe"), y in contains(department(x)) |

position(y) = "Mgr." & "John" in words_of(string_of(x))}.
```

in an appropriate format).

This example shows the potential advantage of the approach proposed: (a) it separates the optimization of database queries cleanly from their abstract definition; (b) no additional description of the fields to be indexed is required; (c) arbitrary strings can appear in fields, and the system is prepared to search using concordances of the words in these fields, which is often an effective search technique. (E.g. we can just as well find the manager of "Peter Doe", or "J. Peter Doe", a task which might well vex an SQL database.)

3.2 Indexing in SETL databases

As illustrated by the simple example given above, each SETL database has an administrator-defined 'to_string(record)' method which applies to each of its records and which yields a 'record summary string' defining those aspects of the record which are to be indexed to improve the efficiency of searches over the database. By cutting these summary strings into words (simply by breaking these strings at all included whitespaces) the system generates an associated internal word_list(record). These word lists are automatically used to build a 'word index' associated with the database. Iterations over the db.contains(wd) iterator objects described above exploit this index.

3.3 Transaction handling in SETL databases

SETL databases, like all other SETL objects, have value semantics. For example, if db is a database object, then the sequence of statements

```
db2 := db;

db := db with new_record;
```

changes db by adding a new record, but leaves db2 unchanged since no new assignment has been made to it. The SETL database implementation creates the required 'logical copies' in an efficient way, whose cost is only weakly dependent of the size of the database objects involved.

This gives an easy way of obtaining transaction-like effects. To begin a transaction, which may have to be aborted, simply write

```
db_copy := db;
```

Then edit db as desired. To 'commit' the transaction, simply erase the copy by writing db_copy := OM. To back out of the transaction instead, simply write db := db_copy; this will restore db to its original state.

3.4 Value-semantics of SETL Databases

The following short program illustrates several other significant properties of SETL databases, specifically:

a) As already mentioned, they obey value-semantics rather than pointer semantics rules;

b) Records in these databases need not have any fixed structure;

c) These databases behave like ordinary SETL objects, i.e. can be put into tuples and/or sets, can be passed as parameters to subroutines;etc.

The program is:

```
program test;                  -- test program for SETL database class
use SETL_database;                      -- use the free format database class
use string_utility_pak;                 -- we need breakup & join functions

 -- create an empty database
db := SETL_database(tuple_to_stg,”spam.db");
```

```
 -- Add first protein sequence
 [db,first_protein]:= db **with** ["CAE85316",
            "manklflvcatfalcflltnasiyrtvvefdeddasnpmgprqkcqkefqqracqk”,
            "Arabidopsis thaliana"];

-- Add second protein sequence
[db,second_protein]:=db with ["CAC80708",
          "anyggdkqygretrqtgdyenpihstggqyeqdvrqtdeygnpvrrtdey",” Helianthus niveus”];

-- Add third protein sequence
[db,third_protein]:=db with ["AAL69565",
          "msccngkcgcgmypdvevsattvmivdgvapkqmfaegsegsfvaeg",
           ”Helianthus tuberosus“];

db_save := db;  -- save the original database

db(first_protein) with :=
 "Eukaryota; Viridiplantae; Streptophyta; Embryophyta;”+
 “Tracheophyta; Spermatophyta; Magnoliophyta; eudicotyledons;”+
 “core eudicots; rosids; eurosids II; Brassicales; “+
 “Brassicaceae; Arabidopsis";

db(second_protein) with :=
          "Eukaryota; Viridiplantae; Streptophyta; Embryophyta; “+
 “Tracheophyta;Spermatophyta; Magnoliophyta; eudicotyledons;”+
 “core eudicots;asterids; campanulids; Asterales; Asteraceae;”+
 “Asteroideae;Heliantheae; Helianthus";

-- Add ‘L:’ prefix to search the ‘locus’ tuple element
print("Edited Database");
for key in
  (db.contains("L:CAE85316")+ db.contains("L:CAC80708") +
   db.contains("L:AAL69565")) loop
              print(db(key));
end loop;

print("Saved Database");
for key in
  (db_save.contains("L:CAE85316")+
   db_save.contains("L:CAC80708") +
   db_save.contains("L:AAL69565")) loop
              print(db_save(key));
end loop;

print("\nComparison of records in the two databases");
print("db_save(first_protein): ",db_save(first_protein));
print("db(first_protein): ",db(first_protein));

-- make a tuple of databases
database_list := [db,db_save];
```

```
print("\nQuery from a list of databases");
for database in database_list loop
            for key in database.contains("K:rosids") loop
                        print(database(key));
            end loop;
end loop;
-- convert [locus,seq,organism,keywords] tuple
-- to index string. Words appearing in the 4th element of the
-- tuple are prefixed with 'K:'. Similar prefixes are added to
-- the other elements of the tuple.
procedure tuple_to_stg(p);
   [locus,-,org,kw] := p;
   x:=" L:"+locus+" "+
         join([" O:"+k:k in breakup(org?"",";.,/* ")],"")+
         join([" K:"+k:k in breakup(kw?"",";.,/* ")],"");
   return x;

end tuple_to_stg;
end test;
```

The preceding program produces the following output.

```
Edited Database
["CAE85316", "manklflvcatfalcflltnasiyrtvvefdeddasnpmgprqkcqkefqqracqk",
"Arabidopsis thaliana", "Eukaryota; Viridiplantae; Streptophyta;
Embryophyta;Tracheophyta; Spermatophyta; Magnoliophyta; eudicotyledons;core
eudicots; rosids; eurosids II; Brassicales; Brassicaceae; Arabidopsis"]
["CAC80708", "anyggdkqygretrqtgdyenpihstggqyeqdvrqtdeygnpvrrtdey", " Helianthus
niveus", "Eukaryota; Viridiplantae; Streptophyta; Embryophyta;
Tracheophyta;Spermatophyta; Magnoliophyta; eudicotyledons;core eudicots;asterids;
campanulids; Asterales; Asteraceae;Asteroideae;Heliantheae; Helianthus"]
["AAL69565", "msccngkcgcgmypdvevsattvmivdgvapkqmfaegsegsfvaeg", "Helianthus
tuberosus"]
Saved Database
["CAE85316", "manklflvcatfalcflltnasiyrtvvefdeddasnpmgprqkcqkefqqracqk",
"Arabidopsis thaliana"]
["CAC80708", "anyggdkqygretrqtgdyenpihstggqyeqdvrqtdeygnpvrrtdey", " Helianthus
niveus"]
["AAL69565", "msccngkcgcgmypdvevsattvmivdgvapkqmfaegsegsfvaeg", "Helianthus
tuberosus"]

Comparison of records in the two databases
db_save(first_protein): ["CAE85316",
"manklflvcatfalcflltnasiyrtvvefdeddasnpmgprqkcqkefqqracqk", "Arabidopsis thaliana"]
db(first_protein): ["CAE85316",
"manklflvcatfalcflltnasiyrtvvefdeddasnpmgprqkcqkefqqracqk", "Arabidopsis thaliana",
"Eukaryota; Viridiplantae; Streptophyta; Embryophyta;Tracheophyta; Spermatophyta;
Magnoliophyta; eudicotyledons;core eudicots; rosids; eurosids II; Brassicales;
Brassicaceae; Arabidopsis"]

Query from a list of databases
["CAE85316", "manklflvcatfalcflltnasiyrtvvefdeddasnpmgprqkcqkefqqracqk",
"Arabidopsis thaliana", "Eukaryota; Viridiplantae; Streptophyta;
Embryophyta;Tracheophyta; Spermatophyta; Magnoliophyta; eudicotyledons;core
eudicots; rosids; eurosids II; Brassicales; Brassicaceae; Arabidopsis"]
```

The program seen above creates a database and inserts three records, each a simple pair, into it. The database is then copied to a second variable 'db_save', and the original database is modified by editing its first two records, to which additional string components are freely added. (The simplicity of this operation, which is a bit clumsy in a standard SQL database since it changes the number of SQL 'columns', illustrates one of the advantages of SETL databases: since records in them are entirely free-form, additional elements of any kind can be added to records in a completely dynamic way.) The edited records are retrieved from the edited database using an iterator, but then also from the saved database, which is seen to contain the original records, unchanged. This illustrates a second advantage of SETL databases, their value semantics, on which a 'transactional' capability can be built in the manner noted above. Finally we include the databases in a tuple, from which the databases are then recovered and used. This example illustrates the fact that databases behave like ordinary SETL objects, and so can be put into tuples and/or sets, passed as parameters to subroutines, etc.

3.5 Implementing SETL Databases

The underlying structure used to realize the large objects supported by SETL (and most other databases) is the B-tree. This famous structure can be thought of as a way of maintaining large tuples in a form that allows fast execution of all the standard tuple operations, i.e., component access, component change, component insertion, slicing, and slice assignment. B-trees can be thought of as arising by recursive elaboration of the simple idea of dividing very long tuples T (e.g. tuples of length 1 million) into sections of some smaller length, e.g., length approximately 1000. If this recursive division is done, then a tuple of length 1 million will be represented by a tuple R of tuples, each of length roughly 1000, the whole list R also being of length approximately 1000. To make an insertion into the middle of such a structure (regarded as a representation of the large tuple T that would be obtained by concatenating all the components of R) is then much faster than it would be in a flat representation of T. Insertion in the middle of a flat representation might require all following components to be moved, so if T is of length one million, 500,000 of its components might have to be moved. But in the alternate representation suggested above the insertion is made in a sub-tuple of R having length roughly 1000 that is easily located, and so should require no more than 500 components to be moved. This gives a speedup that can be as high as 1000.

As already stated, the B-tree structure arises by recursive elaboration of the idea just described. Rather than arbitrarily dividing long tuples like T into some fixed number of pieces in the two-level way sketched above, we divide it into a tuple of tuples of tuples ..., down to as many levels as needed, limiting each of the tuples ('nodes') that appear at each level of this nested tree structure to some convenient maximum length. It turns out to be most convenient to insist that (with the one exception of the tree root) each 'node', (i.e., tuple) occurring in the data structure have a length lying between some integer n and (2n – 1). This ensures that insertions remain very fast while at the same time limiting the number of tree levels that must be searched to locate a given component of the large tuple T that the tree structure R represents. Specifically, this number of levels can be no more than $\log_n L$ where L is the length of the tuple represented.

The process of insertion of a new component into the large tuple represented by the recursive tree structure R will force insertion of a component at the bottom tree level, and this may cause the tree node into which the insertion is made to overflow the maximum length (2n – 1) to which it is supposed to be limited. But when this happens we can simply divide that node into two nodes of length n, inserting one of these at the next level up in the tree. Even though the process of recursive insertion thereby triggered may propagate all the way up the tree to its root, it remains orderly at each recursive level, as the more detailed code shown below makes plain.

3.5.1. Value semantics for large B-trees

For our intended database application, the crucial advantage of the recursive B-tree structure is that it allows value semantics to be implemented efficiently for large tuples. If, for example, we copy a tuple t_1 into a tuple t_2 by writing $t_2 := t_1$, and then change a designated component of t_1 by writing $t_1(i) := x$, we only need to copy and modify those tree nodes of the changed tree t_1 which lie on the tree path down to the point at which the modification occurs. All other tree nodes can simply be shared between the two trees t_1 and t_2. Thus the amount of node copying and modification required by the rules of value semantics is the product of O(n log L), where L is the length of the tuple represented, and (2n – 1) is the maximum allowed node size.

The following generic B-tree code reflects these considerations, some stated explicitly and others implicitly. The database native classes, described at the end of this chapter, have been derived from a prototype SETL class 'Btup', containing two vectors 'cum' and 'tup'.

```
class btup;    -- B-tree class
 class var debug := false;

 procedure create();  -- create blank btup
 procedure set(t);  -- set btup object from standard SETL tuple

 procedure check_consistency();  -- check consistency of tree

end btup;

class body btup;    -- B-tree class
 const minw := 5, maxw := 2 * minw - 1;
        -- minimum number of descendants, except at top

 var h := 1,cum := [],tup := [];   -- height, cumulant tuple, top-level tuple;
  -- Note that B-trees have value semantics, but with sharing of sub-trees.
  -- The cumulant records the number of descendants of each node,
  -- plus those of all siblings to its left.

 var iter_ptr,compno;      -- iteration pointer and count

 procedure create();  -- create empty B-tree top from tuple
  iter_ptr := newat(); compno := newat();
       -- set up iteration stack pointer and compno pointer
  return self;
 end create;
```

The 'tup' instance variable held in each tree node stores the list of child nodes of that node, or, in the case of nodes of height 1, stores the components of the represented tuple that come under it. The j'th component of the 'cum' instance variable, whose value is a tuple identical in length to 'tup', stores the total number of leaves that come under any of the nodes tup(1)...tup(j). These 'cum' vectors allow a fast binary-search like process to search down the tree for a twig of given index i within the large tuple T that the tree represents . This recursive search structure is seen in the procedure self(i) that appears in the following code, which is programmed recursively, as a single line. The same is true for the component change operation seen in the procedure self(i) := x that follows. It is worth thinking through the action of this latter routine in detail to understand how much node copying it implies when a component is changed, and where these copies occur in the code.

```
procedure self(i);  -- component extraction;
   -- descend iteratively to desired leaf using cumulant to find child containing it
  return if h = 1 then tup(i) elseif i <= cum(1) then tup(1)(i)
    elseif exists c = cum(j) | i <= c then tup(j)(i - cum(j - 1))
     else OM end if;
 end;

 procedure self(i) := x;  -- component change
   -- descend iteratively to desired leaf using cumulant to find child containing it
   -- then change the value at this node. Note that the cumulant need not be changed
   -- in this case, since here it represents the number of leaves beneath each node,
   -- which this operation does not change.
  if h = 1 then tup(i) := x; elseif i <= cum(1) then tup(1)(i) := x;
    elseif exists c = cum(j) | i <= c then tup(c)(i - cum(j - 1)) := x;
     else abort("index " + str(i) + " is out of range"); end if;
 end;
```

3.5.2. Single and multiple cumulants

The 'cum' vector illustrates the important idea of storing 'cumulant' quantities, derived by some addition-like operation from the children of a node, at each of the nodes of a B-tree. This idea generalizes readily. Multiple such 'cumulants' can be stored in each node. Each such cumulant supports a corresponding form of fast binary search in the tree. This idea is explored more fully in the next section.

Since all of the twigs in a B-tree are required to have the same height, that is, to lie at the same distance from the tree root, it follows that every tree node lies at the same distance from all of the twigs that come under it. This height is third item of value data stored in each node.

With respect to all of these node data items, particularly 'cum' and 'tup', node objects have value semantics. In other words, change of any component of these two tuple-valued instance variables causes a copy of the changed object to be made in such a way as to guarantee that no other object referenced directly or indirectly by an independent variable is affected. It should be clear that this copy mechanism implies that the large tuples represented by our system of tree nodes also have value semantics.

3.5.3. Constructing SETL Databases out of B-trees

The B-tree object class 'btup' implemented by the SETL class code just described and further detailed in section 3.8 below is logically ancestral to most of the other codes needed to realize the database facility at which we aim. Thus, most of the actual codes used are derived from this initial prototype by working additional functionality and feature into it, without really changing its basic structure very much. We shall see that such 'conservative reworking' allows us to define not only various useful kinds of large string and large tuple objects, but also all of the large maps used for indexing SETL databases in the manner described at the start of this Chapter. Another essential feature, which can be worked into the same basic B-tree structure is the reference counting mechanism used to give B-trees the value semantics whose importance we have already explained. The fact that this latter mechanism allows us to implement database modifications using only very limited numbers of tree node changes is essential both to the efficiency of the whole database system, and also to the crash-recovery capabilities.

We begin our account of this chain of generalizations by explaining one of the simplest of them, the use of additional cumulants to realize other important data structures by generalized B-trees.

3.5.4. B-trees Incorporating More General Cumulants.

The B-tree data structure seen in the preceding code can be adapted to realize other data objects than tuples. For example, we can also represent very large strings, e.g., strings of several gigabytes in length, by similar B-tree structures, simply by replacing the initial cumulant 'cum' stored in the tree nodes by one which cumulates number of characters rather than number of nodes. The idea here is to store very large strings by breaking them up into pieces, say of a few hundred characters each, which are referenced by an overlying B-tree (of a few million nodes). In this B-tree the cumulant stored represents the number of characters in all the leaves, which fall below some particular tree node and its left siblings. A variation on this idea would be to maintain strings divided into words by the occurrence of non-alphabetic characters, and into lines by the occurrence of end-of-line characters, in such a way as to allow very rapid access either to a particular character position, or to a particular word by its numerical position in the sequence of all words, or to a particular line by line number. For this, we could simply maintain three auxiliary cumulants, one for total number of characters, the second for total number of word breaks, the third for total number of line-end characters. It should be clear that the code seen above can be adapted to handle all such cases, and that it can retain all the advantages seen above. That is, all supported accesses and edit operations can be executed in time proportional to the logarithm of data object size, and value semantics, with the advantages that we have noted above, can be maintained in the presence of active edit operations.

The cumulants described in the preceding section all are count-like, in that they are calculated for parent nodes by summing the corresponding values for their children. However, the same cumulant technique will work even if operation used to calculate the cumulant of a parent node from those of its children is an associative operation different from simple summation. Only associativity, not even commutativity, is required. This observation opens many possibilities. For example, the maximum and minimum operations in any ordered set are associative. Hence, if the leaves of a B-tree store elements (such as strings or integers) belonging to such a set, we can keep a cumulant derived from the maximum or minimum of children at each of the B-tree's nodes, and keep these cumulants updated as the tree is edited, in much the way seen in the code displayed above, without losing the logarithmic efficiency of the operations which this code implements. Suppose then that we aim to keep the tree leaves in sorted order. The availability of a cumulant representing the maximum of all the leaves coming under or to the left of any of our tree nodes makes it possible to find the largest leaf smaller than a new leaf to be inserted in logarithmic time. Hence we can insert this new leaf at its proper position in logarithmic time, thus always keeping the sequence of leaves in their desired order.

Here is another, more sophisticated but less generally useful variant of this same idea. Suppose that we wish to store very long parenthesized strings in a manner allowing the closing parenthesis matching a given open parenthesis to be located rapidly, even though these two parentheses may be separated by millions of characters and many nested pairs of matching parentheses. This can be done as follows. In each node we keep two cumulants, one 'free_opens_in' representing the total number of open-parentheses coming under the node which are not matched by any close-parenthesis coming to their right, the second 'free_closes_in' representing the total number of close-parentheses coming under the node which are not matched by any open-parenthesis lying to their left. To combine these quantities as calculated for two strings when the two strings are concatenated we can use the following formula.

```
if (foi1 := free_opens_in(stg1)) >= (fci2 :=
                free_closes_in(stg2)) then
        free_opens_in(stg1 + stg2) := foi1 - fci2 +
                free_opens_in(stg2);
        free_closes_in(stg1 + stg2) := free_closes_in(stg1);
else
        free_opens_in(stg1 + stg2) := free_opens_in(stg2);
        free_closes_in(stg1 + stg2) := free_closes_in(stg2) +
                (fci2 - foi1);
end if;
```

This algorithm can be thought of as implementing an operation on pairs [num_free_opens_in,num_free_closes_in] which, because of its relationship to the obviously associative string-concatenation operation, is also associative. Hence these quantities can be kept together as cumulants in the nodes of a B-tree representing strings. It is not hard to see that, with these quantities available, the parenthesis which matches a give open- or close-parenthesis can be found by a logarithmically efficient recursive search.

3.6 Low level implementation of the SETL database facility

A prototype of the SETL database system described above has been implemented as a set of high performance specialized native “C” B-tree packages for most of the structures needed to support the top level code. The SETL specification of these native packages is available in the ‘setldb.stl’ file of the SETL distribution [12]. But before we can discuss the low-level architecture of the system, we need to examine what is involved in this transformation.

For the reasons outlined in the first section of this chapter, we want to manage large objects represented by B-trees and stored on disk, be these simple strings or full databases, and we want them to have value semantics. The initial in_RAM prototypes of these objects, realized by the codes described earlier in this chapter, are implemented entirely by SETL objects, and so inherit their value semantics from that of their underlying SETL objects. In this setting the SETL interpreter supplies the implicit reference count management needed to support value semantics. For objects stored on disk this is no longer the case, and so the reference count management needed to give these objects their desired value semantics must be programmed explicitly.

3.6.1. Reference-count management.

We separate reference-count management into two parts:

- A first or 'bottom' part having to do with the counting references to disk records as these are set up and removed by the codes internal to the packages of routines (those described in the following section) which implement operations on disk-stored strings (and other similar objects) represented by B-trees, and

- A second or 'top' part relating to the handling of reference counts in procedures, which operate more at the application level, by using these lower-level routines. Note that routines of this second class are unaware of the very existence of internal B-tree nodes, and so never reference them directly. However, they do reference the root nodes of such objects, and so may modify the reference counts of such root nodes.

Reference counting can be implemented by making each disk record (for an object of interest) store a field that keeps track of the total current number of references to the record, either from variables in a program manipulating such records, or from other records. First consider reference-count management for (the limited family of) 'bottom' level routines, i.e. those that manipulate internal tree nodes. Whenever such a record, referenced by a variable obj, is about to be modified (e.g. by an operation like obj.set_cum(i,x) or obj.set_tup(i,x) which modifies a field within it), we check the refcount of the record. If this is 1, we can be sure that 'obj' is the only reference to the object, and so we can simply proceed with the change. However, if the refcount is > 1, then there are other references to the same record, and the rules of value semantics forbid the edit about to be performed from affecting the data value seen by these other references. To avoid this conflict, we first copy the record that we are about to edit (thereby replacing it with a brand new record whose reference count is certainly 1), and then modify the new record. When this copy operation is done the number of references to the old copy of the record diminishes by 1. The check-and-replace operation needed for this process can be thought of as an assignment

```
obj := obj.maycopy();
```

inserted at the start of edit operations like obj.set_cum(i,x). The operation 'maycopy' just returns 'obj' if its count is 1, but otherwise returns a fresh copy and decrements the refcount of the old copy by 1.

Creation of the new record copy increments the number of references to all of the records to which the copied record refers. The operation 'maycopy' can handle this supplementary action also.

This is all that we need do in the case of edit operations like obj.set_cum(i,x) which simply modify a numerical value. An additional step is required for operations like 'obj.set_tup(i,x)' which modifies a value which points to another record. This step must decrement the number of references to the object 'obj.tup(i)' which was originally referenced by the field being modified (unless this field was initially unused). In addition, this step must also increment the number of references to the object x (unless x is undefined). This action can be thought of as an operation

note_replace_ref(obj.tup(i),x);

inserted at the start of the 'obj.set_tup(i,x)' procedure but not at the start of operations like 'obj.set_cum(i,x)'. Similarly any statement x := obj.tup(i); decrements the number of references to the record value of x (if any) and increments the number of references to obj.tup(i). Hence a call note_replace_ref(x,obj.tup(i)) should be attached to each such operation.

To create the new records required for some of the operations just described, we need a way of allocating the disk pages, which will hold them. If disk space is not to be consumed progressively until no more remains, pages allocated to records that are no longer in use must be reclaimed. A record falls out of use whenever its reference count falls to zero, as may, for example, happen to the record referenced by 'obj.tup(i)' when we execute the operation 'obj.set_tup(i,x)'. The simplest (but not the most efficient) way of managing page reclamation is to maintain a free pages list, to the front of which records are linked when they are freed, and from which recycled pages can be recovered by delinking them as needed. This is a simple one-way list. Records put on the free list can retain any references to whatever other records they initially contain. However, when a record is delinked from the free-list in preparation for re-use, it implicitly loses all these references, so each of the other records that it references must have their refcount decremented by 1, perhaps causing them to be added to the free-list themselves. (Alternatively, all the children of a record being freed can be processed immediately to decrement their reference counts and free them recursively if these counts fall to zero.)

Whenever a simple assignment x := y; is used to replace the value of one variable by that of another, the record referenced by x (if any) loses one reference, and that referenced by y (if any) gains one. These actions can be implemented by a call adjust_counts(x, y);

To complete our account of the bottom-level reference-count manipulations needed to implement value semantics for disk-stored structures, it only remains to consider the treatment of procedure calls and returns. A call f(expn_1,...,expn_n) can be viewed as a group of assignments

$$\text{param}_1 := \text{expn}_1; \ldots\ \text{param}_n := \text{expn}_n$$

of the call arguments to the parameter values of f, which are originally **OM**. Hence calls simply increment the reference-count values of those of their arguments which have references as values. This set of assignments is followed by a jump, which has no effect on reference counts. All variables other than the parameter variables of called procedures should be given the value **OM**; recursive calls should be considered to introduce wholly new sets of parameter and internal variables.

The operation

return expn;

can be viewed as an assignment

retvar := expn;

of 'expn' to the return variable of the function call, followed by a jump which has no effect on reference counts. The return also deallocates all the internal and parameter variables of the procedure being returned from. This implicitly sets all these values to **OM**, and so should decrement the reference count of every variable whose immediate pre-**return** value is a record. This can be done by inserting a call

cleanup([v_1,...,v_m]);

immediately before each **return** statement, which should however first be rewritten in the form

retval := expn; **return** retval;

The 'retval := expn;' statement will have its normal effect on reference counts; the '**return** retval;' statement which follows it will have none. Function calls, which ignore their returned values, should be treated as if they read

retval := f($expn_1$,...,$expn_n$); retval := OM;

The 'cleanup' routine seen above simply examines each component of its argument tuple and decrements the number of references of each such component, which is a record.

3.6.2. Buffered Disk Records.

All the SETL database native packages call a basic record and storage management component (routines starting with 'DR_'), which handle record allocation and deallocation, management of the free list, buffered read-record and write-record operations, record copying and record reference-counting operations. This same family of routines also provides a series of diagnostic calls for assessing memory usage, and a group of auxiliary debug-oriented procedures designed for tracking down memory leaks.

This package must know the structure of all the possible record types, for example compound records or leaves, and all the different kinds of B-trees used to construct the overall database facility, as their layout differs substantially. The current implementation recognizes six different record variants, identified by a different record tag.

We use the following protocol for reference count manipulation: reference counts are maintained for objects of all relevant types by inserting ‘incref’ and ‘xfref’ calls. The use of these routines reflects the consideration laid out in the preceding section.

The ‘incref(x,n)’ routine has a null action on objects of all other types. Whenever we create an object of one of these types, it is given a refcount of 1.

Whenever an assignment x := y is made to a variable whose value before or value after the assignments may be one of these types, we execute the routine ‘xfref(x, y)’ which decrements the reference count of the object y (if it has one of the relevant object types) and increments that of y. Assignments f(z) := y have a like action, namely xfref(f(z),y). Operations like s with x, which put x into a composite, increment the reference count of y, as do operations like z := arb(s) and y := f(z) which extract it from a composite.

Whenever quantities x, y etc. are passed to a procedure their reference counts are incremented; whenever a procedure is exited, the refcount of each of its arguments and local variables must be decremented. Before any modifying assignments to f or s, like f(z) := y, s with y, or f.component := n, we check the refcount of f (or s); if this is greater than 1, we replace f by a copy cf of f (and we must increment the refcount of each element of f by 1, but decrement the refcount of the original version of f by 1); the change is then made to the fresh copy of f.

If the reference count of f falls to zero when decremented, f is added to the free list, and the reference count of each object it references is also decremented by 1; this may cause these objects to be added to the free list in turn.

Each object obj for which we manage refcounts explicitly must therefore provide a method obj.increment_refs(n) which iterates over all the objects which it references and adds n to their reference counts; and also a method obj.increment_ref(n) which calls the standard incref(x,n).

3.6.3. Large String Management with reference Counting.

A SETL package of routines called ‘B_tree_for_bigstring’ and the underlying native routines (all the methods starting with ‘BNR_’) provide basic B-tree manipulation, as adapted for trees whose leaves contain strings, and in particular handles most of cumulant and reference count maintenance. This package is largely a reworking, as a partly native package, of the generic B-tree class described previously. Its routines use string length as a cumulant. They differ from the generic B-tree routines presented above and in section 3.8 below principally by the addition of reference count maintenance instructions and by modifications of the recursive leaf search sequences seen in the generic code needed to handle character positions rather than leaf numbers.

The '`big_string_pak`’ and the corresponding ‘big_string’ class superimpose a string-like syntax on the underlying B-tree capabilities which

‘B_tree_for_bigstring’ provides, and also handle most leaf-level character manipulation.

3.6.4. *The full Database design*

Once we have the ability to store efficiently arbitrarily large SETL strings, we can proceed to design SETL databases of the type described in the first section of this chapter. These are implemented using underlying objects of two types (ignoring those other structures needed for crash recovery). These are the *logical record list* and the *word index*. The logical record list can be thought of as an ordered list of strings, each consisting of the binstr version of a SETL object (the 'record'), prefixed by a 4- (possibly 5-) byte identifier field (the 'record id', generated to be unique). This list is kept in lexicographic order of the record id fields within it.

In abstract logical terms, the word index is an ordered tuple of pairs [h,id], where h is the hash h = Hash_function(w) of a word w, and id is the index of a database record R whose string summary contains a (blank-delimited) word having the hash h. Since the (system) hash function used makes it unlikely that two distinct words drawn from the collection appearing in such a string have the same hash, almost all pairs [h,id] in the word index will correspond to cases in which a desired word w appears in R's string summary. Abstractly speaking, the word index is kept in lexicographic order of the pairs [h,id] in it. However, since this can be expected to create long runs of pairs with identical first components, efficiency is improved by suppressing the first component of every pair in such a run, except for the very first pair in the run. This results in a tuple of elements

$$h, id_1, id_2, .., id_n, h', id'_1, id'_2, .., id'_{n'}, , h'', id''_1, id''_2, .., id''_{n''}...$$

where the elements h appear in increasing order, and the elements id appear in increasing order between any two successive elements h but not overall. The cumulants kept for this tuple record represent both the number of components descending from each node N in its representing B-tree and the largest hash h descending from N and any of its left siblings. This data structure gives us fast access to the last tuple component of type h satisfying any specified inequality h <= H, and the next tuple component of type h' following h. In this way, we find the run id_1, id_2, ..,id_n of components of type id which appears between h and h', and so we have what we need to implement database iterator objects and db.contains(wd), and the further iterator objects

iter_obj + iter_obj (union iteration)

iter_obj * iter_obj (iterate over intersection),

iter_obj - iter_obj (iterate over difference)

deriving from them.

Logical record list object and word index objects must, like the bigstring objects described earlier and the SETL database objects which record list and word index objects serve to implement, have value semantics. Since they are stored on disk, they must be supplied with the internal reference-count facilities needed to implement value semantics for very large, disk-stored objects.

Two objects are used to access the records stored in two ‘bigstring’ objects. The first, described in the previous section, stores the list of records as a large string. The second big_stg_for_wdoc_pak, which is based on B_tree_for_wdocstring and the ‘WO_*‘ native routines, is a specialized ‘bigstring’ optimized for storing strings of record numbers.

The code needed to implement the logical record list described above is supplied in a package called B_tree_for_dbix and the corresponding 'DBIX_*' native routines. These implement a specialized B-tree with two cumulants: one for the record size, the second for the last record id. This structure is used to index the records stored in a large string B-tree.

Finally, words used for indexed search in the database are indexed using a package called B_tree_for_wdix package and its associated 'WIX_*' native routines. This B-tree also comes with two cumulants: the first cumulant is used for the total number of occurrences, the second cumulant indicates the last index word. This last structure indexes the records stored in the specialized bigstring for word occurrences.

The following simple database dump, from the program of section 3.4 shows the internal structure of the database (before its records are edited):

```
DATABASE RECORDS:
record 1: [1..116] ["CAE85316",
"manklflvcatfalcflltnasiyrtvvefdeddasnpmgprqkcqkefqqracqk",
"Arabidopsis thaliana"]
record 2: [117..224] ["CAC80708",
"anyggdkqygretrqtgdyenpihstggqyeqdvrqtdeygnpvrrtdey", " Helianthus
niveus"]
record 3: [225..331] ["AAL69565",
"msccngkcgcgmypdvevsattvmivdgvapkqmfaegsegsfvaeg", "Helianthus
tuberosus"]
INDEXED WORDS AND WORD OCCURENCES:
O:: (1..1 )[2]
O:niveus: (2..2 )[2]
L:AAL69565: (3..3 )[3]
L:CAC80708: (4..4 )[2]
L:CAE85316: (5..5 )[1]
O:Arabidopsis: (6..6 )[1]
O:Helianthus: (7..8 )[2, 3]
O:thaliana: (9..9 )[1]
O:tuberosus: (10..10 )[3]
```

The database consists of a large string of 331 characters. Using the B_tree_for_dbix index, we can locate each record in the bigstring. The corresponding ‘word occurrences bigstring’ has 10 records indexed with the B_tree_for_wdix B-tree.

Since all needed B-tree operations, can be implemented within the logical record list and word index classes provided, little remains to be done in the top-level database class. This toplevel code simply needs to coordinate the logical record list and word index objects that it contains, and to give database operations their intended syntax. The iterations on 'contains' objects are provided by an auxiliary iterator object class.

3.6.5. Generalized Databases

Since database objects themselves are so simple, we can readily create specialized new kinds of databases by modifying the code seen above to extend its functionality. For example, one can create databases, which store other databases within records, even hierarchically. Also the records held in generalized databases need not be SETL objects small enough to be held in RAM, but can include very large strings and tuples held on disk and edited only piecemeal. The B-trees used to realize these strings and/or tuples can support additional cumulants of the kind described earlier, so that they can be kept in some sorted order whenever insertions are made in them, or can support various kinds of dynamic summation or specialized access. The codes described in this chapter are not hard to extend to support such features if desired.

Various other cumulants allow representation of other kinds of disk-storable objects. One such example is the 'large tuple of long strings'. This can have billions of components, and individual components can be billions of bytes long. The crucial operations linking such objects to the SETL environment can be represented as

stg := obj(n..[j,k]);   and   obj(n..[j,k]) := stg;

The first of these operations extracts characters j through k from the n'th component of the vector of strings and returns it as a standard SETL string (which should be no more than a few megabytes in length). The second assigns a standard SETL string to this same range of characters. We also have

obj(n..m) := tup;   and   obj(n..m) := **OM**;

where tup is a standard SETL tuple of strings. The first of these inserts a vector of strings into the stated range of the 'large tuple of long strings'; the second deletes a range of components.

To represent these objects we need only introduce a family of B-trees with two cumulants: total number of characters in a node's descendant leaves and total number of breaks between vector components in a node's descendant leaves.

Another important case is that of string-to-string mappings, which in abstract terms are sets of pairs of strings [key, val], the key being of no more than standard SETL size, but the string value being possibly very large. The abstract mapping from 'key' to 'val' can be many-to-many, but for simplicity in the present remarks we shall suppose that it is single-valued. Here the main operations can be represented as

stg := obj(key..[j,k]);   and   obj(key..[j,k]) := stg;

The first of these operations extracts characters i through j from the val string located by the key 'key'. The second assigns a standard SETL string to this same range of characters. We also have

obj(key) := stg; and obj(key) := **OM**;

where stg is a standard SETL string. The first of these inserts a new [key,val] pair; the second deletes the pair with the stated key.

To represent these objects we can introduce a family of B-trees with two cumulants: total number of characters in a node's descendant leaves and largest key-hash in a node's descendant leaves. Here each record is assumed to consist of four abutting fields

hash_of_key, length_of_key, key, val

Here it should be understood that hash_of_key is a hash of a record's 'key' value. The key itself can be of an arbitrary length, given by the 4-byte length_of_key field. Records are stored in order of their hash_of_key field. Keys with identical hash_of_key values should occur only very rarely; but when they do, they will be stored successively, in random order. Search for the record with a given key then proceeds by binary search on the key's hash, followed by key verification at the bottom level. In the rare case in which several keys have the same hash, a slightly more elaborate serial search at the bottom level is required.

3.7 Comparison with similar database libraries

Objects and operations much like those described in the final paragraphs of the preceding section are central to the Berkeley database management library[53]. This popular system is widely used both directly and as the basic data management component for many public domain and commercial database engines.

Berkeley DB is a very solid embedded database system that can be used in applications requiring high-performance concurrent storage and retrieval of key/value pairs. However, during our experiments with it for storing very large genomics objects we found that the DB->put operation in DB_DBT_PARTIAL mode, which is equivalent to the

obj(key..[j,k]) := stg;

described above, reads the entire value portion of the record. Since Berkeley DB uses a simple architecture to store large data portions in overflow pages, updates in the middle of a large data section involve rewriting all the subsequent chunks. A similar architecture is used in Postgres SQL [75] to store large fields. We note that the extended B-tree structure described in this chapter solves this problem.

The other work on efficient storage for large objects described in [14][15][16] is based on trees, but does not exploit their internal semantics. The system proposed here can easily be extended, supports fast incremental updates, and is thus a perfect addition tool for VALIS.

3.8 Details of the other principal B-tree operations

The two basic routines in the ‘Btup’ class code are those for the operations 'self(i)' and 'self(i) := x', presented above. After these two most essential routines, the next most crucial is the following concatenation routine.

```
procedure self + x;  -- concatenation

 if is_tuple(xx := x) then x := btup(); x := x.set(xx); end if;
    -- force second argument to btuple
 if #cum = 0 then return x; end if;
 if #x.cum = 0 then return self; end if;  -- null cases

 new := btup();   -- create an empty B-tuple

 if x.h = h then   -- concatenated btup has same height

  new.tup := tup + x.tup; new.h := h;  -- perform a top-level concatenation

  c1l := cum(#cum); new.cum := cum + [c + c1l: c in x.cum];
      -- get cumulant of concatenation, adjusting second part
  return new.ifsplit();   -- split if necessary

 elseif x.h < h then  -- concatenated btup is shorter

  new.tup := tup; new.cum := cum; new.h := h;
    -- copy the top node of this tree

  end_elt := tup(nt := #tup);  -- the final descendant of this tree's root
  end_elt := end_elt + x;  -- recursively catenate x, 1 level down

  if end_elt.h < h then    -- the subconcatenation has not split
   (new.tup)(nt) := end_elt;   -- just install
   new.cum(nt) := new.cum(nt) + x.cum(#x.cum);
    -- adjust the cumulant to reflect the added descendants

   return new;  -- return the new tree
  end if;

    -- Otherwise the subconcatenation has grown taller, i.e. has split.
    -- The descendants of the new top level root
    -- replace the prior last node at this level
    -- (which is the node to which x was concatenated)
  new.tup(nt..nt) := end_elt.tup; new.h := h;
      -- replace the last child of the original tree
      -- with the children of the new split sub-tree

  c1ml := if nt = 1 then 0 else cum(nt - 1) end if;
    -- get cumulant prior to last node of original tree
  new.cum(nt..nt) := [c + c1ml: c in end_elt.cum];
    -- add this to all subsequent cumulants

  return new.ifsplit();   -- split if necessary

 else  -- otherwise concatenated element is taller

  new.tup := x.tup; new.cum := xc := x.cum; new.h := xh := x.h;
    -- copy the top node of x
  first_x_elt := x.tup(1);  -- the first element of x
  tot_cum := cum(#cum);   -- total cumulant of original tree

  first_x_elt := self + first_x_elt;
    -- recursively catenate this tree to first child of x
  if first_x_elt.h < xh then  -- the subconcatenation has not split
   (new.tup)(1) := first_x_elt;   -- it becomes first child of the new tree
   for j in [1..#xc] loop
    new.cum(j) +:= tot_cum;   -- adjust the later cumulants
   end loop;
   return new;  -- return the new tree
  end if;

    -- otherwise the subconcatenation has grown taller, i.e. has split
    -- the descendants of the new top level root replace
    -- the prior first node at this level
    -- (which is the node of x to which our original tree was concatenated)
  new.tup(1..1) := first_x_elt.tup;
      -- replace the first child of the original tree
      -- with the children of the new split sub-tree

  new.cum(1..1) := first_x_elt.cum;
    -- likewise replace the cumulant of the first child of the original tree
  for j in [3..#xc + 1] loop   -- adjust all later cumulants
    new.cum(j) +:= tot_cum;
  end loop;

  return new.ifsplit();   -- split if necessary

 end if;

end;
```

Concatenation is most easily understood in the special case that the two trees being concatenated are of the same height. In this case, it should be clear that they can be concatenated simply by concatenating their top-most nodes. If this generates a top node whose length is less than the limit 2n, on which we insist, we have only to set the cumulant vector correctly; otherwise we split the children of the over-fat top node that results into two parts, making each of them the child of a separate parent, and introducing a new top node having these nodes as its children. But if the two nodes being concatenated are not of the same height, then either the first or the second will be taller than the other. Suppose, to begin with, that the first tree t_1 is taller than the second tree t_2. Then we descend to the rightmost child t'_1 of the first tree, and recursively concatenate t'_1 with the second tree. If the resulting tree c'_1 is of the same height as t'_1, then it replaces t'_1 as the final child of t_1, thereby defining a new B-tree r_1 whose internally stored cumulant we simply need to adjust. But if c'_1 is taller than t'_1 it will have two children, each of the same height as t'_1. In this case these two children replace the single child t'_1 of t_1, resulting in a new B-tree r_1. If this r_1 does not have too many children, its stored cumulant is simply adjusted and it becomes the concatenation result. But if r_1 has too many children, we spit these children evenly among two new parent nodes, which then become the two children of a result tree one level higher than the original t_1, and so forth iteratively. This conditional splitting operation, like all others in the code which follows, is performed by the utility routine 'ifsplit()' seen here.

```
procedure ifsplit();  -- split into 2 nodes if overpopulated

  if (nt := #tup) <= maxw then return self; end if;

      -- needn't split if not above length limit
  t1 := tup(1..nto2 := nt/2); t2 := tup(nto2 + 1..); -- split the top node in half
  c1 := cum(1..nto2); c2 := cum(nto2 + 1..);   -- split its cumulant in half
  cum1 := cum(nto2); cum2 := cum(nt);
```

```
     -- get cumulants for the two new nodes that will be formed

     -- form a new root with just two descendants
   new1 := btup(); new1.h := h; new1.tup := t1; new1.cum := c1;
     -- initialize first child and its cumulant
   new2 := btup(); new2.h := h; new2.tup := t2;
   new2.cum := [c - cum1: c in c2]; -- initialize second child and its cumulant

   newtop := btup();  -- create a new empty node
   newtop.tup := [new1,new2];
   newtop.cum := [cum1,cum2]; newtop.h := h + 1;
     -- attach children, cumulant, and set height
   return newtop;
 end ifsplit;
```

The case of concatenations in which t_2 is taller than t_1 is handled symmetrically to the case just described.

The concatenation procedure we have just described is used as a subroutine by various other important procedures in the set seen below, including end-slice extraction, general slice extraction, and slice assignment.

```
procedure self(i..);  -- end slice extraction

  if i > (cncp1 := cum(nc := #cum) + 1) or i < 1 then
    abort("end slice index out of range: " + str(i)); end if;
  if i = cncp1 then return btup(); end if;  -- empty result
  if i = 1 then return self; end if;    -- the whole shebang

  if h = 1 then    -- minimal height case; slice the tuple
   new := btup(); new.h := 1; new.tup := tup(i..); new.cum := [1..nc - i + 1];
   return new;
  end if;

  must := exists c = cum(j) | c >= i;
    -- find the child position to which the slice propagates
  cumbef := if j = 1 then 0 else cum(j - 1) end if;
     -- get the cumulant prior to position of the sliced child
  tj := tup(j);    -- get the child to which the slice propagates

  if j = nc then return tj(i - cumbef..); end if;
    -- just return slice of the child if this is last
  tail := btup(); tail.h := h; tail.tup := tup(j + 1..);
    -- otherwise get siblings following the sliced child
  cj := cum(j); tail.cum := [c - cj: c in cum(j + 1..)];
    -- adjust cumulant for this isolated 'tail' group of siblings

  return tj(i - cumbef..) + tail;
    -- catenate the 'tail' to the sliced child, and return

 end;
```

End-slice extraction, which is written as t(i..) where t is a B-tree object and i an integer, is performed as follows. We first locate that child t' of the top node of t whose associated subtree contains the leaf with index i. Let c be the total number of leaves in children of t which precede t'. Then we form the B-tree t_2 whose children are those children of t which follow t', form t'(i - c..) recursively, and concatenate t'(i - c..) and t_2 + t to get the desired result. It should be clear that a prefix-slice extraction t(1..i) could be performed in a manner symmetrical to the procedure just described, and that a general slice extraction t(1..i) can be viewed as a combination of a prefix-slice and an end-slice operation. The detailed code for general slice extraction seen in what follows does essentially this, but in a slightly optimized way, which we leave to the reader for further examination.

```
procedure self(i..j) := x;    -- slice assignment

 if is_tuple(xx := x) then       -- force x to btup if it is a tuple
   x := btup(); x := x.set(xx);
 elseif type(x) /= "BTUP" then    -- otherwise x must already be a btup
   abort("illegal slice-assignment right-hand side: " + str(x));
 end if;

                    -- check that first index is in range
 if i > (cncp1 := (cnc := if (nc := #cum) = 0 then 0 else cum(nc) end if) + 1)
                    or i < 1 then
   abort("first slice-assignment index out of range: " + str(i));
 end if;

     -- check that second index is in range
 if j < i - 1 or j > cnc then
         abort("second slice-assignment index out of range: " + str(i));
         end if;

 if i = 1 then       -- the over-written part of this tree is a prefix

   if j = cnc then  -- the whole initial tree is over-written; just copy x to self
     h := x.h; tup := x.tup; cum := x.cum; return x;
         -- modify this btup; return right-hand side
   end if;

        -- otherwise only a prefix of the initial tree is over-written
   tail := self(j + 1..);
         -- extract the trailing B-tup section that is not over-written
   new := btup(); new.h := x.h; new.tup := x.tup; new.cum := x.cum;
         -- make a copy of x

   new := new + tail;      -- catenate the assigned part plus the retained part
   h := new.h; tup := new.tup; cum := new.cum;    -- modify this btup
   return x;                     -- return right-hand side

 end if;
        -- in this final case, a middle portion of the original tree is over-written
 pref := self(1..i - 1);       -- get the retained prefix
 new := btup();
 new.h := x.h; new.tup := x.tup; new.cum := x.cum;
         -- the assigned part

 if j = cnc then    -- over-written part is suffix
   new := pref + new;      -- catenate the retained part plus the assigned part
 else          -- over-written part is middle
   tail := self(j + 1..); new2 := btup();
   new2.h := x.h; new2.tup := x.tup; new2.cum := x.cum;
   new := pref + new + tail;
         -- catenate the retained part plus the two assigned parts
 end if;

 h := new.h; tup := new.tup; cum := new.cum;      -- modify this btup
 return x;                     -- return right-hand side

end;
```

Slice assignment t(i..j) := t^* is performed as if the result to be formed had been written as t(1..i - 1) + t^* + t(j + 1..). The remaining B-tree operations in the 'Btup' class code all have easy expressions in terms of the basic operations we have just described.

The prototype 'Btup' code [31] reflects all of the foregoing design considerations, which it expands in detail.

Next we show templates for some of the key procedures used to manage the iterator objects associated with our btup objects, and to handle iterations over these and their Boolean combinations.

```
procedure iterator_start;     -- initialize simple iteration over btup
    -- sets up iterator stack as value referenced by iter_ptr.
    -- This is a stack of pairs [tup,posn]
  stack := [];    -- to be built

  node := self;
  for j in [1..h] loop
    stack with:= [notup := node.tup,if j = h then 0 else 1 end if]; node := notup(1);
  end loop;

  ^iter_ptr := stack;      -- attach stack to iteration pointer

end iterator_start;

procedure set_iterator_start;
        -- initialize second-form iteration over btup (similar to iterator_start)
        -- sets up iterator stack as value referenced by iter_ptr
  stack := [];    -- to be built
  ^compno := 0;

  node := self;
  for j in [1..h] loop
    stack with:= [notup := node.tup,if j = h then 0 else 1 end if]; node := notup(1);
  end loop;

  ^iter_ptr := stack;      -- attach stack to iteration pointer
end set_iterator_start;
```

The family of routines for iterating over B-trees works by maintaining a stack of tree nodes representing the sequence of ancestors leading to the leaf x currently reached by an iteration. Each stack entry stores a tree node t, along with the position i of the next lower node in the chain among the children of t. If the leaf x is not the last child of the bottom-most node t' in this chain, the iteration is advanced simply by moving on from x to the next child of t'. Otherwise we must locate the bottom-most node t^* in the sequence whose final child has not yet been reached by the iteration, advance to the next child t_2 of t^*, and then rebuild the part of the stack following this t_2 by appending the chain of children leading to the first node in the subtree headed by t_2. Details of the code just outlined are found in the 'iterator_next()' routine seen below.

```
procedure iterator_next();     -- step simple iteration over btup
    -- returns value as singleton tuple  whose value  is a pair, or OM if terminating
    -- advances iterator stack referenced by iter_ptr
  stack := ^iter_ptr;    -- retrieve the iterator stack

  height := 1;

  for j in [ns := #stack,ns - 1..1] loop
    if (sj := stack(j))(2) = #sj(1) then
      height +:= 1; removed frome stack;    -- remove any exhausted element
    else
      exit;
    end if;
  end loop;

  if height = 1 then
    removed frome stack;  -- pop the top element; then advance it
    [tup,loc] := removed;
    result := tup(loc +:= 1); stack(ns) := [tup,loc];
    ^iter_ptr := stack; return [result];
          -- return singleton tuple built from leaf element
  end if;

  if stack = [] then return OM; end if;
          -- iteration is exhausted

  removed frome stack;
          -- pop the top element, then advance it and use to rebuild rest of stack
```

```
 [tup,loc] := removed; node := tup(loc +:= 1); stack with:= [tup,loc];

 for j in [1..hm1 := height - 1] loop
        -- rebuild the stack, starting with the node that was advanced
   stack with:=
        [notup := node.tup,if j = hm1 then 0 else 1 end if];
          node := notup(1);
 end loop;

 removed frome stack;  -- pop the top element; then advance it
 [tup,loc] := removed; result := tup(loc +:= 1); stack(ns) := [tup,loc];
 ^iter_ptr := stack; return [result];
        -- return singleton tuple built from leaf element

end iterator_next;
```

The closely related 'set_iterator_next()' routine adapts this same idea to realize iterations of the modified form 'for y = t(i) loop..."

```
procedure set_iterator_next();     -- step second-form iteration over btup
  -- returns value as singleton, or OM if terminating
  -- advances iterator stack referenced by iter_ptr
  stack := ^iter_ptr;    -- retrieve the iterator stack
  ^compno := cno := ^compno + 1;    -- advance the component number

  height := 1;

  for j in [ns := #stack,ns - 1..1] loop
    if (sj := stack(j))(2) = #sj(1) then
      height +:= 1; removed frome stack;    -- remove any exhausted element
    else
      exit;
    end if;
  end loop;

  if height = 1 then
    removed frome stack;  -- pop the top element; then advance it
    [tup,loc] := removed;
    result := tup(loc +:= 1); stack(ns) := [tup,loc];
    ^iter_ptr := stack; return [[cno,result]];
        -- return singleton tuple built from leaf element
  end if;

  if stack = [] then return OM; end if;    -- iteration is exhausted

  removed frome stack;
        -- pop the top element, then advance it and use to rebuild rest of stack
  [tup,loc] := removed; node := tup(loc +:= 1); stack with:= [tup,loc];
```

```
 for j in [1..hm1 := height - 1] loop
         -- rebuild the stack, starting with the node that was advanced
   stack with:= [notup := node.tup,if j = hm1 then 0 else 1 end if]; node := notup(1);
 end loop;

 removed frome stack;  -- pop the top element; then advance it
 [tup,loc] := removed; result := tup(loc +:= 1); stack(ns) := [tup,loc];
 ^iter_ptr := stack; return [[cno,result]];
         -- return singleton tuple built from leaf element

end set_iterator_next;
```

3.9 Historical notes

In his 1973 book[54] Knuth suggested (as an exercise) using B-trees augmented with what we call a “count cumulant” to access elements of large lists efficiently. For many years these “Ranked B-trees” were used almost exclusively for sampling in databases, although acceptance/rejection (A/R) sampling methods were usually preferred as they could be applied to standard B-trees.

Unfortunately these ranked B-trees dropped largely out of use until Srivastasa [55] rediscovered them in 1988 and used them to implement statistical databases. He called his augmented B-trees TBSAM trees and used them to compute averages, sums, standard deviations, and higher order statistics efficiently.

In 1995 Hellerstein proposed the Generalized Search Tree (GiST), a template indexing structure that allows domain experts to customize database system indices, but his project ended in 1999 and only recently [73] has the GiST library finally been used to store Russian-Doll trees [74] in Postgres.

The idea of storing chunks of large objects in B-tree leaves probably appeared first in the Exodus [15] system in 1986. Notwithstanding the advantages of this approach, these data structures were then ignored for many years. As we have seen above, many research and commercial databases still use inferior techniques. Amazingly, even Microsoft has shipped many versions of its Sql Server with a BLOB architecture based on linked lists of blocks, and has only recently (2000) replaced this crude structure with a structure similar to the one used in Exodus.

More recently, ideas similar to those presented in this chapter have been used in the Bento and Quilt data storage facility of the OpenDoc [17] compound document architecture. This project, started by Apple in 1992, moved to CI labs, a consortium of Apple, IBM, Microsoft and others, but was terminated in 1997.

The implementation details of Quilt and Bento are not in the public domain, but from an application point of view each Quilt file contains and organizes an arbitrary number of efficient random access streams and/or B+tree dictionaries that are interleaved in varying formats that are named and typed so that all content is tagged with meta-information that clarifies the purpose of each stream or dictionary. In 1997 the main designer of Quilt/Bento proposed a system called IronDoc[56], which is supposed to be the public domain successor of Quilt, but at the time of this writing it remains an unimplemented design.

3.10 Summary

The databases described in this chapter have distinct advantages over relational databases for bioinformatics data storage. Current databases for biological data usually store only metadata, and leave the bulk of the data in flat files or BLOBs. Extending the basic B-tree data structure with multiple ‘cumulants’ allows us to store unstructured or semi-structured data such as experiment results or genome annotations efficiently.

Chapter 4

GUIs in a multi-language environment

4.1 Issues in GUI design

As noted in the introduction to this thesis, Graphical User Interfaces play a very important role in bioinformatics. Many of the most widely used applications, such as the UCSC genome browser [41] and Genbank [40] are Web Browser based. GUIs are essential parts of tools to analyze microarrays, for example Spotfire[42][43], also of programs and tools for assembling and finishing DNA sequences, of various phylogeny tree generators, and many other applications.

Many toolkits, libraries, frameworks and environments are available to make the complex task of building the GUI interfaces of applications easier. The GUI Toolkit Framework Page [7] lists hundreds of them.

GUI development frameworks can be divided (very roughly) into two major classes: code-based and RAD (Rapid Application Development) environments. In code-based environments the developer builds the GUI in code using the programming language of his choice. In RAD environments, GUIs are built by placing graphical widgets directly on a model of the interface and then customizing them.

Early RAD tools like MENULAY [89] (1982) offered automatic conversion to C programs of graphically described user interfaces. Macromedia Director (introduced as Micromind Director in 1988) is a quintessential representative of a RAD development system, as it allows most application development (including animation design) to be done graphically, code being added only as needed. (Director, in spite of some clumsiness in its internal scripting language, LINGO, is still unsurpassed for the development of multimedia presentations). Visual Basic (of which version 1.0 was released in 1991) is the most widely used RAD system.

In RAD frameworks, the environment hides the complexity of the underlying libraries and allows the user to place GUI components and set their (initial) properties without writing a single line of code. This makes it much easier to prototype many applications in Visual Basic or Director than in other environments. To make bioinformatics application development and customization accessible to biologists, we have to provide something similar.

4.2 Building GUIs with VALIS

4.2.1. VALIS Forms and widgets

To facilitate rapid development of GUIs for bioinformatics applications, VALIS Studio provides facilities to build them graphically. New graphical ‘forms’ can be created using the VALIS Studio main menu, as seen in Figure 8 below.

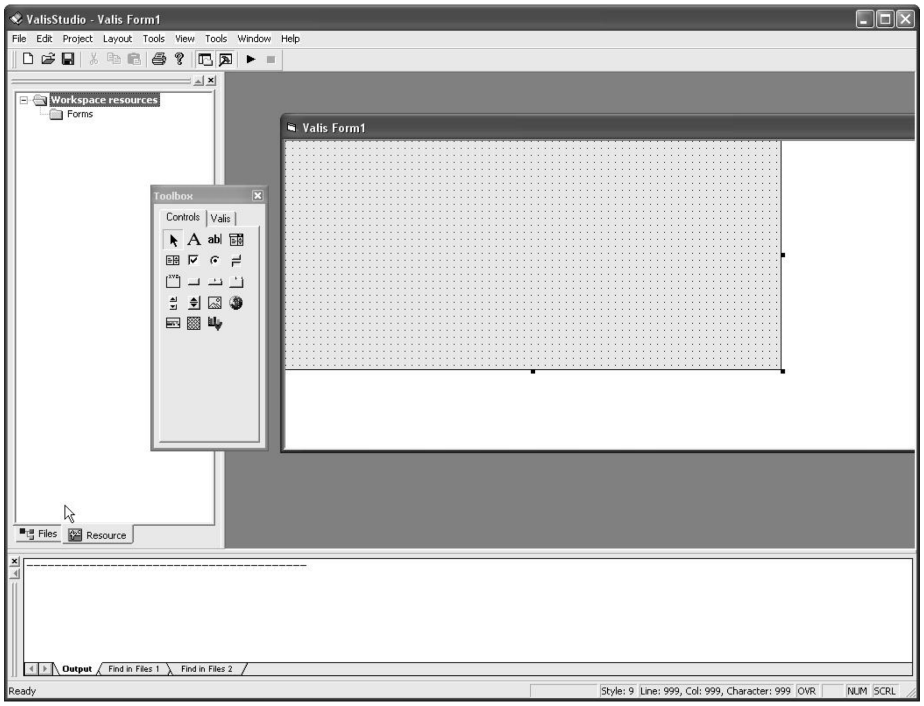
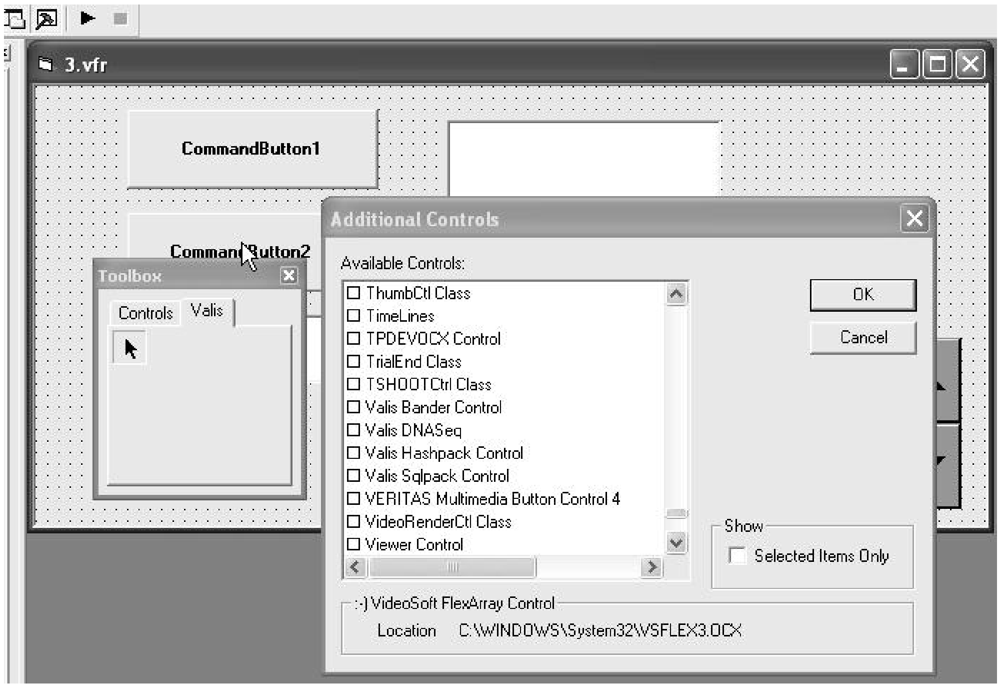


Figure 8. A newly created form in VALIS Studio

The figure shows the 'Control Toolbox' widget, which contains the set of standard graphical widgets (Buttons, Lists, Labels, etc.) available to the user. Any widget seen in the toolbox can be placed onto a VALIS form by dragging its icon onto the developing form.

4.2.2. VALIS Components

Of course this system would be completely useless for bioinformatics if we could not add custom components going beyond the standard controls available. In fact, the user can expand the collection of widgets available in the Toolbox. Additional components are added by right-clicking in the Toolbox widget. This operation recalls a list of all the Widgets that have been ‘registered’ on the machine (see the figure below).



Available Controls:

- ☐ ThumbCtl Class
- ☐ TimeLines
- ☐ TPDEVOCX Control
- ☐ TrialEnd Class
- ☐ TSHOOTCtrl Class
- ☐ Valis Bander Control
- ☐ Valis DNASeq
- ☐ Valis Hashpack Control
- ☐ Valis Sqlpack Control
- ☐ VERITAS Multimedia Button Control 4
- ☐ VideoRenderCtl Class
- ☐ Viewer Control

:-) VideoSoft FlexArray Control
Location C:\WINDOWS\System32\VSFLEX3.OCX

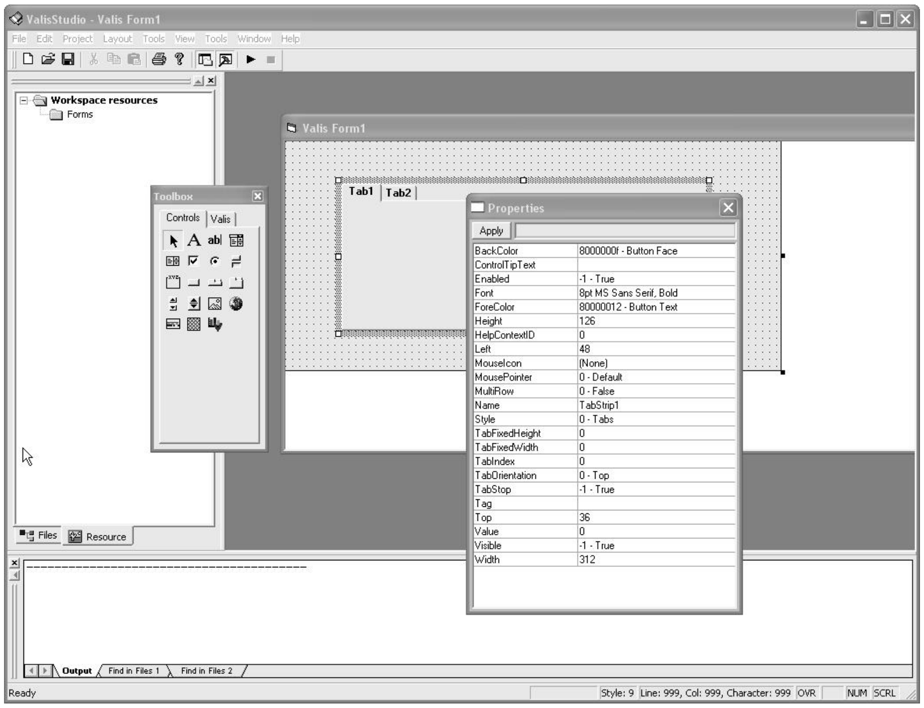
Figure 9. Customizing the Control Toolbox

These additional widgets, which the system user can either acquire or build, will behave exactly like the standard widgets.

Selecting additional widgets places their icons in the Toolbox and makes them available to the user. In the next chapter we will describe some of the high performance controls we implemented to develop some interesting VALIS applications.

4.2.3. Design time customization

Once widgets are placed on VALIS forms, the user can customize them by accessing and modifying a set of ‘Properties’ on which their visual appearance is based. Both design time and run-time customization are possible. At run-time (as we will see later) customization can be performed by VALIS scripts. At design time the user can select any widget and recall its property collection with a right-click, as shown in the following figure:



BackColor	8000000f - Button Face
ControlTipText	
Enabled	-1 - True
Font	8pt MS Sans Serif, Bold
ForeColor	80000012 - Button Text
Height	126
HelpContextID	0
Left	48
MouseIcon	(None)
MousePointer	0 - Default
MultiRow	0 - False
Name	TabStrip1
Style	0 - Tabs
TabFixedHeight	0
TabFixedWidth	0
TabIndex	0
TabOrientation	0 - Top
TabStop	-1 - True
Tag	
Top	36
Value	0
Visible	-1 - True
Width	312

Figure 10. Editing the properties of a widget

A window listing all the widget properties appears and the user can change them. A similar mechanism is available to modify the properties of form objects themselves. Some widget properties, notably position and size, can also be modified graphically by moving and resizing the widget directly on the form being designed.

Complex hierarchies of widgets can be constructed (as in the Simpathica example given in a later chapter) by placing widgets on special 'container' widgets which behave like form objects. (An example of this class of widgets it the Tab Strip.)

4.2.4. Adding forms to a VALIS project

Once the graphical construction of the forms to be used in a project is complete, these objects (and all their design-time properties) can be saved to disk and then added to a special 'Forms folder' available in the VALIS studio workspace (see figure below).

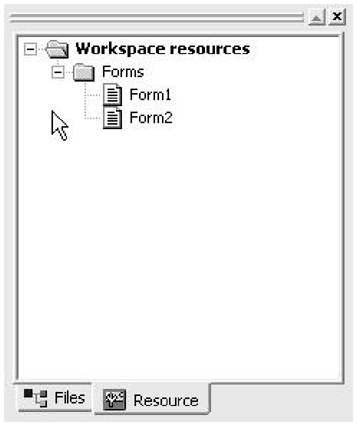


Figure 11. The Forms folder

Forms (and the widgets contained in them) added to this folder are then made available to the VALIS scripts by using the 'Named Item' mechanism described in Chapter 2. (This will be detailed in section 4.4 below.) At run-time each scripting language available in the VALIS environment can further modify all properties of all the forms and widgets.

4.2.5. Widget properties and event handling

In the simple example given below, a Button widget has been added to Form1. Such button widgets can generate the events listed in the following table (events generated by other widgets are similar).

BeforeDragOver	Triggers event when a dragged object reaches the drop target
BeforeDropOrPaste	Triggers event when an object is about to be dropped or pasted into a control
Click	Triggers event when a control is clicked with the mouse; also when the user selects a value in a multi-value control, such as the list box
DblClick	Triggers event when a control is double-clicked with the mouse
KeyDown	Triggers event when a user presses a control, navigation or function key
KeyPress	Triggers event when the user presses a key.
KeyUp	Triggers event when the user releases a control, navigation, or function key
MouseDown	Triggers event when the user presses a mouse button
MouseUp	Triggers event when the user moves the mouse
MouseMove	Triggers event when the user releases a mouse button
Error	Triggers event when an error is encountered

Table 5.Events generated by CommandButtons

To illustrate the way in which events are handled in VALIS, note that (as seen in the table above) any button widget can generate events named “Click”. Such events are handled in VBScript by writing a ‘callback routine’ and giving it the name and structure illustrated by

```
Sub Form1_CommandButton1_Click()
        Print("Click from VB")
End Sub
```

In JavaScript such a handler would be written as

```
function Form1_CommandButton1::Click()
{
    // Close another previously opened form
    Form2.Hide();
}
```

using the (undocumented) Javascript “::” syntax. A Perl callback would be written as

```
sub Form1_CommandButton1_Click {
        $Valis->Print("Valis call from PL");
}
```

The same handler would be coded in Python as

```
def Form1_CommandButton1_Click():
        Valis.Print("here is the python callback")
```

Events are always handled by such callback routines, whose names associate widget names with event names. As seen in these examples, the precise syntax used to support event handling differs slightly among various scripting languages. But each scripting language allowed has such a mechanism.

VALIS handles widget property access using syntax like that used in Visual Basic. In VB one would refer to a ButtonY object inside a form ‘X’ as

‘FormX.CommandButtonY’. We support exactly the same syntax in VALIS. Note however that in the example code above we refer to a button object as ‘Form1_CommandButton1’. These two alternate syntactic styles are both allowed. The VB style of access is available because CommandButtonY is a property of FormX. The other style (preferred in VALIS) is available since FormX_CommandButtonY is added as a COM ‘Named Item’. (Addition of such items is needed to process incoming events. This mechanism will be described in some detail in section 4.4 below.)

4.2.6. Run-time customization

Widget properties can be modified at run-time by operations written using syntax like

JScript	Form1_CommandButton3.caption="JScript";
VBScript	Form1_CommandButton3.caption="VBScript"
Python	Form1_CommandButton3.caption="Python"
Perl	$Form1_CommandButton3->{caption}="Perl";

Table 6. Accessing widget properties in different languages

In addition to properties, forms and widgets can have public methods that can be called by the scripts (see section 4.2.7 for an example).

VALIS graphical applications are projects that show at least one form, and in which execution is driven by form and widget events. Such applications end when all their forms revert to hidden state.

Note that forms are created before the ‘valismain’ startup function is called, but that VALIS commences execution with all the forms in hidden state. Forms can then be shown or hidden using the Show() or Hide() methods.

The following very simple example shows how useful this architecture is. In the example, the ‘valismain’ function is written in Python, and subroutines and event callback functions can be implemented in other languages as shown above.

```
# VALIS main function defined in Python
# Form1 becomes automatically available to the
# scripts after being added to the VALIS forms folder
# but is in hidden state

def valismain():

        # Set the background color property of
        # Form1 to green
        Form1.Backcolor=Valis.RGB(0,255,0)

        # Set the caption of the button named
        # CommandButton3 of form ‘Form1’

        Form1_CommandButton3.caption="JS";

        # Update the title of form ‘Form1’
        Form1.Caption="SPAM"

        # Show the form on screen
        Form1.Show()


// Click event handle for CommandButton1 in VBScript

Sub Form1_CommandButton1_Click()
        Form1.Backcolor=Valis.RGB(0,255,0)
End Sub
```

As seen here, the “Backcolor” property of object Form1 is set by this simple routine to some value (Green), the ‘Caption’ of the window to “SPAM” and the form is then displayed on the screen (see Figure 12 below).

(This code snippet also illustrates a significant difference in coding style between Visual Basic and VALIS. In Visual Basic customization code is always linked to a specific form (although one can access properties of other forms by supplying another form name as a prefix), so one can refer directly to the “Backcolor” property, without having to add the form qualifier. This syntactic decision was made because we have to work with multiple languages, and otherwise would have ended up with a multitude of files, difficult to organize.)

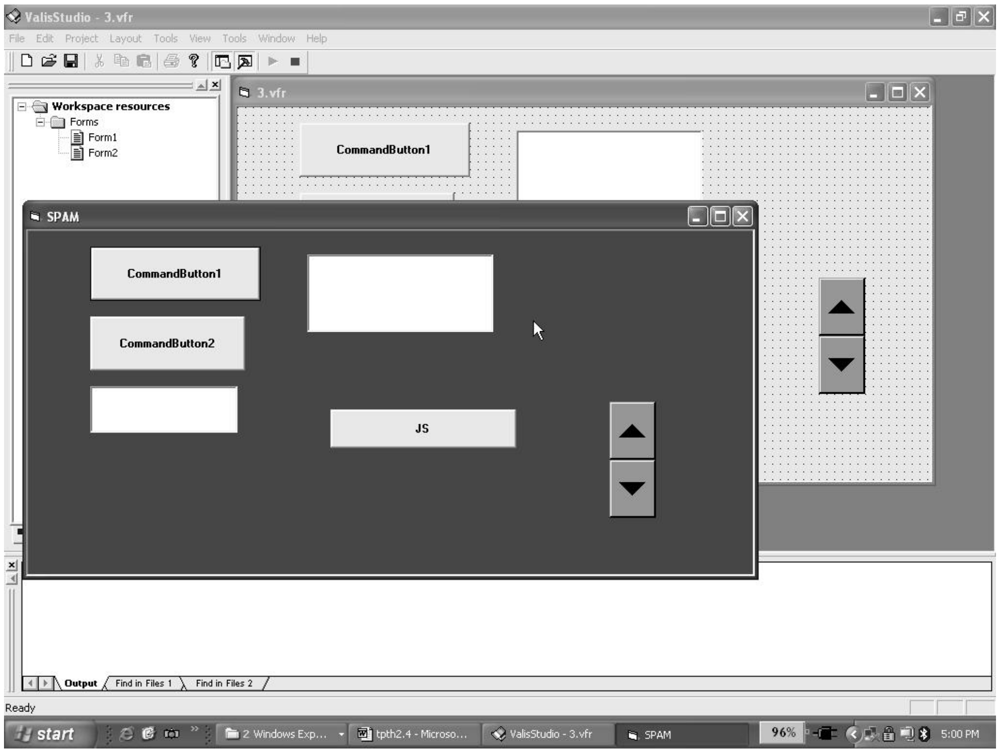


Figure 12. Simple VALIS graphical application.

4.2.7. Run-time construction of GUIs

Some of the design-time capabilities of the VALIS form designer (in particular widget placement and resizing) are also available to scripts at runtime. Container widgets such as the form designer and the MultiPage widget maintain an internal list of the widgets that have been added to them. Access to this list is available at run-time using methods provided by a container's Controls property. The list of widgets can be augmented using the Add method of this Controls property.

The following code snippet shows how a list of Label and TextBox widgets can be added to a MultiPage widget. (A similar mechanism is available for the form designer object.)

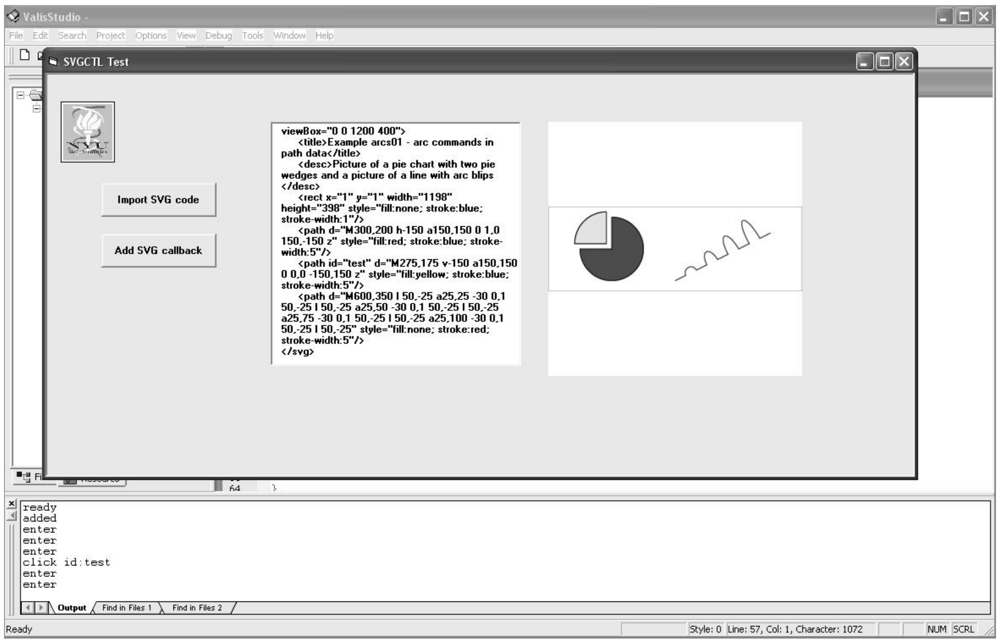
```
// Here we are adding maxl labels and controls to form1

for (i=0; i<maxl+1; i++) {

 // Add a new ‘label’ control to the currently
 // selected pane of the Multipage2 control of Form1
 // Save a reference in the labelControls array
 labelControls[i] =
      Form1_MultiPage2.SelectedItem.Controls.Add(
              "Forms.Label.1", "", FALSE);

 // Customize the newly created label
 labelControls[i].Width = 132;
 labelControls[i].TextAlign = 3;

 // Add a new textbox
 textBoxControls[i] =
      Form1_MultiPage2.SelectedItem.Controls.Add(
      "Forms.TextBox.1", "", FALSE);

// Customize it
 textBoxControls[i].Width = 132;

}
```

4.2.8. Widgets of Canvas type

Various special widgets which behave like Tcl/TK ‘canvas’ objects can be added to a VALIS form. One such widget is the Adobe SVG viewer. SVG is a language for describing two-dimensional graphics and graphical applications in XML. Drawing on SVG involves generating SVG primitives (lines, ovals, rectangles, text, etc.) to capture a description of the output at a higher level of abstraction.

This powerful widget can be customized at run-time in the same way as any other VALIS widget. The SVG viewer allows event callbacks to be attached to SVG graphical elements. In the following example a ‘mouseover’ event on the SVG element called ‘test’ will trigger a callback to the routine called enter (defined elsewhere). (The code attaches a similar callback to the ‘click’ event of this same element.)

```
// Get a reference to the internal object model of the
// SVG viewer present in Form1
SVGDoc=Form1_SVGCtl1.getSVGDocument();
// If the control is ready..
 if (Form1_SVGCtl1.ReadyState == 4) {

  // Obtain a reference to element ‘test’
  e=SVGDoc.getElementById("test");

  // The listener routine will be called when
  // element ‘test’ is clicked
  e.addEventListener('click',listener,false);
  // The enter routine will be called when the
  // mouse moves over the same element
  e.addEventListener('mouseover',enter,false);
}
```



```
viewBox="0 0 1200 400">
    <title>Example arcs01 - arc commands in
path data</title>
    <desc>Picture of a pie chart with two pie
wedges and a picture of a line with arc blips
</desc>
    <rect x="1" y="1" width="1198"
height="398" style="fill:none; stroke:blue;
stroke-width:1"/>
    <path d="M300,200 h-150 a150,150 0 1,0
150,-150 z" style="fill:red; stroke:blue; stroke-
width:5"/>
    <path id="test" d="M275,175 v-150 a150,150
0 0,0 -150,150 z" style="fill:yellow; stroke:blue;
stroke-width:5"/>
    <path d="M600,350 l 50,-25 a25,25 -30 0,1
50,-25 l 50,-25 a25,50 -30 0,1 50,-25 l 50,-25
a25,75 -30 0,1 50,-25 l 50,-25 a25,100 -30 0,1
50,-25 l 50,-25" style="fill:none; stroke:red;
stroke-width:5"/>
</svg>
```

```
ready
added
enter
enter
enter
click id:test
enter
enter
```

Figure 13. Using SVG as a Canvas-type widget

SVG graphical models can be built using XML tools (like the Microsoft XML parser) and then copied to the Adobe Viewer. For an example of this construction see the Simpathica example in Chapter 5 of this thesis.

4.3 Building a Multi-language RAD environment; Technical issues and design choices

A form based RAD environment must at a minimum provide the following features:

1. Forms must have well-defined design time and run-time representations and allow hierarchical assembly of multiple

subcomponents.

2. It must be possible to save all design-time specified layouts and other properties of all forms.

3. It must be possible for scripts to change the properties of both forms and the simple and compound objects they contain.

4. Scripts must have some way of receiving and handling all form-generated events.

Any developer proficient with ActiveX technologies will know how to map these desiderata into Microsoft COM interfaces. In particular, COM provides standard interfaces for defining components, both visible and invisible (ActiveX components), constructing compound documents (OLE Compound Documents), modifying item properties (OLE Automation and Property Pages), generating and receiving events (Connection Points) and saving the state of compound documents (COM Structured Storage)[44].

4.4 Setting up connections between scripts, widget properties and event streams

Once having built interactive forms using a form designer component with the features described in the preceding section, we must make it possible to modify and customize forms and widgets from scripts and to route form events to callback routines provided in script code.

Both capabilities are easily obtained by adding a “Named item” to each scripting engine for each form and widget used in the project. (This ‘Named Item’ mechanism was described in Chapter 2.)

When any of the languages available within VALIS queries the VALIS CScript object (this is the object implementing the ActiveX scripting Host in VALIS) by passing a form name to it, the IDispatch interface of that form is returned to the script, which can then manipulate its properties.

A similar mechanism is used for widgets added to the VALIS forms. Once their corresponding ‘Named Items’ have been added to a scripting engine environment (these objects are marked with a special flag indicating that they can generate events), the scripting engine can request the Host to produce an IDispatch reference to that object. The VALIS Scripting Host will then obtain this reference from the form designer and return it to the scripting engine that requested it.

Once having acquired such a pointer, a list of the names of the events that the object can generate can be obtained by a scripting engine directly from the widget using standard COM interfaces. The following paragraph summarizes these interfaces.

In the COM world, objects capable of generating events are called “Connectable Objects”. To generate events, an object must implement four related interfaces: **IConnectionPointContainer**, **IEnumConnectionPoints**, **IConnectionPoint**, and **IEnumConnections**. The existence of the first interface (**IConnectionPointContainer**) signals that an object has the capability of generating events and possesses a set of ‘Connection Point’ sub-objects. Access to these sub-objects is provided using the **IEnumConnectionPoints** interface. Once having obtained a reference to such a sub-object, a client can use the **IEnumConnections** to obtain a list of all the currently active connections to that particular connection point. Then, using the **IConnectionPoint**::**Advise** method, a client can register a ‘Sink’ routine with the connectable object. Once this is done, then whenever an event is generated, this ‘Sink’ routine will be called. These connections are summarized in the following figure:

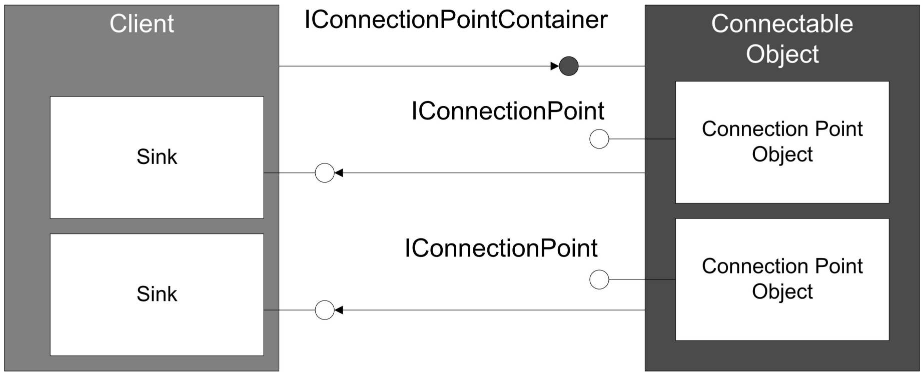


Figure 14. The Connection Point interface

This looks complicated, but luckily the scripting engines available do most of the work for us. We just have to make sure that whenever a “Named item” is added to an engine, its flag SCRIPTITEM_ISSOURCE is set. Then, to handle events from the forms object, we just need to write code like the following (as in our CScript class):

```
// We are considering all the forms present in the
// VALIS project. pWorkSpaceElem points to the current
// Form internal workspace structure

// Now loop trough each of the languages present in the
// VALIS project we want to run
for (int lang=1;lang<=numLanguages;lang++)
  if (Languages[lang].m_pIActiveScript!=NULL) {

    // Obtain the name
    CString formname=pWorkSpaceElem->m_name;
    // Convert Cstring to OLE string
    LPCOLESTR      pstrItemName = T2COLE(formname);
```

```
// Add the form as a ‘Named Item’ that can source
// events
hr=Languages[lang].m_pIActiveScript->AddNamedItem(
        pstrItemName,
        SCRIPTITEM_ISVISIBLE|SCRIPTITEM_ISSOURCE);
if(FAILED(hr))
{
        xxxxxxxxxxxxxxxxx
}
```

Controls in a form are handled in much the same way. We query the form object to find all its controls, and then add “Named Items” having the format FormName_ControlName, and capable of sourcing events to the scripting engine environments from which these controls may need to be accessed.

```
// ‘controls’ points to the CControls interface of
// the Form Designer object.
// The following loop is executed for each
// scripting language and each form present
// in the VALIS project

// Loop trough the list of the controls present
for(long c=0; c<controls.GetCount(); c++)
{

  // Get the name of the control
  CString StrControlName
        =controls._GetItemByIndex(c).GetName();

  // Add the form control to the current scripting
  // engine’s environment, designating it as
  // FormName _ ControlName

  LPCOLESTR    pstrControlName;
  CString cname=formname+"_"+strControlName;
  // Convert Cstring to OLE string
  pstrControlName = T2COLE(cname);

  // They are also capable of sourcing events
  hr=Languages[lang].m_pIActiveScript->AddNamedItem(
        pstrControlName,
        SCRIPTITEM_ISVISIBLE|SCRIPTITEM_ISSOURCE);
        if(FAILED(hr))
        {
                xxxxxxxxxxxxxxxxxx
        }
}
```

(This code shows why we can also refer to Button1 in our earlier example using the syntax "Form1_CommandButton1", rather than as an attribute of the "Form" object.)

It is responsibility of the scripting engines to set-up the Connection Point interfaces described above. Each engine has a different syntax to implement Connection Point 'Sinks'. (As has been illustrated in the introductory section of this chapter.) As an example of how events are routed once the connection point interfaces has been set up correctly, we describe the handling of a mouse click on a form button:

- The Windows operating system sends a message to the active window. (In the VALIS forms case, these messages go to the form designer component.)

- The form designer component then routes the message to the correct widget (the widget hierarchy, visibility and Z-order properties are used to determine this widget).

- The widget then routes the event to all connected scripting hosts using the Connection Points interface.

The interfaces that we have described allow events to be handled in all the supported languages. The only counterintuitive result of adding the same item simultaneously to different engines, is that if each engine has a Sink for a certain event, they will *all* be called, in some undefined order when the event occurs. This follows from the semantics of the Microsoft Connection Point interface.

4.5 Minimizing the VALIS GUI implementation effort by exploiting Microsoft's form designer technology

The technologies that Microsoft uses in their RAD products, for example Visual Basic, Access, Office Forms and Visual Foxpro, have been extremely successful. Like the Microsoft Scripting Interfaces considered earlier, Microsoft's carefully standardized GUI interfaces improve the quality of Microsoft's software and the productivity of programmers developing it.

For this reason we have restricted our attention to the Microsoft technologies to implement the form designer subsystem in VALIS. (But in chapter 6 we will see what would be needed to build a cross platform environment.)

However, although well defined and documented, these technologies are not always easy to apply. During the development of VALIS, we were able to short-circuit involvement in the many details of these interfaces by realizing that since Microsoft is using similar form designers in many applications, it must have defined standard interfaces even for these form designers. Source code for such form designers is not available. Nevertheless binary versions of the Microsoft form design component could be used as the basis for the

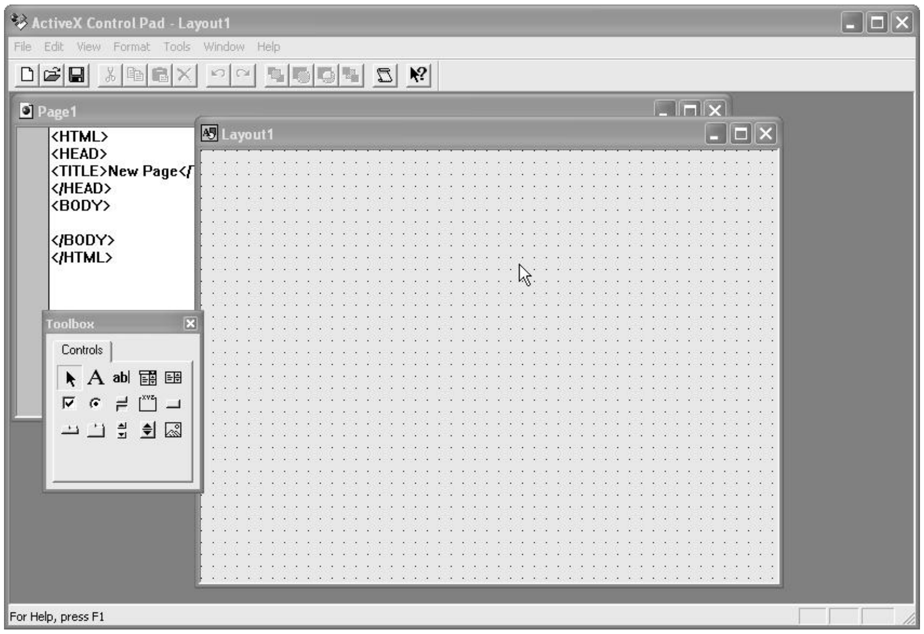
VALIS Studio environment once a complete description of the relevant interfaces was obtained.

4.5.1. Microsoft's ActiveX Form Designers

The Visual Basic 6.0 manual states that form designers other than those provided by Microsoft can be used if developed using the "ActiveX designer SDK". The source for this SDK and a description of the Microsoft ActiveX designer interfaces is available in the January 2000 Microsoft Developer Network CD.

As noted above, many Microsoft Products provide form designers these are generally implemented as COM components developed in conformity with the interfaces described in the ActiveX designer SDK. One can experiment with any of these form designers within Visual Basic. Using the interfaces documented in this SDK, developers can also integrate form designer components into their applications.

There is also a free downloadable Microsoft application (the ActiveX Control Pad[45] shown in Figure 15 below) that contains a complete form designer COM component which follows the specifications of the ActiveX designer SDK. This application was originally released to support point and click design of Internet Explorer web pages which contain ActiveX controls and make use of scripts written in Jscript and VBScript. Source for this application is note available, but once the application is installed, we can gain access to its form design component without having to buy a Microsoft product containing one.


ActiveX Control Pad - Layout1
File Edit View Format Tools Window Help
Page1
Layout1
Toolbox
Controls
For Help, press F1

```
<HTML>
<HEAD>
<TITLE>New Page</
</HEAD>
<BODY>

</BODY>
</HTML>
```

Figure 15. The Microsoft ActiveX Control PAD

In the following sections we refer to the form designer component installed by the ActiveX Control Pad application simply as "form designer". Integration of this form designer object into VALIS is accomplished as follows:

1. The form designer object is embedded in our application using its standard COM/OLE interfaces. These interfaces are applicable to any ActiveX control and to ActiveX compound documents and provide mechanisms needed to save and restore compound documents.

2. The object model of the form designer must be extended to support properties and methods required by VALIS but not available initially in the binary component. The COM 'aggregation' technique used to accomplish this, given that the source code is not available, is explained below.

3. The form designer must be made accessible to the scripting engines. (This is done using the ‘Named Items’ mechanism described in Chapter 2 and the Connectable Objects interfaces described in section 4.4.)

4.5.2. Embedding the Form Designer

We embed the form designer object in which we are interested by defining a special compound document in which the form designer appears as the sole ‘item’ (embedded object). This document is created in the normal way using the COleDocument MFC (Microsoft Foundation Classes) class, and then adding the form designer object, which is an instance of the COleClientItem MFC class. (Any object of this class can be part of a COleDocument as an ‘item’.) In our code the COleClientItem class is specialized to a subclass named CContainerItem.

The implementation of the CContainerItem object is not straightforward. Not only must this object implement the numerous interfaces required for OLE embedding and the special IActiveDesignerRuntimeSite interface (required by the ActiveX designer specifications), but it also has to supply Ambient Properties (described next) and potentially extend the object model of the designer (as described in the following section).

Ambient Properties are used by OLE containers to make certain of their properties available to the controls they contain. They allow controls to adapt themselves to the environment in which they are running. In the form designer case, an example of an ambient property is the path to the project directory, but there are many others. Ambient properties are implemented by programming a special IDispatch interface of the CContainerItem object. Form designer controls use the IDispatch::Invoke method to get the value of the property corresponding to a particular numerical DISPID. The form designer controls must be notified whenever an ambient property is changed.

4.5.3. Extending the Form Designer object model by COM 'aggregation'

Since it meets the standard ActiveX control specifications, the designer object supports OLE automation, so we can use its IDispatch interface to Get and Set properties and invoke functions. In particular, as we have seen in section 4.4 above, we can add the form designer to the scripting engines object models as a "Named Item", just as we did before with the predefined "Valis" object.

But some properties and methods specific to the host application containing the form, and not to the designer itself, need to be supported. For example, the window containing the form designer can be shown or hidden from the user and it must have a 'Caption' property.

Although these properties and methods are actually handled by the VALIS application, it is advantageous to refer to them from VALIS scripts as if they were predefined properties and functions of the form designer object. For example, if this is done and a VALIS project uses a form named 'FormX', we can then refer to the title of the window containing this form directly (in Jscript) as 'FormX.Caption'.

The COM feature is called "aggregation" can be used to implement this feature even for components for which source code is not available. (The form designer component provided by the ActiveX Control Pad supports aggregation.) COM does not support inheritance, but instead achieves code reuse through Containment and Aggregation. Containment works as follows. Suppose, for example, that we have an object A implementing interface IA. If we want to build an object B supporting interfaces IB and IA, we can do so by creating in B an internal copy of the object A and then routing the calls to IA to the internal object A. Doing this we need expose directly the interface IA from A to the clients.

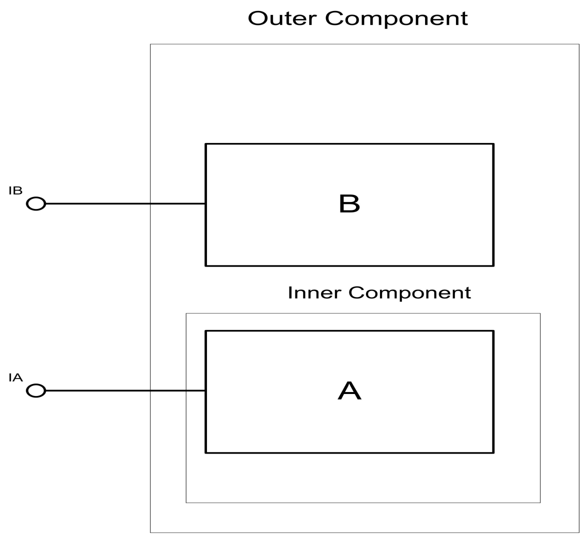


Figure 16. COM Aggregation

Aggregation works instead by exposing directly the inner component's interfaces to the clients. Using this technique, whenever interface IA is requested from object B by passing a globally unique numerical identifier for such an interface to B, the internal object's interface is returned directly to the client. (See figure Figure 16 above.) Aggregation can be implemented using the techniques described in [33].

We use this technique to extend the form designer's IDispatch interface (compiled in the form designer component) with a custom version. That is how we can add custom properties, e.g., the form name, or methods, like Show() and Hide() to the form designer provided by the ActiveX Control Pad.

4.5.4. Additional features of the ActiveX Form Designers

In addition to the standard OLE Compound document interfaces, the ActiveX Designers can implement certain COM interfaces specifically designed for them.

In particular, interfaces exist for the host to manipulate the widget toolbox and supply an alternative implementation. This would greatly improve the usability of the form designer interface, although we have not used it in VALIS.

Another useful service is the ability of the designer to supply type information for the object (in our case the form) dynamically, as the object is edited. This is useful if one wants to implement nifty features like AutoCompletion while writing script code.

Finally, provisions exist to support code navigation from the designer. For example, clicking on a button during form editing could take directly to the corresponding event handler defined in the scripts.

All these additional features are described in detail in the SDK[46], and we refer to that document for more information.

4.6 A survey of other GUI building tools available

Since in VALIS we support many scripting languages, we could build GUIs by direct use of the libraries and tools offered by these languages. To assess this possibility, we start by considering the most commonly used bioinformatics scripting languages, Python and Perl. First consider Python, which provides many toolkits for building GUIs [6]. At the time of this writing, the most realistic Python-based tool choices are essentially four: TKinter, PyQT, PyGTK and wxPython.

Tkinter, which is the default Python GUI, builds on TCL. As noted in a previous chapter, TKInter simply uses a native package providing a few basic functions which allow communication between Python scripts and TK essentially through the TCL programming language. This solution has certainly the advantage of being cross-platform, but it has many weaknesses:

a) One has to deal directly with the intricacies of TCL/TK.

b) There are no RAD environments for this toolkit, which probably reflects the difficulty of synchronizing graphical views of forms directly with the Python code that generates them.

c) The interface is slow, since all the property changes and events responses must go through the TCL interpreter.

d) Finally, and this is a major drawback of most of the toolkits available, it is difficult to expand the widget set.

Another open source option is wxPython, which is based on the wxWindows [48] cross platform widget set. This is a C++ library accessed by a Python native package. This library is not fast either, but seems to perform better than TCL/TK.

However, there are major drawbacks to using this and similar native graphics libraries. wxWindows makes no special provision for custom widgets. Moreover wxWindows and other libraries require binary compatibility between the Python native packages and the C++ library, so that fixes and improvements on the wxWindows library sometimes force the recompilation of these native packages.

BOAConstructor[49] is a RAD application based on wxPython with features comparable to Microsoft’s Visual Studio and other RAD environments. But a simple inspection of the BOAConstructor code shows some of the shortcomings of this toolkit. For example, the list of properties of the widgets and their types cannot be dynamically determined, and all of them are hard-coded in the GUI design application.

In the last few months, there has been a push to adopt some kind of GUI description language based on XML, but again the lack of standards for designing components makes this effort almost useless.

Probably the only group that has focused closely on these issues is the team behind the KDE desktop[50]. This group has mimicked Microsoft’s OLE architecture, calling it Open Parts. In fact, their Microsoft Office clone, Koffice, is one of the best public domain programs available, as is their Web browser called Konqueror; but both of these lack scripting and macro capabilities, offering instead a primitive scripting protocol called DCOP[51]. Cross-language integration in KDE (as in GNOME) is achieved by using CORBA. The KDE developers have made the mistake of replacing COM with a component architecture based on CORBA. This is a reasonable solution for integrating out-of-process external components, but for in-process components (provided in shared libraries or windows DLLs) CORBA users must pay an unacceptable marshalling penalty (especially when dealing with graphical user interfaces).

The company behind the QT widget set, Trolltech, has released a product QSA[10], the QT Scripting Architecture, providing a JavaScript interpreter that can use scripts to control C++ programs derived from Trolltech’s QObject class. These programs must be compiled with Trolltech's Meta Object Compiler, which extends C++ with "Signal" and "Slots" primitives to handle widget events, and also adds some Run Time Type Information. If one uses C++ as a principal tool for application development, this is an interesting and elegant approach, although one lacking the simplicity and language neutrality of COM based technologies.

A similar open source project, called KJSEMBED[52], was started in September 2003, but paradoxically Microsoft's ActiveX Scripting is an open technology, while Trolltech's is proprietary and only available to holders of their "Qt Enterprise Edition" license.

In conclusion, although in last few months there has been an interest in scripting technologies in relation to GUI design, the lack of a standard component and scripting architecture has had a punishing effect on most of the open source efforts.

4.7 Summary

Graphical User Interfaces are very important in bioinformatics prototyping. We have built on the multi language scripting engine previously described and developed an architecture, based on well known but complex Microsoft technologies.

Exploiting an obscure application, Microsoft's ActiveX Control Pad, we were able to build a RAD environment having features similar to Visual Basic, but supporting multiple scripting languages simultaneously.

We believe that this environment is potentially superior to competing efforts by both companies and large open source development teams.

Chapter 5

VALIS in Action

Having now described the foundations of our architecture, it is time to show how it can be used to develop useful genomics applications. Our multi-language scripting architecture gives us immediate access to a vast library of high level and native code and many language interpreters. Besides JavaScript and Visual Basic script, which come with Windows machines, ActiveX scripting compliant engines exist for Perl[47], Python[47], Haskell [23]and Ruby[57]. Moreover, we developed a scripting engine for SETL, and members of the NYU/Courant Bioinformatics group (Dr. M. Antoniotti and Mr. V. Mysore) have collaborated to develop engines for Common Lisp [39] and R[38] that are now available within VALIS. (The R project [37] is an open source alternative to the S programming language and environment developed at the Bell Laboratories and the S-PLUS commercial product based on it.) R provides a wide variety of statistical (linear and non-linear modeling, classical statistical tests, time-series analysis, classification and clustering), and graphical primitives.

In the course of work in this thesis we have developed additional high-performance components for data analysis and visualization. These include components for the GSL (Gnu Scientific Library) numerical library [58] and visualization tools for genomic maps, sequences and graphs (based on Adobe's SVG viewer[59]).

5.1 Application Example 1. A Whole Genome browser

The increasing number of whole genome sequence data sets available gives researchers an opportunity to explore genetic relationships both between species and within individual species. To exploit this opportunity fully, new tools for rapid analysis of whole genomes are required.

The goal of whole genome comparison is to point exact and approximate inter-genome matches and to characterize the features of the other large and small regions of interest (i.e. single point mutations, insertions, deletions, transpositions, tandem repeats etc.). Standard dynamic programming alignment algorithms do not work well in this setting because of the enormous data sets involved.

Many tools have been developed to study the features common to multiple genomes. Among them, the best known are MUMmer[60], which uses suffix trees [61] for pairwise alignment of whole genomes, and REPuter [62] which is also based on the use of suffix trees. Other tools, like SequeX[63], which is based on suffix arrays [66] and an extension of the String B-tree[64], try to compute and visualize k-mer statistics of genomes, computed from all k-mers (i.e. all substrings of length k occurring in a genome). Finally tools like QUASAR [65] use a moving average of k-mer statistics as an index of similarity.

The problem with most of these tools is that it is difficult to select the parameters that will best reveal genome features and also to visualize the results generated. The biologist who wants to experiment with multiple parameters and find patterns and test strategies to select important regions rapidly will not willingly wait for the time consuming analysis and preprocessing programs that such tools require. (As an amusing sign of this impatience, biologists have even invented a system [13] that assigns pitches to the four DNA bases according to their thermal stability, and found that converting the DNA sequences to music helped (some of them, at least) intuit the meaning of specific sequences, and also made memorizing and recognizing specific DNA patterns easier.)

In this section we show how easily we can create interactive whole genome browsing and comparison tools for large datasets within VALIS. In particular, we can easily imitate and extend tools like the UCSC (University of California at Santa Cruz) genome browser. These allow navigation of various genomes annotated with fixed "tracks" of pre-computed data.

5.1.1. The VALIS Bander Object

We begin our account of applications by describing the 'Bander' display widget which VALIS provides. This is a visualization tool useful for analyzing very large genomes. It displays a set of "bands" which show positional information about genomes. Each band can be thought of as a one-dimensional object representing a map from genome positions to scalar values representing local genome attributes in the vicinity of that genome position. For instance, a band representing the repeat structure of a genome shows regions belonging to known and/or dynamically identified repeats. Similarly, an "energy" band shows the Gibb's free energy, entropy and enthalpy of a short word starting at each genome location. Since, especially for very large genomes, we cannot expect to compute all the information we need beforehand the bands must sometimes be computed on the fly.

The VALIS Bander object is programmed (in a model/view style) as follows:

a) Individual data bands are created. Each band handled by the Bander object must have a type and a name and will represent an array of data.

b) The data bands are connected to their data sources. The data to be loaded into a band can be fetched from flat files, from the VALIS free format database, from a remote database or computed dynamically.

c) New bands can be computed from other bands using an extensive repertoire of unary and binary operations, moving averages, thresholding operations, filters, or using remote servers, an example of which would be the suffix array engine described in the following section.

d) Band views are created. The Bander control displays a set of "graphs" that show the data present in one or more bands. Graphs can be line graphs, histograms, scatter plots, annotations etc.

e) Finally, the graphs must be connected to the data bands. (Each band can be connected to multiple graphs.)

The Bander object is programmed in a multithreaded fashion, involving a loader thread and a compute thread. When a band region is selected dynamically for display, the loader makes sure that all the data associated with this region is fetched from the local and remote databases. Once all the data dependencies are resolved, a compute thread sweeps the bands and computes values to be shown. While the Bander object is idle, the loader thread fetches more data regions for the area being analyzed to improve the response time when the user scrolls along in the display

Bander customization and the addition or deletion of bands can take place at run time.

5.1.2. Using VALIS to build a genome browser

To implement the display portion of a genome browser within VALIS we could easily design a simple form that includes a Bander object along with a few scrollbars and buttons to navigate along the chromosomes for which data is to be displayed.

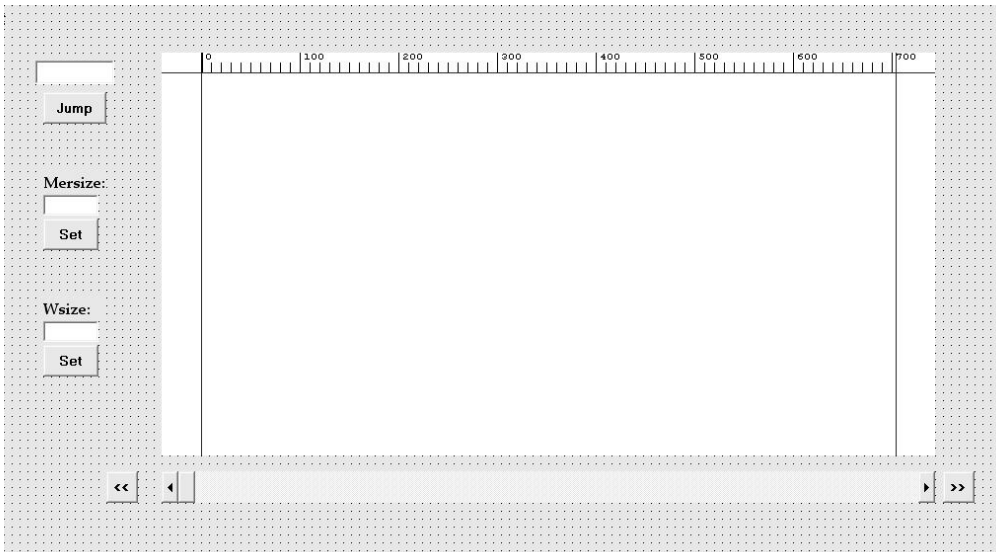


Figure 17. Form containing the VALIS Bander object

We show the required steps of construction in JavaScript. A typical genome-browser utility based on the VALIS Bander object would start by customizing a few basic Widgets:

```
var
 position=0,  // Will keep the first position being displayed
 seqlen=0,    // The maximum sequence length
 jumpSize=0,  // Used to customize the ‘jump’ buttons at the
               // left and right of the scrollbar
 merBand=0,   // Global variable to save the ‘mer’ band number
 avgBand=0,   // Same for a moving average
 windowSize=50,  // Initial window size for the moving average
 merSize=14;     // Initial mer size
```

```
function valismain() {

 Clear(); // Clear VALIS Studio output window
 // Change the window caption and the bander background
 Form1.Caption = "Valis Q.U.A.S.A.R. test";
 Form1_Bander1.SetBackground(RGB(255,255,255));
```

The first “band” to be added to our assumed display is a sequence band fetched from a free-format database (described in Chapter 3):

```
   // Fetch Human chromosome 1 from the database

 b1=Form1_Bander1.AddBand("char","Seq");
 Form1_Bander1.DBBand(b1,"hg15.db","CHR1");
```

We could then add a few markers to signal genome regions not fully sequenced yet (‘N’ regions) and AT rich regions:

```
// Mark regions not sequenced yet

 // Add a graph and a band named ‘N’
 // The graph name will be displayed in the bander
 g1=Form1_Bander1.AddGraph("bool","N");
 bn=Form1_Bander1.AddBand("bool","N");

 // Perform operation ‘CharBand’ on band b1 and output the
 // result on band bn. The result is TRUE when the input
 // character is included in the string passed as the third
 // parameter (in this case n or N
 Form1_Bander1.CharBand(b1,bn,"Nn");
 // Connect band bn to graph g1 and set the band vertical size
 Form1_Bander1.ConnectGraph(g1,bn);
 Form1_Bander1.SetSize(g1,10);

 // Mark AT regions with similar operations
 g2=Form1_Bander1.AddGraph("bool","AT");
 ba=Form1_Bander1.AddBand("bool","AT");
 Form1_Bander1.CharBand(b1,ba,"AaTt");
 // Set the foreground color of the band
 Form1_Bander1.SetColor(ba,RGB(30, 175, 133));
 Form1_Bander1.ConnectGraph(g2,ba);
 Form1_Bander1.SetSize(g2,10);
```

Next we can fetch annotations like the repeat masker regions and the coding sequences from the database:

```
// Add repeat masker blocks from the ‘goldenpath‘ genome database
// The database has been loaded in a free-format database
// This band will contain a list of ‘blocks’ or genome regions
g6=Form1_Bander1.AddGraph("block","RM");
b6=Form1_Bander1.AddBand("block","RM");
 // Fetch, from the database, the regions marked by the
// repeatmasker program for chromosome 1
Form1_Bander1.DBBand(b6,"hg15.db","MASK1");
Form1_Bander1.SetColor(b6,RGB(0,200,0));
Form1_Bander1.ConnectGraph(g6,b6);


// Add coding sequences from the same database
g5=Form1_Bander1.AddGraph("block","CDs");
b5=Form1_Bander1.AddBand("block","CDs");
 // This time the key is ‘MRNA1’
Form1_Bander1.DBBand(b5,"hg15.db","MRNA1");
Form1_Bander1.SetColor(b5,RGB(153, 204, 255));
Form1_Bander1.ConnectGraph(g5,b5);
```

Given a “mer” engine running on one of our servers, we can compute the number of times each “merSize”-mer is repeated in some other genome, in this case the fruit fly, and show it as a histogram:

```
// Add an integer band that will contain the mer frequencies
 merBand=Form1_Bander1.AddBand("int","Freq");

// Set the data source to the mer server running on the cluster
// The server type is ‘merserver’ and the host and port numbers
// are indicated. The output will be stored in the ‘merBand’ band
// created above.  Form1_Bander1.ServerBand(merBand,"merserver",
          "master.valis.nyu.edu:2001");
// Send a command to the server. (select the fruit fly genome.)
Form1_Bander1.Send(merBand,"select_fly");
Form1_Bander1.SetColor(merBand,RGB(95, 95, 95));
Form1_Bander1.Enable(merBand);


// Show it as an histogram (type ‘hist’) and name the graph ‘Freq’
g8=Form1_Bander1.AddGraph("hist","Freq");
// Connect the graphical view with the data band
Form1_Bander1.ConnectGraph(g8,merBand);
```

Finally, we can add a moving average of this mer frequency band and show it as a line graph:

```
// Create the band for the moving average
 // The band type is integer and the internal name Q1
 avgBand=Form1_Bander1.AddBand("int","Ql");
 // The graph will be displayed as a line graph and the band
 // name will be QUASAR
 g9=Form1_Bander1.AddGraph("line","QUASAR");

 // Set the color of the newly created band
 Form1_Bander1.SetColor(avgBand,RGB(255, 0, 0));
 // The avgband operation computes on the fly a moving
 // average of length windowSize of the merBand band
 // storing the result in band avgBand
 Form1_Bander1.AvgBand(merBand,avgBand,windowSize);
 Form1_Bander1.ConnectGraph(g9,avgBand);
 Form1_Bander1.SetSize(g9,200);
```

The rest of our code just completes the initialization of the widgets just described, and updates the mer size and window size displayed:

```
 updatemersize();  // Update the mersize parameter displayed
                   // in the form
 updatewsize();   // The same for the Window size parameter


 // Get the chromosome length
 seqlen = Form1_Bander1.GetMaxrows();
 position=0;    // Starting position
 jumpSize=700;  // Positions to skip when clicking on one of the two
                // jump buttons
 // The scrollbar min and max values are assigned.
 Form1_ScrollBar1.Min=0;
 Form1_ScrollBar1.Max=seqlen;
 // Smallchange is the amount a scrollbar value will be increased
 // or decreased when clicking on the scrollbar arrows.
 Form1_ScrollBar1.SmallChange=300;
 // Largechange is the used when clicking between the scrollbar
 // current position and one of the arrows.
 Form1_ScrollBar1.LargeChange=100000;

 Form1.Show(); // Display the form
}
```

To add interactive controls, for example navigational controls, we just have to provide a few simple callbacks:

```
// Event callback called when the value of scrollbar1 changes
function Form1_ScrollBar1::Change() {
 position=Form1_ScrollBar1.Value;
 // Move the starting position of the bander object
 Form1_Bander1.SetStart(position);
}

// Absolute jump. Called when button 'Jump' has been clicked
function Form1_Jump::Click() {
 // Obtain the absolute position from the text field above
 // the jump button
 position=Position.Text;
 Form1_Bander1.SetStart(position); // Jump to that value
 Form1_ScrollBar1.Value=position;  // Update the scrollbar
}

// Handle the events from two small jump buttons appearing
// at the left and right of the scrollbar
// When jumping left…
function Form1_JumpLeft::Click() {
 // compute the new position
 position=position-jumpSize;
 if (position<0) position=0;
 // Update the bander object’s staring position
 Form1_Bander1.SetStart(position);
}

// A similar callback handles the right jump button click event
function Form1_JumpRight::Click() {
 position=position+jumpSize
 if (position>(seqlen-jumpSize)) position=maxrows-jumpSize;
 Form1_Bander1.SetStart(position);
}
```

The following final section of the code handles changes in the parameters we use to browse the chromosome (mersize and moving average length):

```
// Callback for event ‘Click’ of the SetMer button
function Form1_SetMer::Click() {
 // Get the new mersize from the text field above the button
 m=Math.round(Form1_Mersize.text);
 // Check for valid mersize
 if ((m>5) && (m<1001)) {
  merSize=m;
  // Update the mersize on the form and send a new mersize to
  // the server
  updatemersize();
 } else {
   // rollback to the previous value
  Form1_Mersize.text=merSize;
 }

}

// Handle ‘Click’ event for button SetW (set
// moving average window length)
function Form1_SetW::Click() {
 w=Math.round(Form1_Wsize.text);
 if ((w>10) && (w<1001)) {
  windowSize=w;
  updatewsize(); // Update the moving average operation
 } else {
   // Invalid value. Rollback to previously saved length
  Form1_Wsize.text=windowSize;
 }

}

// Utility functions

// Called when the mersize parameter changes
function updatemersize() {
 // Update the mersize field on the form
 Form1_Mersize.text=merSize;
 // Send the new mersize value to the server
 Form1_Bander1.Send(merBand,"mersize_"+merSize);
```

```
  // Trigger recalculation
 Form1_Bander1.SetStart(position);
}




// Called when the moving average window length changes

function updatewsize() {
 // Update the field on the form
 Form1_Wsize.text=windowSize;
 // Change the windowSize parameter for the AvgBand
 // operation. The old operation used to compute avgBand
 // is replaced by the new one
   Form1_Bander1.AvgBand(merBand,avgBand,windowSize);
 // The bander object automatically recomputes the new bands
  // and requests a form refresh
 }
```

The end result of the simple code presented above is the interactive application shown in the following picture:

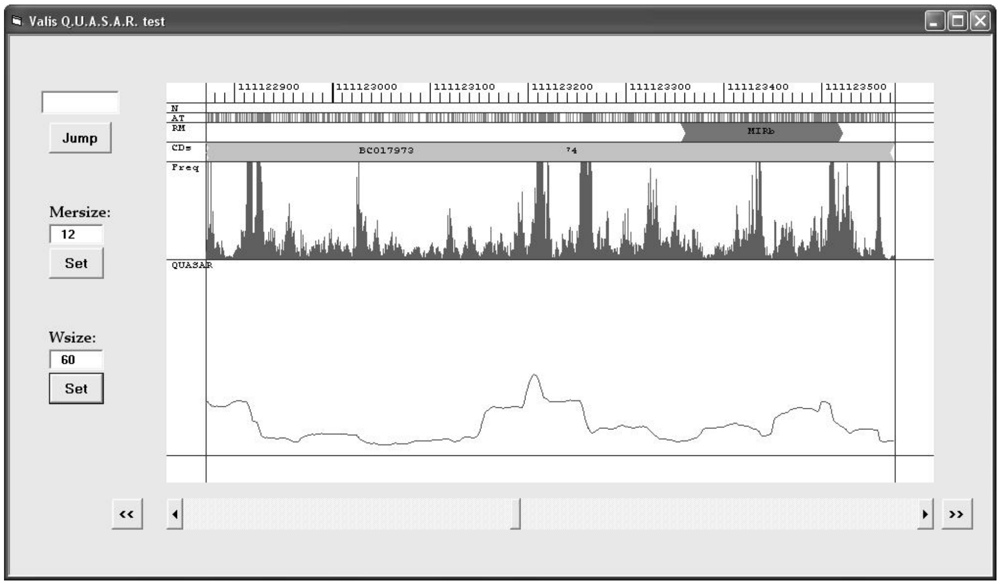


Figure 18. A VALIS genome viewer

5.1.3. Suffix array tools available within VALIS

To compare multiple genomes, and even to study the statistics of words occurring in a single genome (as in the above example), VALIS makes tools based on suffix arrays available.

A suffix array is basically a sorted list of all the suffixes of a string. If a suffix array is coupled with information about the longest common prefixes (LCP) of adjacent elements, exact substring match searches can be performed in O(P + log N) time, where P is the length of the substring to be found and N is the length of the query string.

Suffix arrays can obviously be constructed from suffix trees in linear time, but only a direct space-saving construction is of practical interest. Three [68][69][70] direct linear time construction algorithms have been described recently, but their space requirements for full-genome applications are also large. To find something suitable for such applications we have tested most of the O(N log N)-time algorithms. These can be divided in two classes: standard string sorting algorithms applied to all the suffixes of a string, whose best representative is the one due to Bentley and Sedgewick[76], and specialized construction algorithms, like the one proposed by Manber and Myers [66] and optimized by McIlroy[77], Sadakane[78], Larsson and Sadakane [80]; and finally the "Deep Shallow" algorithm proposed by Ferragina and Manzini[81] and the external memory construction by Gonnet and Baeza Yates [79] and Sadakane[78].

In our experiments with the largest human chromosomes (taken from the Golden Path genome database, freeze hg15), the Deep Shallow algorithm performed best, and so was adopted as our basic small to medium size suffix array construction tool.

We aim to deal with very large genomes, for example the human genome, and we aim to compare multiple large genomes even though the Deep Shallow algorithm cannot handle some of these desired datasets. To accomplish this level of performance we use a suffix array merge algorithm and distribute multiple chromosomes and genomes on a computer cluster. Each node of this cluster can keep suffix array data for a few full chromosomes in memory. Queries can be implemented in a parallel fashion using the MPI [92] library to provide real-time response to the exact match queries generated by our Bander object.

5.1.4. A historical survey of suffix tree and suffix array algorithms

Manber and Myers invented the suffix-array data structure as an alternative representation of suffix trees with lower space requirements. While suffix trees can be constructed in linear time and are ideal for the kind of string queries useful to genomics, they are unlikely to be used for searching strings larger than bacterial genomes.

Abouelhoda and others [67][71] have also proposed an extended representation of the suffix array for achieving bounds similar to suffix trees for supermaximal, maximal and tandem repeats of genomic strings.

To our surprise, we found that both Gonnet's and Sadakane's suffix array merge algorithms, very well known in the field, are actually incorrect. The suffix merge step used in this algorithm aims to build the suffix array for a very large string T of length n, by dividing it into sections of length n/m, and incrementally merging large suffix arrays and a smaller suffix array corresponding to a new section of the input string. The merge step deals with suffix arrays for two strings T_D and T_M of length n and m respectively to obtain the suffix array of the concatenated string T_D T_M.

The problem with this algorithm is the following: suppose that two suffixes of T_D, S_i and S_j appear in the large suffix array in a certain lexicographic order, say $S_i < S_j$; this order will be conserved by the merge step, but this could be incorrect, since the concatenation of T_M with S_i and S_j can potentially change the lexicographic order. A simple example would be with the two suffixes "AT" and "ATC" of T_D, and T_M starting with 'T'.

Sadakane's algorithm is very similar, but the counting phase is performed by traversing all the suffixes of T_M instead of T_D.

A simple solution to correct the errors in Gonnet's algorithm is to compute a suffix array of the last k characters of T_D concatenated with T_M instead of simply computing the suffix array of T_M, before applying the merge. In the counting stage, we can just ignore the last k characters of T_D. Finally in the merge step, we discharge all the suffixes of length k or less. A similar fix can be applied to Sadakane's version.

Of course the resulting suffix array can guarantee the accuracy of the lexicographic order only up to length k. But, for practical cases, k can be safely set to a few thousands. The extra work needed during the construction of the suffix array of a new section is negligible due to the O(m log m) complexity.

5.2 Application Example 2. Simpathica

Our second example shows how multiple scripting languages can be very useful in rapid construction of tools for bioinformatics and computational biology. To this end, we consider the Simpathica system developed by the

NYU Bioinformatics group as part of its DARPA Biocomp project[83].

The Simpathica/XSSYS system serves to construct, simulate and analyze the behavior of metabolic and regulatory networks. Analysis of pathways is done by formulating queries about their temporal evolution in an appropriate logic language. The system is logically divided into a front end and a simulation system, i.e. Simpathica proper and its analysis back-end XSSYS.

Biochemical pathways can be entered into the system either via the main Simpathica user interface or in a XML format. The system will then simulate the pathways entered and will produce trace objects. The XSSYS backend, written in Common Lisp, manipulates these traces (or traces produced by other simulation software or experiments) and evaluates Temporal Logic queries over them.

The Simpathica front end takes as input descriptions of metabolic and regulatory pathways constructed from a set of standard building blocks, which describe a repertoire of biochemical reactions, and can display these pathways in a graphical representation.

Simpathica then transforms this graph into an internal XML representation that can be also used for data exchange purposes. This internal representation consists of a set of Ordinary Differential Equations (ODEs) along with initial conditions. These ODEs are then translated into Octave (or Matlab) code, which performs the actual simulation by integrating the set of equations. The result of such a simulation is the trace object to be input into the XSSYS trace analysis system.

The output of the Simpathica front end consists of an XML model and a trace object produced indirectly by the chosen ODEs integrator (Octave in our case).

Once these are available, the XSSYS system takes the trace object and a temporal logic query and evaluates the truth-value of the query. If the query turns out to be false over the trace, XSSYS will also return a counterexample (in the form of a time index indicating a point where the trace falsifies the query).

The modules produced for the BIOCOMP project initially used the OOA Agent Architecture[84], to facilitate integration between modules written in different languages and produced by different groups. However the OAA architecture initially selected to speed up prototyping of the BIOCOMP system has many shortcomings:

1) In this architecture, each agent must register with “facilitator” written in Prolog. Agents cannot be started automatically when they are needed.

2) The facilitator serves to solve queries written in an “Interagent Communication Language” (ICL) that must be built by the clients. Writing clients and servers is not straightforward.

3) Performance issues arise for in-process calls; limits are imposed on message sizes; parameters of the ICL queries are mangled by the OAA infrastructure.

In VALIS we can do better. Once having assembled all the underlying building blocks needed, e.g. the XML parsers, graph viewers, ODE integrators, the XSSYS subsystem, it is possible to prototype in VALIS a system like Simpathica/XSSYS in a few days.

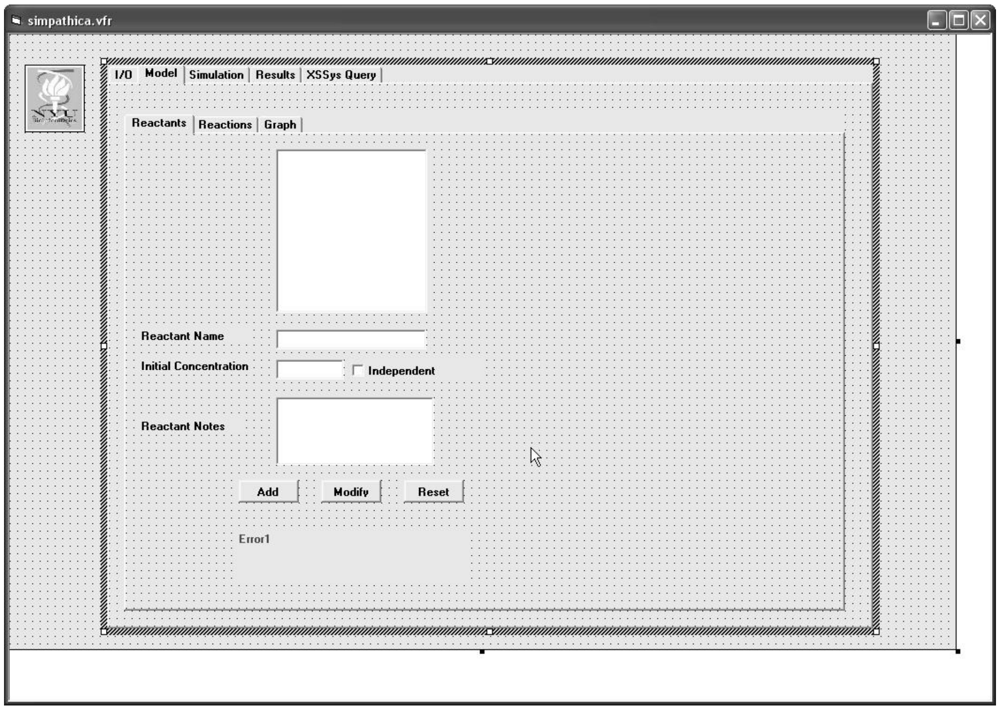


Figure 19. GUI design for Simpathica

A basic graphical user interface can be put together graphically in a VALIS form in a few minutes, since most of the widgets needed are standard controls of the form designer. The interface can be organized using multiple 'Tab' container widgets and using different tabs for I/O, the model editing widgets, the simulation pane, the graphical results of the simulation and the interface with the XSSYS subsystem. The figure seen above shows the tabs and the 'model editing' pane.

The only two graphical elements needed that are a bit unusual are a viewer for showing a graphical representation of the pathways, and a Graph component for presenting the simulation traces. For the first of these widgets, we use the Adobe SVG viewer. This is a standard ActiveX control and can render models written in the SVG language with zooming capabilities. To visualize traces we can use a simple Microsoft Chart control.

Since most of the internal data structures with which Simpathica/XSSYS works are based on XML, it is appropriate to use the versatile XML parser [93] from Microsoft to handle them. In VALIS this can be made available using just one code line:

```
xmlparser=CreateObject("Msxml2.DOMDocument.4.0");
```

A model of a pathway can be easily stored into XML files and retrieved using functionalities provided by the XML parser object. Once loaded and parsed this model is used to update the internal data structures (namely the ‘compounds’ and ‘reactions’ lists) and the corresponding graphical widgets.

We construct a graphical representation of the model from the internal XML representation and feed it to the SVG widget. We use the DOT language [94] (a general graph description language) as an intermediate language for this graphical representation. The DOT code is produced by applying a style sheet to the XML model. For example, a Repressilator model [97] will yield the following DOT code:

```
digraph G {  x0 [ label="pTetR"];  x1 [ label="pLambdaCI"];
x2 [ label="pLacI"];  x3 [ label="mtetR", style=filled];
x4 [ label="mlambdacI", style=filled];  x5 [ label="mlacI",
style=filled];
y6 [ shape=point] ;  y7 [ shape=point] ;  y8 [ shape=point] ;
y9 [ shape=point] ;  y10 [ shape=point] ;  y11 [ shape=point] ;
x3 -> y6 [weight=10, label="SIMPLE1", arrowhead=none] ;
y6 -> x0 [weight=10] ; x2 -> y6 [style=dotted] ;
x4 -> y7 [weight=10, label="SIMPLE2", arrowhead=none] ;
y7 -> x1 [weight=10] ; x0 -> y7 [style=dotted] ;
x5 -> y8 [weight=10, label="SIMPLE3", arrowhead=none] ;
y8 -> x2 [weight=10] ; x1 -> y8 [style=dotted] ;
x0 -> y9 [weight=10, label="OUT4"] ;
x1 -> y10 [weight=10, label="OUT5"] ;
x2 -> y11 [weight=10, label="OUT6"] ;  }
```

In this representation x0 trough x5 and y6 trough y11 are nodes (each one with certain properties, i.e. label, style etc.); a list of the edges follows.

The Graphviz system [94] can produce a variety of other graphical representations (among them SVG) once provided with models described in the DOT language. We reworked this system into an ActiveX control, which is then made available to VALIS.

```
// graph is the DOT description of the model.
// Use the graphviz control to obtain SVG code.
thexml=graphviz.DotToSvg(graph);

// ignore everything before <svg>
i=thexml.indexOf("<svg");
if (i>=0) thexml=thexml.slice(i);

// Parse the SVG model with the XML parser
xmlparser.loadXML(thexml);

// Copy it node-by-node to the Adobe SVG viewer
// using a script by Chris Bayes
// http://www.bayes.co.uk/xml/index.xml?/xml/utils/domtodom.xml
domtodom(xmlparser);
```

This program fragment yields the graph:

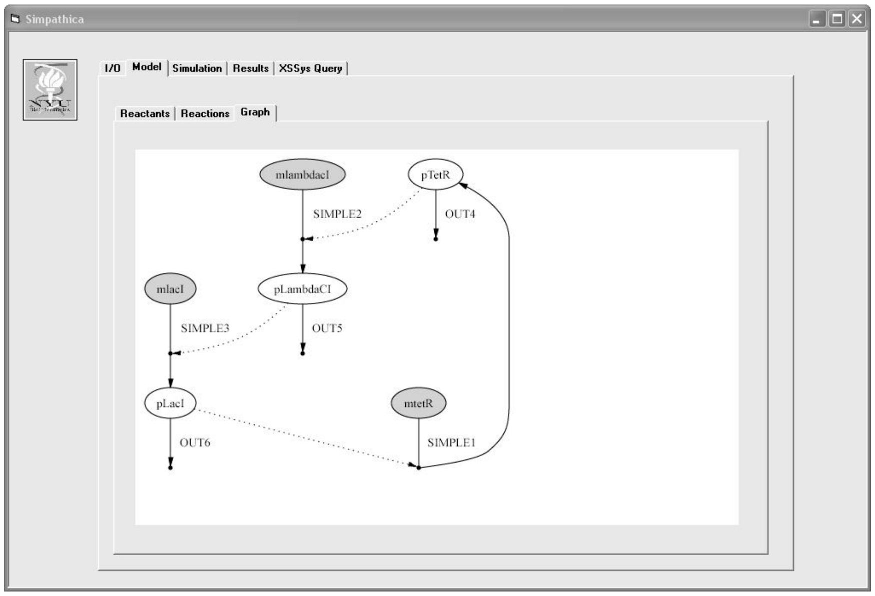


Figure 20. The SVG viewer embedded in a VALIS form

The SVG viewer allows the user to navigate through this graph.

The internal model used to produce the graph representation can be transformed into an intermediate representation suitable for the generation of a set of ODEs. This intermediate representation is obtained with the application of another XML style sheet:

```
 // Build
 xmlmap = buildXML();
 // Apply the xslmap style sheet. Result in xmlgraph
 xmlmap.transformNodeToObject(xslmap, xmlgraph);

 //validate the xmlgraph against the graph schema
 error = xmlgraph.validate();
 if (error != 0) {
  alert("Error in the parse of the graph file:\n" +
xmlgraph.parseError.reason);
  return;
 }
 // Obtain the python ODE definition

 script = xml2py(xmlgraph);
```

Without much difficulty, we can then dynamically produce some Python code (in the xml2py function above) with the step function for the integrator:

```
def pathway(X,t):
 xdot = []
 xdot.append(+1*(+1*X[3]**(1)*X[2]**(-1))-1*(+1*X[0]**(0.5)) )
 xdot.append(+1*(+1*X[4]**(1)*X[0]**(-1))-1*(+1*X[1]**(0.578151)) )
 xdot.append(+1*(+1*X[5]**(1)*X[1]**(-1))-1*(+1*X[2]**(0.5)) )
 xdot.append(0 )
 xdot.append(0 )
 xdot.append(0 )
 return xdot

initial = [0.01,0.2,0.01,0.2,0.2,0.2]
```

A Python ODE integrator (based on Numeric Python[95]) will integrate the ODEs generated as above.

```
from scipy import *
from scipy.integrate import *
from Numeric import *
```

```
time=0
intResult=0

def integrate(equations,simtime,simsteps):
 global time,intResult
 fromTime = 0
 toTime=simtime
 steps=simsteps
 precision = (toTime - fromTime) / float(steps)
 time = arange (fromTime, toTime, precision)
 intResult = 0

 exec("global intResult,time\n"+
   equations+"\nintResult = odeint(pathway, initial, time)")
```

This Python function is called directly from Javascript once the simulation is started:

```
// Call the Python integrator. Pass the equations and the simulation
// parameters
integrate(script,simtime,simsteps);
```

The 'integrate' Python function shown above continues by updating the Microsoft Chart control (see Figure 21) which has been added to the Simpathica form. The traces are then saved to a temporary file

```
 cols=intResult.shape[1]
 rows=intResult.shape[0]

 # Set char type to line plot
 Form1_Chart.chartType=3
 # Customize the grid
 Form1_Chart.Plot.Axis(0).AxisGrid.MajorPen.Style=0
 Form1_Chart.Plot.Axis(1).AxisGrid.MajorPen.Style=0
 Form1_Chart.Plot.Axis(2).AxisGrid.MajorPen.Style=0
 Form1_Chart.Plot.Axis(0).CategoryScale.DivisionsPerLabel=rows/10
 Form1_Chart.Plot.Axis(0).CategoryScale.DivisionsPerTick=rows/10

 Form1_Chart.ColumnCount=cols
 Form1_Chart.RowCount=rows
 for col in xrange(0, cols):
  Form1_Chart.Column=col+1
  for row in xrange(0,rows):
   Form1_Chart.Row=row+1
   Form1_Chart.Data=intResult[row,col]

 for row in xrange(0,rows):
   Form1_Chart.Row=row+1
 global time,intResult
```

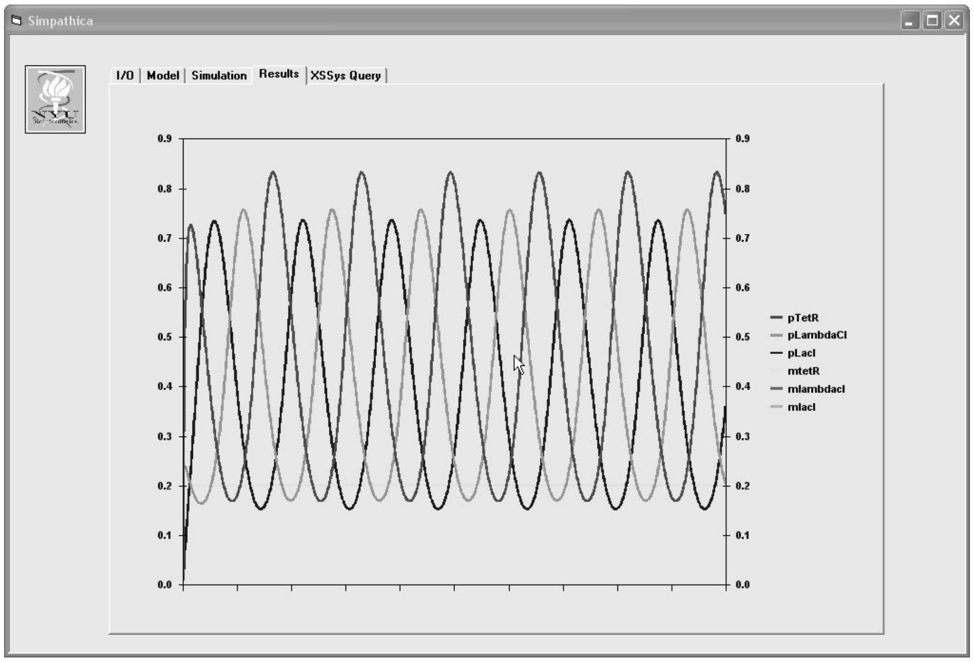
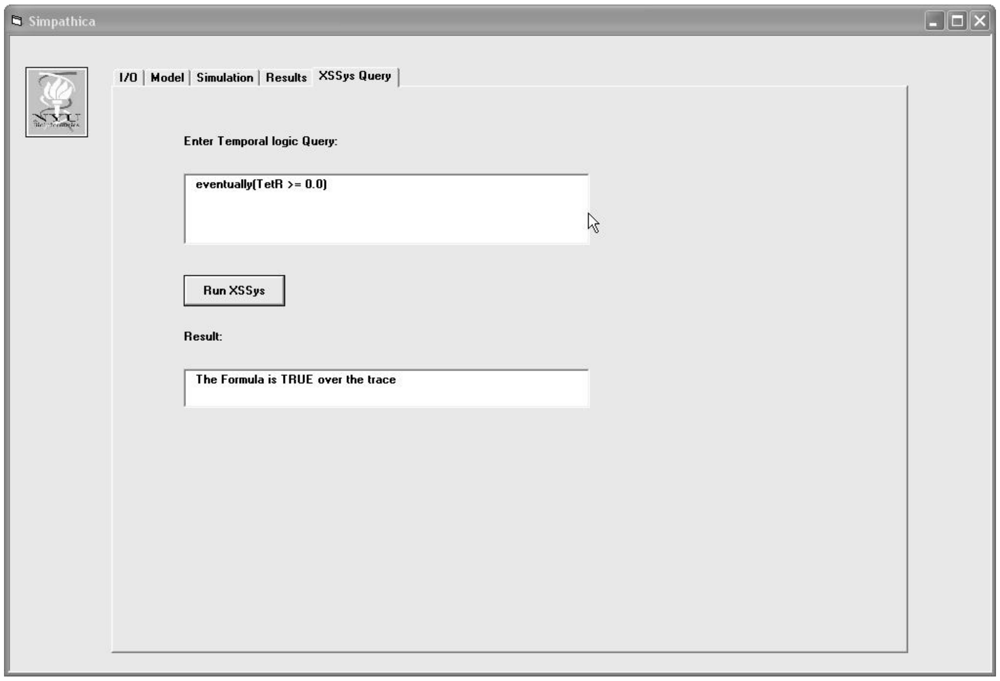


Figure 21. Simulation of a repressilator

The XSSYS query event (generated by the button in the 'XSSYS Query' pane shown in Figure 22) can be handled by some JavaScript:

```
 function Form1_RunXSSys::Click()
{

 Clear();
 // Clear the result text widget
 Form1_TLResult.text="";
 // Get the temporal logic query string
 query=Form1_TLQuery.text;
 // Call the LISP subsystem
 result= xssysquery ( tempfile, query);

  switch (result) {
  case -1:
   result="Error (most likely a syntax error)!";
   break;
  case 0:
   result="The Formula is FALSE over the trace";
   break;
  case 1:
   result="The Formula is TRUE over the trace";
   break;
  case 2:
   result="The Formula is always TRUE";
   break;
  case 3:
   result="The Formula is sometimes TRUE";
   break;
  }
 // Display the result
 Form1_TLResult.text=result;
}
```



eventually(TetR >= 0.0)

The Formula is TRUE over the trace

Figure 22. The XSSYS Query pane

That will in turn call the front end to the XSSYS system in Common Lisp:

```
(defun xssysquery (a b)
(load-trace a)
  (analyze-this b)
  )


(defun load-trace (filename)
 (if (probe-file filename)
       (progn
         (setf xssys::*the-current-trace*
               (xssys:load-trace filename :default))
  xssys::*the-current-trace*)
-1))
```

Finally, the code that handles events from the Forms and customizes the interface can readily be written in JavaScript.

5.3 Summary

In this chapter we have shown two sample VALIS applications designed to let biologists explore complex genomes and biochemical pathways. VALIS allows these applications to be constructed rapidly and connected together seamlessly. Our Simpathica implementation illustrates the technique of multiple language prototyping. The availability of many powerful scripting languages and numerous public domain libraries and components within VALIS support a substantially improved rapid prototyping capability for computational and systems biology.

Chapter 6

Conclusion

Bioinformatics is a challenging application area for computer science, and in particular for programming language designers. We have seen that scripting languages, graphical user interfaces and database systems play a major role in prototyping and developing applications in bioinformatics.

In this thesis we have presented a novel system that allows seamless integration between scripts written in different programming languages, without the use of IDL descriptions or Object broker wrappers and that works with today's languages and compilers. We have described a new data storage system that has marked advantages when dealing with unstructured and unbounded data, common in scientific fields and bioinformatics. This new programming language design environment gives us the ability to prototype systems with full graphical user interfaces rapidly, using standard ActiveX controls. We have also described a battery of widgets and algorithms, which permit rapid prototyping of large-scale interactive genomic applications within VALIS

6.1 Other systems comparable to VALIS

The high degree of standardization defined by Microsoft .NET (and the public domain MONO effort) lays out a path to the future of large-scale software which all language designers will have to deal with in the coming period. To reach this future will take some time, because it will take many years to convert the great mix of native code currently existent to managed c++ code conforming to standards like that defined by the Microsoft CLR (Common Language Runtime). Moreover, early attempts by Microsoft and ActiveState developers to target the CLR from Python and Perl have failed[85], suggesting that the CLR it is not yet fully adequate to support Python and other programming languages having similar semantics. To address this issue, Microsoft is currently considering a revision of the CLR.

6.2 Porting VALIS to other platforms

We have successfully run VALIS (both its non-GUI and the GUI portions) under WINE in the Linux operating system (using an Intel processor). This demonstrates the feasibility of supporting ActiveX designers and controls by API emulation in non-windows environments (see also the crossover plugin [87]). This is definitely not the best solution, since in environments such as the Apple MacOSX operating system an Intel processor would have to be emulated.

Most of the interpreters used in our Windows-based version of VALIS are already supported in other environments and operating systems. Therefore, by providing a few COM libraries, we can hope to port the non-GUI parts of VALIS to other environments by a simple recompilation. All the COM libraries needed to implement this much are already available within the public domain WINE project[86]. (The Winelib component of WINE already offers a cross-platform version of these libraries.) Moreover, cross-platform alternatives to COM, like XPCOM [96] developed by Netscape, already exist.

We have seen in section 4.6 that most cross-platform GUI design toolkits currently available lack the scripting capabilities of ActiveX controls (OLE Automation) and that many do not support the creation of non-standard widget sets. The GUI building capabilities and the graphical widgets of VALIS would then need to be supported by a cross-platform GUI component architecture with characteristics similar to that offered by ActiveX controls.

A possible solution would be to use a cross-platform widget set like QT or GTK for rendering and to define interfaces similar to OLE and OLE Automation that VALIS graphical widget developers must use. (The KDE component architecture could be extended to offer such interfaces.)

Another approach would be to use Mozilla XUL (XML based Used interface Language [98]) to design cross platform graphical user interfaces in VALIS. Rapid application development using Mozilla seems promising (see [98]), but currently offers no RAD environments and only JavaScript (or C++) is available to code application logic. But Mozilla might be extended to support interfaces similar to ActiveX scripting instead of XPCONNECT to gain access to other scripting engines.

Bibliography

[1] *The VALIS project*, http://bioinformatics.cims.nyu.edu/Projects/valis

[2] *The BioPerl Toolkit*, http://www.bioperl.org, 2003

[3] *The Bioperl toolkit: Perl modules for the life sciences,* Genome Res. 12(10): 1611-1618, October 2002

[4] Guido van Rossum, *Extending and embedding the Python interpreter*. Amsterdam, Stichting Mathematisch Centrum, (1995), also at http://www.python.org/doc/ext/ext.html, 2003

[5] Marina Chicurel, *Bioinformatics: Bringing it all together*, Nature 419, 751–757, 2002

[6] Boudewijn Rempt, *GUI Programming with Python: QT Edition*, Opendocs Llc (January 2002), also at http://www.opendocs.org/pyq, 2003

[7] Andy Tai, *The GUI Tookit, Framework Page*, http://www.atai.org/guitool, 2003

[8] John Ousterhout, *Scripting: Higher Level Programming for the 21st Century,* IEEE Computer 31,23-30,March 1998

[9] Kevin J. Sulluvan, John C. Knight, *Experience Assessing an Architectural Approach to large-scale Systematic Reuse*, Proceedings of

the 18th International Conference on Software Engineering, Berlin, March 1996,220-229, also at http://citeseer.nj.nec.com/43110.html, 2003

[10] Trolltech, *QSA Overview*, http://www.trolltech.com/products/qsa/, 2003

[11] *Haskellscript*, http://www.haskell.org/haskellscript/index.html, 2003

[12] J.T. Schwartz and Salvatore Paxia,*The SETL Database package*, http://bioinformatics.cims.nyu.edu/Projects/setl, 2003

[13] Kenshi Hayashi and Nobuo Munakata. *Basically musical*. Nature 310, 96, July 1984

[14] Alexandros Biliris. *An efficient database storage structure for large dynamic objects*. In Proceedings of the International Conference on Data Engineering, pages 301-308,1992

[15] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. *Object and file management in the EXODUS extensible database system*. In Proceedings of the Conference on Very Large Data Bases (VLDB), pages 91-100, 1986

[16] Tobin J. Lehman and Bruce G. Lindsay. *The starburst long field manager*. In Proceedings of the Conference on Very Large Data Bases (VLDB), pages 375-383, 1989

[17] Orfali, R., Harkey, D., and Edwards, J. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons, 1995

[18] L. Wall and R.L. Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, 1992

[19] G. Van Rossum, *Python Language website*, http://www.python.org, 2003

[20] John Ousterhout. *The TCL/TK website*, http://www.tcl.tk, 2003

[21] Mike Cowlishaw. *The Rexx language*, http://www2.hursley.ibm.com/rexx, 2003

[22] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[23] Leijen, D., and Hook, E. M. J. *Haskell as an automation controller*. In Third International School on Advanced Functional Programming (AFP'98) (Braga, Portugal), Lecture Notes in Computer Science, 1608, pp. 268-289, Springer Verlag, 1999

[24] D.M. Beazley. *SWIG and Automated C/C++ Scripting Extensions*, Dr. Dobb's Journal, No. 282, February, 1998. p. 30-36

[25] Nat Goodman. *BioPerl: Myth vs. Reality*, O'Really bioinformatics conference, 2003

[26] Tcl Developer Xchange. *Advocacy Page*, http://www.tcl.tk/advocacy/, 2003

[27] M. Antoniotti, F. C. Park, A. Policriti, N. Ugel, and B. Mishra. *Foundations of a Query and Simulation System for the Modeling of Biochemical and Biological Processes*. In Proc. of the Pacific Symposium of Biocomputing (PSB'03), 116-127, 2003

[28] W. Kirk Snyder. *The SETL2 programming language*. Technical Report 490, Courant Institute of Mathematical Sciences, New York University, January 1990. also at http://www.bioinformatics.nyu.edu/Projects/setl/report.pdf

[29] W. Kirk Snyder. *The SETL2 programming language: Update On Current Developments*, Courant Institute of Mathematical Sciences, New York University, 1991, http://www.bioinformatics.nyu.edu/Projects/setl/update.pdf

[30] David J. Bacon. *SETL for Internet Data Processing*. Ph.D. Thesis, Courant Institute of Mathematical Sciences, New York University, 2000

[31] J.T. Schwartz, Salvatore Paxia, *Programming in SETL*, (Draft in progress), http://www.settheory.com/, 2004

[32] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes in C, 2d ed*. Cambridge University Press, New York, NY, 1986

[33] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[34] Object Management Group CORBA. *OMG website*, http://www.omg.org, 2003

[35] Jean-Claude Wippler, *Minotaur*, http://www.equi4.com/minotaur, 2003

[36] Steele, G. L. Jr., *Common Lisp, The language, Second Edition*, Digital Press, 1990.

[37] Robert Gentleman and Ross Ihaka, *The R Project for Statistical Computing*, http://www.R-project.org, 2003

[38] Venkatesh Mysore and Salvatore Paxia, *R for VALIS*, http://www.bioinformatics.nyu.edu/~mpvenkat/R-1.7.0/R_For_Valis.htm, 2003

[39] Marco Antoniotti, *CLAXS - ActiveX Scripting Engine for Common Lisp*, http://www.bioinformatics.nyu.edu/~marcoxa/, 2003

[40] GenBank, *Nucleic Acids Research* 30(1) (2002) 17-20

[41] Jim Kent et al., *UCSC Genome Browser*, http://genome.ucsc.edu/, 2003

[42] Ahlberg, Christopher and Wistrand, Erik, *IVEE: An information visualization and exploration environment*. IEEE Information Visualization ’95, IEEE Computer Press, Los Alamitos, CA (1995), 66-73

[43] Spotfire Inc, *Spotfire*, http://www.spotfire.com, 2003

[44] Brockschmidt, K., *Inside OLE, Second Edition*, Microsoft Press, Redmond WA, 1995

[45] Microsoft, *The ActiveX Control Pad*, http://download.com.com/3000-2406-861343.html, 2003

[46] Microsoft MSDN, *ActiveX Designer Programmer's Reference*, April 1999

[47] *Active State, Visual Perl, Visual Python, Visual TCL*, http://www.activestate.com/Products/, 2003

[48] J. Smart, *The wxWindows Cross-Platform GUI Library*, http://www.wxwindows.org/, 2003

[49] Riaan Booysen, *Boa Constructor*, http://boa-constructor.sourceforge.net/, 2003

[50] *The K Desktop Environment*, http://www.kde.org/, 2003

[51] Scott Wheeler, *KDE Scripting with DCOP*, Linux Magazine, 46-48, November 2003

[52] Richard Moore, *The KJSEmbed Library*, http://xmelegance.org/kjsembed/, 2003

[53] M. Olson, K. Bostic, and M. Seltzer. *Berkeley DB*. In Proc. Of USENIX Technical Conference, FREENIX Track, Monterey, CA, June 1999.

[54] D. E. Knuth. *The Art of Computer Programming, Volume 3. Sorting and Searching*. Addison-Wesley, Reading, MA, 1973

[55] J. Srivastava, J. S. E. Tan, and V. Y. Lum, *TBSAM: An Access Method for Efficient Processing of Statistical Queries*, IEEE Transactions on Knowledge and Data Engineering, 1:4, pp. 414-423, 1989.

[56] David McCusker, *IronDoc*, http://www.treedragon.com/ged/fe/fe.htm, 2003

[57] Yukihiro Matsumoto, *Ruby*, http://www.ruby-lang.org/en/, 2003

[58] *The GNU Scientific Library*, http://www.gnu.org/software/gsl/, 2003

[59] Adobe, *Scalable Vector Graphics (SVG) Viewer*, http://www.adobe.com/svg/main.html, 2003

[60] Delcher, A. L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O. and Salzberg, S.L. *Alignment of whole genomes*. Nucl. Acids Res., 27, 2369-76, 1999

[61] Gusfield, D, *Algorithms on strings, trees, and sequences*. Cambridge University Press, New York, NY, 1997

[62] Kurtz, S. and Schleiermacher, C. *REPuter: fast computation of maximal repeats in complete genomes*. Bioinformatics 15(5):426-427, 1999

[63] Jeong-Hyeon Choi, Hwan-Gue Cho, *Analysis of Common k-mers for Whole Genome Sequences Using SSB-Tree*, Genome Informatics 13: 30-41, 2002

[64] P. Ferragina and R. Grossi. *The String B-Tree: A New Data Structure for String Search in External Memory and its Applications*. Journal of the ACM, 46(2), 1999, 236-280

[65] S. Burkhardt, A. Crauser, H-P. Lenhof, E. Rivals, P. Ferragina, M. Vingron, *Q-gram based database searching using a suffix array (QUASAR)*. 3rd Ann. International Conference on Computational Molecular Biology (RECOMB 99), 77-83, Lyon 11-14 April 1999

[66] U. Manber and G. Myers. *Suffix arrays: a new method for online string searches*. Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, 1990, 319—327

[67] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. *The enhanced sux array and its applications to genome analysis*. In Proc. 2nd Workshop on Algorithms in Bioinformatics, volume 2452 of LNCS, pages 449-463. Springer, 2002.

[68] K. Karkkainen and P. Sanders, *Simpler linear work suffix array construction*, 14th Annual Symposium, Combinatorial Pattern Matching (2003), LNCS 2676, Springer, pp. 55-69

[69] P. Ko and S. Aluru, *Space-efficient Linear time construction of suffix arrays*, 14th Annual Symposium, Combinatorial Pattern Matching (2003), LNCS 2676, Springer, pp. 200-210

[70] D.K. Kim, J. S. Sim, H. Park, and K. Park. *Linear-time construction of suffix arrays*, 14th Annual Symposium, Combinatorial Pattern Matching, 2003, LNCS 2676, Springer, pp. 186-199

[71] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. *Optimal exact string matching based on suffix arrays*. In International symposium on String Processing and Information Retrieval, pages 31-43, IEEE, 2002

[72] J.M. Hellerstein, J.F. Naughton and A. Pfeffer, *Generalized Search Trees for Database Systems*, Proc. 21st VLDB, Zurich, Switzerland, Sep. 1995, 562-573

[73] Oleg Bartunov and Teodor Sigaev, *GiST for Postgres SQL*, http://www.sai.msu.su/~megera/postgres/gist, 2003

[74] Joseph M. Hellerstein and Avi Pfeffer, *The RD-Tree: An index structure for sets*, Technical Report #1252, University of Wisconsin at Madison, October 1994

[75] M. Stonebraker & G. Kemnitz, *The POSTGRES next generation database management system*, Communications of the ACM, vol. 34, no. 10 (1991): 78-92. also http://www.postgresql.org, 2003

[76] J. L. Bentley and R. Sedgewick. *Fast algorithms for sorting and searching strings*. In Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 360-369, 1997

[77] P. M. McIlroy, M. Douglas McIlroy. *Suffix Sort*, http://cm.bell-labs.com/cm/cs/who/doug/ssort.c, 2003

[78] Kunihiko Sadakane, *Unifying Text Search and Compression Suffix Sorting*, Block Sorting and Suffix Arrays, Ph.D. Thesis, 2000, University of Tokio

[79] G.H. Gonnet, R. Baeza-Yates, and T. Snider. *New Indices for Text: PAT trees and PAT arrays,* In W. Frakes and R. Baeza-Yates, editors, Information Retrieval: Algorithms and Data Structures, Chapter 5, pages 66-82. Prentice-Hall, 1992

[80] Larsson, N. Jesper and Sadakane, Kunihiko, *Faster Suffix Sorting*, http://citeseer.nj.nec.com/larsson99faster.html, 2003

[81] G. Manzini, P. Ferragina, *Engineering a Lightweight Suffix Array Construction Algorithm*, Proc. 10th European Symposium on Algorithms (ESA '02).Rome, Italy, 2002. Springer Verlag Lecture Notes in Computer Science n. 2461, 698-710

[82] M. Antoniotti, A. Policriti, N. Ugel, B. Mishra, *Reasoning about Biochemical Processes*, Cell Biochemistry and Biophysics, 38:271-286, 2003.

[83] DARPA *Biocomp*, http://www.biospice.org, 2003

[84] Cheyer, Adam and Martin, David, *The Open Agent Architecture*, Journal of Autonomous Agents and Multi-Agent Systems, vol. 4 , no. 1, pp. 143-148, March 2001

[85] Mark Hammond, *Python for .Net: Lessons learned,* http://www.activestate.com/Corporate/Initiatives/NET/Python_for_.NET_whitepaper.pdf, 2003

[86] *The WINE Project*, http://www.winehq.com, 2003

[87] CodeWeavers, *CrossOver Plugin*, http://www.codeweavers.com/site/products/, 2003

[88] William R. Cook and Warren H. Harris, *The Open Scripting Architecture: Automating, Integrating and Customizing Applications*, http://citeseer.nj.nec.com/555351.html, 2003

[89] Buxton, W., Lamb, M. R., Sherman, D. & Smith, K. C., *Towards a Comprehensive User Interface Management System. Computer Graphics* 17(3). 31-38, 1983

[90] Krishna Sankar, *Internet Explorer Plug-In and ActiveX Companion*, QUE, 1997, also at http://sunsite.iisc.ernet.in/virlib/activex/iecomp/ewtoc.html, 2003

[91] Microsoft Corporation, *Microsoft Visual Basic 6.0 Programmer's Guide*, Microsoft Press, 1998

[92] *The Message Passing Interface (MPI) standard*, http://www-unix.mcs.anl.gov/mpi/, 2003

[93] Microsoft, *Microsoft XML Parser*, http://msdn.microsoft.com/xml/, 2004

[94] AT&T Research, *Graphviz - Open source graph drawing software*, http://www.research.att.com/sw/tools/graphviz/, 2003

[95] *Numeric Python*, http://www.pfdubois.com/numpy/, 2003

[96] Mozilla Project, *XPCOM*, http://www.mozilla.org/projects/xpcom/, 2003

[97] Elowitz MB, Leibler S, *A synthetic oscillatory network of transcriptional regulators*. Nature.403 : 335 – 338 (2000)

[98] Nigel McFarlane, *Rapid Application Development with Mozilla*, Prentice Hall (2003)