Symbolic Computation in Java: an Appraisement*

Laurent Bernardin[1] Bruce Char[2] Erich Kaltofen[3]

[1]Institut für Wissenschaftliches Rechnen, Eidgenössische Technische Hochschule
Zurich, Switzerland; Laurent.Bernardin@inf.ethz.ch
http://www.inf.ethz.ch/personal/bernardi/

[2]Department of Mathematics and Computer Science, Drexel University
Philadelphia, Pennsylvania 19104; bchar@mcs.drexel.edu

[3]Department of Mathematics, North Carolina State University
Raleigh, North Carolina 27695-8205; kaltofen@math.ncsu.edu
http://www.math.ncsu.edu/~kaltofen

1 Introduction

Java, first developed as a programming interface for web browsers, is being pushed by a consortium of companies as a high level programming language for general programming. Sun Microsystems provides, in addition to a compiler into byte code for an interpreter (the Java virtual machine), a multitude of libraries, most notably an object hierarchy for building graphical user interfaces: the Abstract Windowing Toolkit and the Swing components from the Java Foundation Classes. Internet browsers contain Java virtual machines for interpreting byte code of Java programs that are embedded into Internet documents as applets. Java defines a standard framework for multi-threaded execution and for message passing via serialization and socket/datagram protocols. Java assists component composition in two ways. Java objects can discover how to invoke other Java objects at run-time through a process called reflection. Java also supports programming conventions (collectively referred to as "Java Beans") for event-driven inter-component operation. The two together allow tools such as Java Studio [26] to provide convenient visual programming methods of connecting up Java software components. In short, Java is being vigorously developed and we ask the natural question whether Java is suited for symbolic computation and whether our discipline should take advantage of the plethora of freely available software.

This article discusses Java as a symbolic computation development tool, expanding on the pioneering efforts of other researchers [39, 9]. We investigate if Java can compete with C++, or Maple/Mathematica/Axiom for efficient implementation of algebraic algorithms. We suggest uses of many of Java's innovative features for symbolic software design problems. We discuss the suitability of Java for generic programming, a methodology whose origins actually are in computer algebra. We believe the software component approach is required for designing modern systems that include computer algebra. We describe ways in which Java can be used to adapt legacy software into components, and we present our ideas how component interfaces can be structured.

In section 2 we state our requirements for symbolic computation components without making reference to a particular programming style or language. How well our requirements are met by Java and other languages is discussed in section 3. Our personal experience with Java, including efficiency measurements, is documented in section 4. We finally appraise Java as an implementation language for symbolic computation software in section 5.

2 Symbolic Computation Component Requirements

A middle-ware paradigm for making new algorithmic results available to the non-expert end-user of our technology is recognized in [17]. Suppose a symbolic computation researcher wishes to implement a new algorithm for sparse linear algebra. The program should not start from scratch and implement basic long integer and polynomial arithmetic, as there are highly tuned libraries such as GnuMP [23], SACLIB 2.1 [13], and NTL 3.1 [42] available for those tasks. Moreover the program need not develop a distinct user interface or command language but instead be callable from all familiar platforms such as Maple, Mathematica, and an Internet browser. Thus, the implementation ends up in the middle of existing symbolic computation software.

We loosely distinguish between the *plug-and-play* and *generic programming* component design [29]. The plug-and-play methodology requires an application program interface of the finished new package to the users' actual platform: for instance, a user should be able to launch a sparse matrix

*This material is based on work supported in part by the National Science Foundation under Grant No. CCR-9527130 (Bruce Char) and Grant No. CCR-9712267 (Erich Kaltofen).

computation for the new package from within her MuPAD environment. Generic programming, on the other hand, makes the new package independent of, e.g., the underlying polynomial arithmetic that is employed. The implementation has plugs into which certain libraries have to be put. Libraries are plugged into the new package, either using a common object interface standard or by placing an adaptor program between the package and the plug-in. The purpose of plug-and-play is wide dissemination while the purpose of generic programming is reusability of code in a different setting; we shall not be dogmatic about this distinction.

In computer algebra, the study of generic programming techniques and the requirement of plug-and-play components is not new. Common object interfaces for algebraic domains such as polynomial rings go back to [37, 2, 27, 35]. Standards for serialization of mathematical objects are the subject of the OpenMath consortium [1, 15] and have been proposed in [4] and are used in Mathematica's MathLink tool. Vertical integration of components was already posed as a problem in 1974 [34]. There now is a sizable community of researchers investigating the construction of so-called problem solving and integrated development environments [32, 25].

We shall list of what we feel are important ingredients of component interfaces.

Objects as functions The view of mathematical objects as functions is powerful. For example, a matrix is viewed as a "vector-in, vector-out" matrix-times-vector function and many linear algebra problems can be solved very efficiently with this black-box representation [21, and the references given there]. The difficulty is to transmit such a black-box matrix across the components, which may be constructed in different programming languages. The programming content dictionary of the OpenMath standard addresses this question to some degree. We remark that the problem of function distribution is not unique to symbolic computation. It has been seen to occur when computing the optimum of a mathematical function in parallel.

Parallelism Packages and libraries should allow for parallel/distributed execution. The question remains if there is a standard of high-level parallelization directives. Parallelism is important for symbolic computation because it yields a fantastic increase in computer memory and for certain algorithms in processing power.

Storage management Computing languages like Lisp, Java, and Maple perform automatic garbage collection; C and C++ requires the programmer to explicitly manage storage. The challenge for component technology is to combine the different memory models in a single computation.

Exceptions An accepted way of handling exceptions is the try/throw/catch mechanism of C++ and Java. Clearly, the interface between components must include exception handling. One form of exceptions are checkpoints, where the current state of the computation is serialized and saved.

Legacy code It is tempting to start with the bottom layer of a symbolic computation system because all functionality can be custom-made. However, such design leads to monolithic systems which must keep pace with new algorithmic advances. We believe in the utilization of existing software and the formalization of interfaces. There remains the problem of legacy code, often written in a different programming language. Sometimes one can gain efficiency by so-called dirty tricks, which can be formalized as traits [38] in the interface.

Algorithmic shortcuts An important paradigm in symbolic computation is the evaluation/interpolation technique. Some libraries, such as NTL, internally use Fourier points and the fast Fourier transform for efficiency. The component interface can make Fourier evaluation/interpolation available to a caller by again defining a trait for library interface.

In conclusion, we observe that interchangeable components are not restricted to libraries for symbolic computation algorithms. Clearly, a graphical user interface that implements interactive usage of a set of library routines, a garbage collector, a message passing distribution mechanism, a graphics visualization tool, a statistical analysis package, or an online documentation browser, all might be components to which a middle-ware package is connected.

3 A Comparison of Java to Traditional Languages

We now discuss how well Java meets the requirements of section 2 for building symbolic computation components in comparison with C++, Lisp, Smalltalk, Maple, and Mathematica. We will also take into account the A♯/Aldor [43] compiler.

3.1 Generic programming

A main goal of generic programming is the ability to write algorithms with variable types: for example, a Gröbner basis procedure is parameterized with the coefficient field and can be instantiated for a multitude of different fields. In [18] we distinguish three implementations of genericity:

1. the field operations that define the common interface (addition, division, equality tests, etc.) are inline-compiled into the procedure for a selected field;
2. they are retrieved via a function pointer in the stub for the field object (an abstract base class);
3. they are linked into the procedure at load time.

The template feature of C++ is designed for the first instantiation method. A large example is C++'s standard template library, and we have used this approach in FoxBox [17]. Dynamic evaluation is used in the Scratchpad II/Axiom system and in Maple's Domain package. C++ has virtual class member functions for this purpose, which have been adopted by Java under the name of abstract methods. A main point in [18] is that in C++ one can write a generic program so that the choice of the three instantiation methods for type parameters is left to the users.

Originally Java was thought as a computer architecture-independent programming system. It adopted Smalltalk's approach of an interpreted intermediate instruction set, namely, Java's byte code and virtual machine. With automatic garbage collection its execution mechanism resembles that of Maple, Mathematica, and Lisp. However, even

the use of C++'s virtual member functions is discouraged for reason of efficiency. Our experience with Maple and Lisp certainly warns us that Java byte code may suffer even worse inefficiencies. It is expected, nevertheless, that Java programs will be compiled into native machine instructions. Compilers may, for reason of improving the efficiency, dynamically adapt the produced code to the present run-time environment. In section 4.1 we give some timings for "just-in-time" compiled byte code for polynomial arithmetic. We remark that the intermediate byte code mirrors the role of the intermediate C code of some C++ compilers. The separation of architecture specific and architecture independent optimization seems not to introduce inefficiencies.

In addition to abstract methods, the Java language promotes the concept of an interface type. A class object that implements all functions of the interface can be assigned to a variable of the interface's type. The reverse of implementing an interface is reminiscent of a view of a concrete class as an abstract domain [2]. In fact, the Collections package of Java 1.2, which provides the functionality of C++'s standard template library, has largely replaced the template type parameters by interfaces.* An abstract field, for example, can be defined in Java as an interface. The concrete class for a finite field of 2^n elements can now implement both the interface of an abstract field as well as the interface of a vector space over the field with 2 elements; this is the only form of multiple inheritance that Java permits.

The lack of C++-style templates in Java has prompted the development of a precompiler for a generic Java language extension [8]. **But are templates really needed?** We observe that C++'s template expansion mechanism can be used as a precompiler. Blitz++'s [7] expression templates and Athapascan-1's [3] and POOMA's [40] parallelization types are a powerful demonstration of the utility of what is now called template meta-programming. For example, by overloading of the assignment operator the template expansion mechanism can be programmed so as to group assignments left-to-right. Overloading of operators has been used in MITMatlab [24] to parallelize legacy Matlab code. From the above examples of use of the template expansion mechanism we conclude that even more sophisticated manipulation of C++ programs is possible. Java does not allow the overloading of operators and makes such after-the-fact semantic changes impossible. We should add, however, that the intricacies of the resulting C++ programs are currently beyond the ability of most C++ compilers. Ultimately, template meta-programming may be too complicated to be of lasting value to even the system-building programmer.

3.2 Plug-and-play

Java, as with any modern programming environment, is downward compatible meaning that programs written in C or in other programming languages can be invoked by a Java program via its so-called native method mechanism. A Java run-time environment decides if a native code library is available and chooses between a Java or a more efficient native implementation for a given subtask. We have done some experiments with integrating GnuMP in this fashion.

A more serious question is upward compatibility: **how does another program invoke a Java component?** In [39] it is proposed that REDUCE can execute Java programs in a similar manner as browsers do: a Java virtual machine, perhaps restricted to the byte codes that are of importance in the symbolic computation context, is embedded in REDUCE's Java interface. Their approach is attractive for another reason: black box mathematical objects can be transmitted in byte code to this interface. In [29] we have noted that dynamic descriptions of mathematical expressions are not only useful for black box algorithms, but can be used to generate a static data structure for the expression. An example is the Hilbert-like 100×100 matrix $[1/(i^2+j-1)]_{1\le i,j\le 100}$, which should not be transmitted as 10,000 rational numbers.

An alternative approach is facilitated by Java's networking capability. The Java and non-Java components run as independent processes and communicate via a socket protocol or a pipe. No virtual machine needs to be constructed. We have chosen the socket approach for connecting Maple to the distributed symbolic computation tool [11] and to FoxBox [17]. Maple does not yet provide built-in networking procedures and the connection was implemented with Maple's insecure "system" command, which executes any accessible local program. Schreiner has recently distributed Maple with a Java-based scheduler [41]. Java, on the other hand, provides a complete set of network programming tools.

Java assists the programmer in the design of component interfaces. First, so-called marker interfaces can can be used to determine if a component has a certain trait, such as Fourier point evaluation/interpolation. The marker interface to a foreign library is placed in the Java adaptor code. Second, Java code can be probed for type-signature information, a process that Java calls reflection. This is similar to the notion of plug-and-play hardware, where the operating system can determine the properties of a device by a signal protocol. Reflection allows for adaptive interfaces and interactive component assembly and thus relaxes the need for rigid standards.

3.3 Parallelism

Threads With multi-CPU PCs becoming common place, multi-threading becomes an increasingly important tool for expressing fine-grained parallelism. Three features characterize an efficient multi-threaded development environment:

1. Thread control
2. Memory locking
3. Thread-safe libraries

Java provides built-in support for creating and starting multiple threads of execution. C or C++ require additional libraries, like the Posix threads library; Küchlin [31] has endowed Saclib with multi-threading in this manner. A$^\sharp$/Aldor cannot take advantage of these libraries for reasons discussed below. Computer algebra systems like Maple or Mathematica do not provide any tools for thread creation, although an experimental version of Maple uses threads in order to parallelize selected internal operations [5].

Memory locking is vital in a multi-threaded environment in order to prevent memory accesses of independent threads from interfering with each other. The Posix threads library provides semaphores and memory locks as library calls. Java integrates semaphores into the language itself, yielding the potential of enhanced efficiency through data-flow analysis and compiler optimization.

In order to take full advantage of multi-threaded execution, each library linked into the executable must support multiple threads. If this feature is lacking, a multi-threaded

*Private communication from Joshua Bloch at Sun Microsystems.

application is denied access to its most basic support libraries like input and output and becomes virtually useless. Most of today's operating systems have thread-safe system libraries, so this is usually not a problem for C or C++ applications. The Java libraries, which make extensive, possibly internal, use of multi-threading are already thread-safe. The run-time libraries of the A♯/Aldor compiler are presently not thread-safe, which makes multi-threading problematic.

Distributed Computing We have already seen that Java provides built-in support for network communication. Distributed computing can be built on top of this layer using serialized objects to communicate over sockets. However, Java provides the programmer with a more convenient way of expressing distributed computing and remote execution with its remote method invocation (RMI) mechanism, which loosely corresponds to a remote procedure call.

Remote method invocation is integrated into the Java library hierarchy and is convenient if the complete distributed system is built using Java. For integrating Java into programs written in other languages that approach is not an optimal solution since we would need to create Java wrappers for integrating legacy code. However, Java 1.2 includes support for CORBA (Common Object Request Broker Architecure) [14]. and thus communication with programs that follow the CORBA protocol is transparent.

3.4 Memory management

Storage allocation, especially in a parallel/distributed setting, remains a sticky point. In Java garbage collection is automatic, a process that presented itself problematic in our Lisp-based Dagwood system [19]: memory was filled up uncontrollably and a full garbage collection took days of computation. Automatic garbage collection provides convenience to the programmer, but it has its price: the virtual machine must determine if an object is still in use, either by a reference count or by an intermittent mark-and-sweep memory scan. Explicit storage management, as in the constructor/destructor mechanism of C++, is especially appropriate for large data structures, which occur in the symbolic computation context. The algorithms know exactly when storage becomes reusable and can cleverly overlay data of possibly diverse type and in a cache-size sensitive fashion: storage preservation becomes part of the algorithmic objectives. In C++, garbage collection can be implemented and libraries that perform garbage collection can be plugged-in. The allocator template of the C++ standard template library provides a common object interface for garbage collectors. Givaro [20] is an example for a parallel symbolic computation library in C++ that performs garbage collection by reference counting.

Java 1.2 allows the programmer some control over memory allocation through what are called weak references. For example, references to newly made objects can be placed into a table that acts as an object cache. If an object is found in this table, it needs not be constructed but can be shared. Maple implements the object cache technique for sharing one and the same subexpression. A reference in the table should be weak, so that if it remains the only reference to an object its storage is garbage collected and the reference in the table set to null.

3.5 Additional Useful Java design features

If C++ pioneered the design of a standard library for sequential data structures and algorithms, then Java pioneered that of a standard object hierarchy and library for building graphical user interfaces in an operating system-independent fashion. We wish to mention two more features.

Serialization A standard Java feature is conversion of most any object into a serial stream, ready for writing onto a file or transmission over a network. Object oriented compilers like C++ or A♯/Aldor require the programmer to explicitly define the serialization for every single class. With Java, the programmer is able to define a special external representation on a class-by-class basis, but in most cases the default serialization provided by the language is used, yielding instant compatibility of the serialized streams among different code-bases. An object can be serialized, sent over a network and reconstructed by the receiving program without requiring an agreed common protocol (other than using the Java language). Java leaves the programmer in control over what parts of an object the default serialization procedure should save and which should be discarded. During reconstruction, Java takes care to preserve the sharing of objects though multiple references. The programmer may choose to partially override the default reconstruction process by deciding to replace certain serialized objects with alternate instances. Computer algebra systems, like Maple or Mathematica, usually provide serialization for their user level objects but not for internal kernel objects. CORBA implementations also provide serialization techniques for C++. The CORBA specification provides serialization that works between components written in different object-oriented languages, e.g., between C++ and Java.

BigNumbers Java 1.1 has a very limited package for arbitrary precision integer and floating point arithmetic, which was developed mainly for cryptographic applications.

4 Experience with Java Symbolic Computation Programs

The authors have built several symbolic computation packages in languages other than Java: Bernardin has extensive Maple experience: his bivariate polynomial factorization algorithm is now part of the standard Maple library. Char is a co-designer of Maple and has recently worked on problem solving environments. Kaltofen has built the Lisp-based Dagwood system [19] for manipulation of straight-line programs and the C++-based FoxBox system [17] for manipulation of black box symbolic objects. All three authors have experience in parallel computation [5, 12, 16, 30].

4.1 Timings

For a first benchmark we chose multiplication of univariate polynomials over a small prime field. We chose it because this represents a common low-level operation in symbolic computations, which are found at the heart of many algorithms involving univariate and multivariate polynomials over the rationals. We implemented an in-place algorithm for classical polynomial multiplication [36] in both C and Java.

Table 1 reflects the results of multiplying polynomials of degree $n = 10000$ to $n = 50000$. All times are in CPU

seconds. The tests are run on a Sun Enterprise 3500 with 6 CPUs clocked at 336 Mhz. The run-time environment is Solaris 2.6 and Sun's Java JDK 1.2 (beta 5). We compiled the C code with the Sun Workshop C Compiler 4.2. The "C" column of table 1 reflects the timings with the compiler switches set to "-O". For the "C*" column we enabled more powerful optimizations using the options "-native -fast -xO4". However, the latter options imply that the resulting binary is heavily optimized for the specific machine that it has been compiled on. Supported instruction set extensions as well as cache configuration is taken into account by the compiler. This means, that the binary will be portable to a different (yet binary compatible) Sun machine only with a potential performance penalty.

For comparison, we include timings using the modp1 data structure of Maple V Release 5. Arithmetic with this data structure is implemented in Maple's C kernel.

n	Java	Maple	C	C*	Java vs. C*
10000	7s	9s	9s	3s	2.33
20000	30s	37s	36s	13s	2.31
30000	69s	77s	82s	31s	2.23
40000	124s	137s	146s	56s	2.21
50000	196s	218s	231s	89s	2.20

Table 1: Timings for polynomial multiplication over $\mathbb{F}_p$

Here lies another advantage of the Java approach of "just-in-time" compiling. The byte-code is free from platform specific optimizations and the run-time environment will decide which set of platform specific optimizations to use. The last column of table 1 shows the time factor we have to pay for using Java over heavily optimized "C" (the C* column).

It is quite impressive that the latest Java run-time environments are already competitive with conservatively optimized C, especially if you take into account that the code from above makes heavy use of arrays and that Java provides us with automatic array bounds checking which C does not have. We expect the Java runtimes to further improve and get closer to the running times of heavily optimized C code as just-in-time compiler technology matures.

For our second benchmark, we also use polynomial multiplication. But instead of representing the polynomials as an array of 32 bit integers and limiting the algorithm applicability to polynomials over small prime fields, we now represent the polynomials as arrays of pointers to generic coefficient objects. Now we can parameterize our multiplication algorithm by any coefficient ring, that has to be implemented separately.

With Java, we first define a generic type (interface) for `Rings`. We then implement one concrete `Ring`, namely $\mathbb{F}_p$ with p small. We further define a generic dense polynomial ring class which uses the operations of a generic coefficient ring in order to implement polynomial multiplication.

We compare this Java implementation with C++. Here we use templates, which have been described earlier. We define one template class for small prime fields, parameterized by the characteristic. We define another template class for dense polynomials, parameterized by the coefficient ring, i.e. the small prime field class from above.

Table 2 summarizes the results for multiplying polynomials of degree $n = 1000$ to $n = 8000$. It is important to note that the C++ timings do not contain the time spent in memory allocation and deallocation. This is because we had to disable deallocation completely as our algorithm makes use of shared object instances and we did not implement C++ garbage collections. For the same reason, we could not go further than $n = 8000$ in our test since the C++ runtime for the latter already required over 1.2GB of main memory. We used the Gnu compiler 2.8.1 with the options "-O4 -mv8" for compiling the C++ code.

For comparison we include timings using the A♯/ Aldor compiler 1.1.10b (which, as a back end, uses the Gnu compiler with the -mv8 switch as above). We also used the $\sum_{\text{it}}$ [10] library which provides polynomial arithmetic in Aldor.

Note, that for all these benchmarks, the times given include the time needed for constructing the polynomials $\sum_{i=0}^{n} a_i x^i$ and $\sum_{i=0}^{n} a_{i+n+1} x^i$ to multiply, using the following recurrence for a_i:

$$a_0 = 1, \quad a_i = \begin{cases} 1 & \text{if } 2a_{i-1} + 1 = 0, \\ 2a_{i-1} + 1 & \text{otherwise.} \end{cases}$$

n	Java	Aldor	C++	Java vs. C++
1000	2.6s	1.2s	0.9s	2.8
2000	9.1s	4.3s	3.6s	2.6
3000	20s	10s	8s	2.5
4000	36s	19s	14s	2.6
5000	57s	30s	22s	2.6
6000	82s	42s	31s	2.6
7000	111s	56s	42s	2.6
8000	146s	72s	57s	2.6

Table 2: Generic polynomial multiplication timings

One can conclude that for generic algorithms, which play an important role in symbolic computation, Java performs fairly well. The difference of a factor of 2.2–2.3 compared with a factor of 2.6 between our first and second benchmark is because in the second benchmark the C++ timings do not include memory management, whereas the Java timings (as well as the Aldor timings) do.

We also like to point out that in the direct, non-generic first test example we do not perform a modulo p division after every arithmetic operation. Instead, we reduce partial convolutions just before overflow in the accumulated intermediate integer occurs. Timings indicate a savings of a factor of 2 over non-generic arithmetic that reduces after each operation. Therefore, the pointer indirections in the generic arithmetic cost us about a factor of 12. Modulo p polynomial arithmetic for a small prime may be so special that a generic implementation may never yield acceptable results. For generic operations on larger objects, such as long integers, the reference pointers for the abstract methods and arguments needed for its implementation cause much less penalty.

4.2 Design of symbolic computation components using Java Beans

This section describes the use of Java to experiment with the software component paradigm in building Problem Solving Environments (PSEs) that use symbolic computation. Java Beans are made from ordinary Java code by employing additional coding conventions when building a software components. An application is built by plugging together and

configuring instances of Beans. The application builder can view their task as configuration and wiring – plugging together Beans and entering initial values for configuration (data) fields of each Bean. Java Studio [26] is an example of a component composition system for Java Beans. It presents a visual programming tool to application development using component configuration and wiring. We found Java's multithreading features a great aid in building software components, making it easier to handle simultaneous or interrupting incoming events. Java Studio also capitalizes on strengths of Java in the following ways:

- It builds graphical user interface icons automatically for each Bean as it is imported, using reflection to determine the input and output ports as well as the number, name, and type of configuration fields for the component.
- Java Studio can provide a number of graphical user interface-building widgets as "giveaways" because the Java library already provides the base functionality.
- To assist rapid prototyping, components can be loaded and executed on the fly.

Our Bean building efforts have focussed on providing a Bean form for the Matlab and Maple engines found in TechTalk [32]. We called this "MathServerBean". We extended our original technique of engine invocation (talking to a separate engine process through pipes) after we discovered that the same style of interaction could handle Waterloo Maple's MathEdge (C-subroutine style version of Maple) coded as a Java Native method. Engine configuration and invocation for either the "pipe" or "native method" modes of engine can be performed at run time, allowing an application to adapt efficiently to new needs as users develop them.

The role of the Java code in MathServerBean is to initialize the underlying math engine, and await requests for work through incoming events. Events are objects that have embedded text as a component – incoming events are commands to the engine, outgoing events may contain the engine's response. The role of the Java wrapper is fourfold:

1. to initialize the engine
2. to interrupt or shutdown the engine upon receipt of the appropriate incoming event
3. to embellish the basic command with the extra text that may be necessary to get the engine to operate within the Bean (e.g. extra semi-colons, newlines, procedure calls)
4. to parse the output of the engine, stripping away interactive prompts, separating error or warning messages from answers, etc. In order to do proper parsing, it is sometimes necessary to get the engine to emit an end-marker token or signal after processing input. Thus there is synergy between processing done for engine input and output.

One way of viewing the MathServerBean is that it provides the functionality of a shell scripts or an interaction-handling system such as *expect* [33] in a software component (OS-independent, portable, plug and play) form. As might be expected, most of our operational difficulties have to do with the idiosyncrasies of engine input and output on various platforms. We intend to alter our design for MathServerBean so that it depends on a separate engine interaction component to specify the engine I/O protocol. This is an admission that at present, idiosyncrasies of engine interaction cannot be handled simply through data configuration; procedural information needs to be provided sometimes.

We have found the unstandardized text input and output for MathServerBean adequate for our simple needs in application building at this time. In part this is because our input needs are for commands, which are engine-specific, and our output needs are fairly simple (i.e., text works okay as a way of displaying the answer), or not are standard mathematical expressions, e.g., Fortran subroutines or output to be fed into a visualization system. Clearly, however having a data standard that encompasses both control and procedural information as well as mathematical semantics would be beneficial, as it would simplify the engine-specific parsing and translation that we currently employ in moving data between some components.

Since Java Beans for the near-term can only run on a single processor, there is a limit to their usefulness in building distributed systems. However, we believe that our experience with them has given us insights as to the needs and usefulness of symbolic computation systems in other frameworks that do permit distributed operation, such as Java RMI or CORBA.

5 Appraisement of Java for Symbolic Computation

Java, despite its claim to do everything for everyone, suffers from the growing pains of a large programming language and library. There are already several standards defined by versions 1.0, 1.1, and 1.2. In the case of floating point arithmetic Java designers missed hard-learned lessons [28]. It will take some further time before the the common usage patterns stabilize and the interfaces to non-Java components persist.

There are two apparent uses of Java components in symbolic computation. The first is a generic graphical user interface for interactively accessing the functions and objects in a library such as NTL or Saclib. Applets even permit the embedding of such an interface into a web browser. One may view the resulting graphical user interface as a web document with active mathematical components. In [44] documents with active expressions are designed in Oberon.

The second is a library of base classes for arbitrary precision rational numbers and polynomials. Not only may such classes be useful for demonstrating symbolic methods in web applications, such as embedded modules in educational software, where high performance is not crucial, but they can be used in what we call the *contributory parallel web computation project*. Here a computational problem that can be divided into many medium-sized unsynchronized tasks with no shared memory ("embarrassingly parallel") is posted on the Web. All who wish to contribute to the computation can download an applet in their browsers and donate cycles towards the solution of the problem by permitting the thread to run on their computers for a while. Java's sandbox principle is crucial: the Java byte code can be trusted to be computer-virus-free, which may encourage participation. In [6] Java classes are described that manage such a project, and we are in the process to develop the software for a computational problem.

Furthermore, we foresee that Java components will be

develop for coordinating existing software of diverse functionality ("glue between components"), as Java is specifically designed as a programming tool for network tasks and interactive processing. The wrapper idea for legacy programs is sound: it provides a variety of functionality with a modest investment in implementation. Java is well-suited for wrappers because of its built-in support for "talking to programs": portable "shell scripts", portable multi-threaded support for control. We recommend that builders of new symbolic software design of their codes with respect to their usage as software components. In particular:

1. Programs should provide as much functionality as possible through the use of non-proprietary interchange standards. It implies some flexible thinking about where the results of a symbolic computation may end up — not necessarily on a proprietary mathematics system GUI, but perhaps on a GUI custom-built for an application, in a commodity document processor, visualization system, or database. Some of the most valuable functionality of the proprietary systems is not necessarily output as mathematical expressions, but as procedures, numerical data, visualization, or mathematical display.

2. Programmers should assume that their software needs to talk to components that do not share the same process space. This does not mean that a "subroutine load and execute" feature of a computer algebra system is useless — there are many computations that are poorly handled as the interaction of separate components. We mean just that the latter notion is not a complete facility for application building.

A final question remains. **Will large scale symbolic algorithms be programmed in Java?** We believe that our timings in Section 4.1 indicate that it is possible to generate efficient code from Java programs for symbolic computation that are written in a modern generic style with abstract domain types. Despite our concerns for Java's memory model, especially when a high percentage of the available memory is used, we think that symbolic computation software builders can adopt Java for major software design and thus take full advantage of the computing infrastructure that Java has provided.

Acknowledgement The authors thank the referees for their helpful comments.

References

Note: many of Erich Kaltofen's publications are accessible through links in the online BibTeX bibliography database at `www.math.ncsu.edu/~kaltofen/bibliography/`. The Java benchmark examples discussed in Section 4.1 can be downloaded from `www.inf.ethz.ch/personal/bernardi/javabench`.

[1] Abbott, J., Díaz, A., and Sutor, R. S. A report on OpenMath a protocol for the exchange of mathematical information. *SIGSAM Bulletin 30*, 1 (1996), 21–24.

[2] Abdali, S. K., Cherry, G. W., and Soiffer, N. An object-oriented approach to algebra system design. In *Proc. 1986 Symp. Symbolic Algebraic Comput. Symsac '86* (New York, N. Y., 1986), B. W. Char, Ed., ACM, pp. 24–30.

[3] Athapascan-1. `http://www-apache.imag.fr/software/ath1/`, Oct. 1998.

[4] Bachmann, O., Schönemann, H., and Gray, S. A proposal for syntactic data integration for math protocols. In *Proc. Second Internat. Symp. Parallel Symbolic Comput. PASCO '97*, pp. 165–175.

[5] Bernardin, L. Maple on a massively parallel, distributed memory machine. In *Proc. Second Internat. Symp. Parallel Symbolic Comput. PASCO '97*, pp. 217–222.

[6] Bernardin, L. A Java framework for massively distributed symbolic computing. *Mathematics and Computers in Simulation* (1999). To appear.

[7] Blitz++. `http://monet.uwaterloo.ca/blitz/`, Dec. 1998.

[8] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. Making the future safe from the past: Adding genericity to the Java programming language. *SIGPLAN Notices 33*, 10 (Oct. 1998), 183–200. Proc. 13th OOPSLA. See also `http://www.cs.bell-labs.com/~wadler/pizza/gj/`.

[9] Braham, S., Yu, T., and Chan, C. The Java OpenMath library (version 0.5). `http://pdg.cecm.sfu.ca/openmath/lib/`, July 1998.

[10] Bronstein, M. Σ^{it} – A strongly-typed embeddable computer algebra libary. In *Proc. DISCO '96* (Heidelberg, Germany, Sept. 1996), J. Calmet and C. Limongelli, Eds., vol. 1128 of *Lect. Notes Comput. Sci.*, Springer Verlag, pp. 22–33.

[11] Chan, K. C., Díaz, A., and Kaltofen, E. A distributed approach to problem solving in Maple. In *Maple V: Mathematics and its Application* (Boston, 1994), R. J. Lopez, Ed., Proceedings of the Maple Summer Workshop and Symposium (MSWS'94), Birkhäuser, pp. 13–21.

[12] Char, B., Johnson, J., Saunders, D., and Wack, A. P. Some experiments with parallel bignum arithmetic. In *Proc. First Internat. Symp. Parallel Symbolic Comput. PASCO '94* (Singapore, 1994), H. Hong, Ed., World Scientific Publishing Co., pp. 94–103.

[13] Collins, G. Saclib2.1. `http://www.cis.udel.edu/~saclib/update.html`, Sept. 1998.

[14] CORBA/IIOP 2.2 specification. `http://www.omg.org/corba/cichpter.html`, July 1998.

[15] Dalmas, S., Gaëtano, M., and Watt, S. An OpenMath 1.0 implementation. In *ISSAC 97 Proc. 1997 Internat. Symp. Symbolic Algebraic Comput.* (New York, N. Y., 1997), W. Küchlin, Ed., ACM Press, pp. 241–248. See also `http://naomi.math.ca/`.

[16] Díaz, A., Hitz, M., Kaltofen, E., Lobo, A., and Valente, T. Process scheduling in DSC and the large sparse linear systems challenge. *J. Symbolic Comput. 19*, 1–3 (1995), 269–282.

[17] Díaz, A., and Kaltofen, E. FoxBox a system for manipulating symbolic objects in black box representation. In Gloor [22], pp. 30–37.

[18] Erlingsson, U., Kaltofen, E., and Musser, D. Generic Gram-Schmidt orthogonalization by exact division. In *Proc. 1996 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'96)* (New York, N. Y., 1996), Lakshman Y. N., Ed., ACM Press, pp. 275–282.

[19] Freeman, T. S., Imirzian, G., Kaltofen, E., and Lakshman Yagati. Dagwood: A system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Software 14*, 3 (1988), 218–240.

[20] Gautier, T., Roch, J.-L., and Villard, G. Givaro. `http://www-apache.imag.fr/software/givaro/`, Jan. 1999.

[21] Giesbrecht, M., Lobo, A., and Saunders, B. D. Certifying inconsistency of sparse linear systems. In Gloor [22], pp. 113–119.

[22] Gloor, O., Ed. *ISSAC 98 Proc. 1998 Internat. Symp. Symbolic Algebraic Comput.* (New York, N. Y., 1998), ACM Press.

[23] Granlund, T. *The GNU Multiple Precision Arithmetic Library*. Free Software Foundation, Cambridge, MA, 1993.

[24] Husbands, P., Isbell Jr., C. L., and Edelman, A. Interactive supercomputing with MITMatlab. See link at `http://www-math.mit.edu/~edelman/`, Aug. 1998.

[25] Iglio, P., and Attardi, G. Software components for computer algebra. In Gloor [22], pp. 62–69.

[26] Java Studio. `http://www.sun.com/studio/index.html`, 1997.

[27] Jenks, R. D., and Sutor, R. S. *axiom The Scientific Computing System*. Springer Verlag, New York, 1992.

[28] Kahan, W., and Darcy, J. D. How Java's floating-point hurts everyone everwhere. Link from `http://www.cs.berkeley.edu/~wkahan`, 1998.

[29] Kaltofen, E. Challenges of symbolic computation my favorite open problems. *J. Symbolic Comput. 29*, 6 (2000), 891–919. With an additional open problem by R. M. Corless and D. J. Jeffrey.

[30] Kaltofen, E., and Lobo, A. Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica 24*, 3–4 (July–Aug. 1999), 331–348. Special Issue on "Coarse Grained Parallel Algorithms".

[31] Küchlin, W. On the multi-threaded computation of integral polynomial greatest common divisors. In *Proc. 1991 Internat. Symp. Symbolic Algebraic Comput. ISSAC'91* (New York, N. Y., 1991), S. M. Watt, Ed., ACM Press, pp. 333–342.

[32] Lakshman Y. N., Char, B., and Johnson, J. Software components using symbolic computation for problem solving environments. In Gloor [22], pp. 46–53.

[33] Libes, D. *Exploring Expect*, 1st ed. O'Reilly Associates, Dec. 1994.

[34] Loos, R. G. K. Toward a formal implementation of computer algebra. *SIGSAM Bulletin 8*, 3 (Aug. 1974), 9–16. Proc. EUROSAM'74.

[35] Monagan, M. B. Gauss: A parameterized domain of computation system with support for signature functions. In *Proc. DISCO '93* (Heidelberg, Germany, 1993), A. Miola, Ed., vol. 722 of *Lect. Notes Comput. Sci.*, Springer Verlag, pp. 81–94.

[36] Monagan, M. B. In-place arithmetic for polynomials over Z_n. In *Proceedings of DISCO '92* (1993), vol. 721 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 22–34.

[37] Musser, D. R. Multivariate polynomial factorization. *J. ACM 22*, 2 (Apr. 1975), 291–308.

[38] Myers, N. A new and useful template technique: "traits". *Borland C++ Report* (June 1995). `http://www.borland.com/borlandcpp/news/report/CR9506myers.html`.

[39] Norman, A., and Fitch, J. Interfacing REDUCE to Java. In *Proc. DISCO '96* (Heidelberg, Germany, 1996), J. Calmet and C. Limongelli, Eds., vol. 1128 of *Lect. Notes Comput. Sci.*, Springer Verlag, pp. 271–276.

[40] POOMA 2.0. `http://www.acl.lanl.gov/pooma/`, Nov. 1998.

[41] Schreiner, W. Distributed Maple — user and reference manual (V 1.0.7). Link from `http://www.risc.uni-linz.ac.at/software/distmaple/`, Mar. 1999.

[42] Shoup, V. NTL: A library for doing number theory (version 3.1b). Link on web document `http://www.cs.wisc.edu/~shoup/`, Univ. Wisconsin, Dept. Comput. Sci, 1998.

[43] Watt, S. M., Broadbery, P. A., Dooley, S. S., Iglio, P., Morrison, S. C., Steinbach, J. M., and Sutor, R. S. A first report on the $A^\sharp$ compiler. In *ISSAC '94 Proc. Internat. Symp. Symbolic Algebraic Comput.* (New York, N. Y., 1994), ACM Press, pp. 25–31.

[44] Weck, W. *On Document-Centered Mathematical Component Software*. PhD thesis, ETH Zurich, Institute of Scientific Computing, Sept. 1996.