

On Developing 2-D Signal Processing Applications

KP Lam
Computer Science Department,
University of Keele,
STAFFS ST5 5BG, United Kingdom.

ABSTRACT

The rapid growth in capability and performance of modern computer systems can be attributed to the underlying Very Large Scale Integration technology, which enables an unprecedented number of components to fit on a chip and clock rates to increase. However, such a remarkable progress in modular systems construction, used successfully in the recent revolution in digital telecommunications, has yet to be matched by software design (even) in the same field. This paper discusses software development issues on two-dimensional digital signal processing, and assesses the credentials of Java as a practical programming language. Using previous work as a case study, it attempts to further understand the foregoing architectural evolution which may bring about a convergence of the traditionally disparate approaches of software and hardware design.

1. Introduction

The past decade has witnessed impressive advances in large-scale integrated circuit (IC) engineering which has brought home to engineers the customer-specifiable technology of application-specific ICs, or ASICs. Unlike standard ICs, the construction of ASICs engages the designer into a rigorous framework of detailed assessment and critical appraisal for composing the complex electronic systems at hands. Given the outstanding architectural development of programmable logics devices (PLDs), one could argue that useful lessons be learnt, particularly in software development, by designers and researchers in component-based software engineering¹ (CBSE), to facilitate construction of complex and high functionality software by building systems with components as described above. This paper examines this fundamental tenet of CBSE within the context of two dimensional digital signal processing.

Engineering software by virtue of modular designs with component reuse offers an architectural framework which has existed since the earliest day of computing. The ever-increasing demand for building complex and high functionality software in ever shorter time periods has created a new and challenging need for a more organised approach to systems development [2]. In the general context of software development, object-oriented programming offers a technical framework whereby programs are so created that they are more amenable to be reused, thereby aiding the process of systems integration. In this respect, we have chosen for this work an object-oriented programming language, Java [3], which is widely known for its compactness and portability. The ubiquity of high quality Java Virtual Machine (JVM) on computers worldwide, the popularity of the (Java-oriented) Internet computing, and the rapid growth of commercial APIs, have all contributed to increased adoption of Java amongst academic and industrial computer users. Indeed, these attributes have long made the industrial-strength, standardised ANSI C the 'established' high-level programming language for practical signal processing applications.

A further description of Java characteristics instrumental to their support for signal processing and component-based software design are discussed, alongside, where appropriate, features of C that are commonly exploited. The concurrent programming framework is then introduced as a disciplined approach to systems construction. An illustration of the concepts discussed is presented in section 4., using previous work on complex moments generator. Realisation of the proposed generator architecture by software components is described and demonstrated. Concluding remarks summarising key points of this paper are included at end of this paper.

2. Java vs ANSI C - a Comparative Study

Practical signal processing software developers tend to adopt a more traditional approach and prefer using a high-level procedural language such as ANSI C, which is both expressive and relatively terse [1]. In addition, low-level facilities such as byte manipulations was conveniently accessible to experienced programmers to fine tuned programs down to hardware architectural details, even to the detriment of program portability. These two facets of applications programming produce conflicting requirements; on the one hand, the language should be able to express all possible high-level procedural constructs so that the underlying architecture would be sufficiently transparent. On the other hand, the language should also have low-level tools and capabilities, to permit direct access of specific hardware at the programmers' will. This widened gap has, in time, hampered the systematic and flexible development of complex, high functionality software required of modern applications.

¹ <http://www.sei.cmu.edu/cbs/icse99/cbsewkshp.html>, August 1999.

Java adopted the widely used, object-oriented and industry-proven C++ programming language. The inherent pitfalls and complexity in the evolution of the ‘heavy-duty’ implementation language of ANSI C, upon which its superset of C++ was based, have forced upon many refinements of the Java technology to make it compact, compatible and familiar. The rapid developments of Java in the computer science and engineering communities, particularly in terms of the continuous improvements in performance, have since made Java the primary choice of language for implementing Internet/Intranet-based applications and network-aware devices; *eg.* cellular phones and digital TVs. The object-oriented, JVM-bounded Java language is a major asset characterised with many attributes which are vital for practical signal processing. These are discussed below.

- *Portability and Interoperability* - Portable C/C++ programs were written in the past by strictly adhering to ANSI features. For many years, portability of these programs was facilitated by the availability of good compilers, which exist on many different combinations of hardware and operating systems. However, as multimedia capabilities are becoming indispensable, applications that were once only available on specialised platforms, can now be found running on the Internet, often communicating with other distributed applications. The adoption of GUIs, for example, is almost universal. Neither of these languages, nor the combination of both, could offer a capability in such a way that applications can run without modification (or even re-compilation) on a variety of platforms. Java, on the other hand, is becoming a standard technology for delivering cross-platform by virtue of its bytecode architectural model [7]. It decouples development from deployment, and realises the fundamental basis for Java’s “Write Once, Run Anywhere” philosophy.
- *Quality and Performance* - The in-built memory management, the thread handling capabilities (see below), and the proper object-oriented programming techniques that Java enforces, all contributes to its superiority over C/C++. In addition, the Java exception model forces a function that generates an exception to add a *throws* clause to the function declaration, rendering it superior to the *errno*-based macro mechanism of C. Where performance is of prime consideration, highly optimised legacy code could be incorporated into Java program as *native code* which permits integration of the existing C/C++ codes. Further, full Java compilers have been written for most popular platforms. In general, compute-intensive Java applications and applets would have to be compiled into an intermediary between bytecode interpreters and compilers, using a *Just-In-Time* (JIT) compiler [7]. Interactivity and response of multi-tasking applications can be improved using the multi-threading supports of Java language. This high-level programming facility allows Java threads to be executed concurrently, and, where necessary, in parallel, to exploit multi-processor platforms. This represents a significant departure from the traditional approach whereby multiple threads were programmed with operating system supports, most commonly via system libraries. This way, portable concurrent programming can be achieved by leaving hardware mapping decisions entirely to the JVM on which an application runs. A further discussion of this topic is included in section 3.
- *Program Interfaces and Specifications* - As well as being freed from the restraint of compatibility (with C/C++), the object orientation of Java renders a consistent environment complete with class hierarchies for input/output (I/O), GUI design, networking, and many other utilities. Through class management and a consistent object interface, the language provides a technical framework within which applications can be specified using generic programming designs and constructed with portable programs. In the former case, for example, the java I/O package, *java.io*, manipulates file and network data I/O collectively as stream without direct involvement of its source. The emergent of Java API-compliant packages, such as media-based Java API [8], provides both a starting point for building extensible and portable applications that encourages reuse, and systems development by components integration.

3. A Concurrent Concurrent Systems Development Dimension

Java, like C, was originally devised as a programming language for operating systems, but on a much larger and ambitious scale. As such, the object-oriented concurrency has emerged primarily to optimise performance and response of real-time systems. A distinct advantage of object-oriented concurrent programming, is that it facilitates different styles and capabilities provided by different concurrent paradigms including semaphore, monitor and and CSP (Communicating Sequential Processes) [4]. This is in sharp contrast to concurrent C/C++ programming, which is derived from the traditional libraries-based, multi-threaded concurrent systems programming, and consequently, inferior to Java in such important characteristics as encapsulation, security and safety, modularity and extensibility.

Concurrency is seen as an indispensable ingredient for systems development, which enables co-operating system components be designed, constructed and composed, to render a target system that meets a set of specifications. The VLSI technology described earlier is a case in point. In software engineering, concurrency opens up design possibilities that are either impossible or impractical in sequential programs - software, particularly those constructed

under the the traditional framework for procedural design and programming, is perceived as largely sequential. Classic examples of complex and large-scale concurrent software systems are exemplified by modern operating systems such as UNIX™, and contemporary GUI programming. In the latter case, the key advantage of concurrent systems design is that it offers a natural means and analytical techniques for *reactive* programming; a compositional methodology which controls and synchronizes reactive responses (and outputs) to multiple, independent and (often) simultaneously operated inputs. In the broader context, concurrent programming promotes a highly disciplined approach to constructing systems with cooperating processes, which are characterised by: *scalability* (which facilitates reuse), *liveness* (which specifies functional composition), and, *safety* (which stipulates security).

As with the scalability of design, the techniques for ensuring safety generally rely on good engineering practices, including several with a foundation in ‘formalism’, rather than formal methods themselves. On the design level, they necessitate a clear understanding of required properties and constraints surrounding representations of a system component. The constraints typically stem from the definitions of high-level conceptual attributes during the initial design (of classes), and they usually hold regardless of how these attributes are represented and accessed via methods and their data. Similarly, liveness concerns, demand that system resources management be optimised for the given target loads, thus requiring the careful use of suitable synchronisation constructs, and impacting upon the interactions between cooperative components. This has lead to a wealth of general-purpose (concurrent) techniques for structuring and managing the intrinsic complexity of interactions among multiple, cooperative software components [6]. This issue, which is barely of concern to sequential programming, could be critically important for many (real-time bounded) signal processing designs, in which failure to execute within a given time could lead to catastrophic system errors.

4. Building Component-Based Systems - a Pilot Study

Previous work on two dimensional generators for complex moment descriptors (CMD) has demonstrated a de/composition technique for constructing higher order moments using a linear combination of lower order moments [5]. This feature description method is typical of two dimensional signal processing algorithms in terms of the amount of data processed, multi-dimensional representations, and similarities and/or relationships with 1-dimensional signal processing. By exploiting the duality description of Infinite Impulse Response (IIR) filters and moment equations, and the separability of moment computations, it realises individual generators through a cascade network of basic single-pole filters; an important and widely used structure for one dimensional signal processing (by virtue of the z-transform characteristics). Mathematically, given the equivalent relationship of the impulse responses $h_p(t) = t^p u(t)$, $u(t) =$ unit step sequence, and the p^{th} -order moments of the input sequence of $x(t)$ evaluated at the end-point $t = N$, higher order moments could be generated in one dimensional case using single-pole filters of the form $Z^{-1}\{1/(z-1)\} = u(t-1)$ and expressed in the inverse Z transform as:

$$Z^{-1}\left\{\frac{1}{(z-1)^{p+1}}\right\} = \sum_{k=1}^p c_k (t-p)^k u(t-p), \quad p \geq 1 \quad \dots(1)$$

where c_k is a constant. Equation (1) was extended to extract moments of higher dimensions, using synchronous and orthogonal computation of moments along the separable dimensions of the CMD required. In the two dimensional case, therefore, a specific moment M_{pq} can be logically constructed as 2-D rows and columns of single-pole filters, whereby O_{pq} are combined according to a matrix specifications defined by (1) [5]. Figure 1. illustrates the 2-D construction described above.

Any set of 2-D moments could be generated by mapping the required set of identical single-pole filters (*ie.* the basic building blocks), each characterised by the transfer function $H(z) = 1/(z-1)$, onto the corresponding 2-D grid network (of Java threads) as depicted in Figure 1. For each thread, the multiple outputs that it generated by computing $H(z)$ were ‘feed’ (= read) synchronously into its own unit-delayed feedback loop and the network. In addition to the correct ordering of O_{pq} , the requirement for distributed synchronous controls is essential, for two reasons; (a) Non-deterministic objects interference and/or data access by the asynchronously operated Java threads were eliminated, and, (b), the *wait* and *notify* methods could be used in conjunction with the Java’s *synchronized* qualifier to specify the correct ‘data flow’ (control flow) within the 2-D grid [7]. The latter is particularly important for the individual filters composing the first row of the network, where the 2-D input image $f(x, y)$ is read successively in a row-major order.

5. Design Validations

The design was validated using three criterion: portability, robustness and performance.

- *Portability* was further demonstrated by porting the moments generator network from a SUN Ultra5 host Windows/Linux PC with practically no modification required of the Java classes implemented.

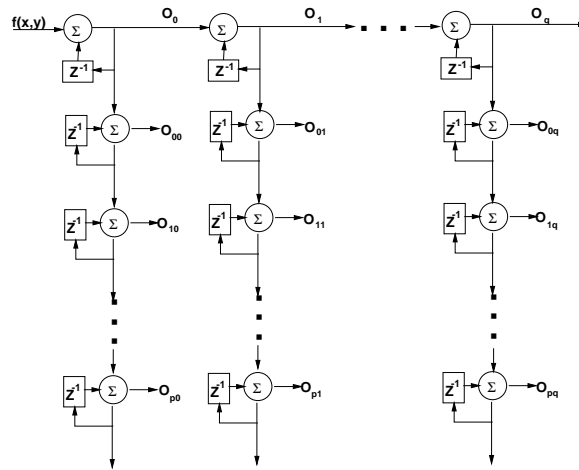


Figure 1. Composing M_{pq} using $H(z)$ components on a 2-D grid.

- *Robustness* was assessed by generating different sizes of the 2-D networks, with the intention of rendering a reasonably large number of objects and data access (to below the machine-specific limits). Different sizes (up to 768×768) of the 2-D input $f(x, y)$ were also applied without any problems.
- *Performance* has been measured primarily against a direct implementation of CMD of (p, q) -ordered moments, with $f(x, y)$ fixed (at 256×256). Two observations were noted; firstly, for small combined values of (p, q) , the direct implementation was a clear winner; over 30% in the trivial cases where $(p, q) = (0, 1), (1, 0)$. This trend continued for larger values of p and q , but with noticeably narrowing gaps. The narrowing could be attributed by the fact that, as (p, q) increases, a significant part of the overheads due to the management of context-switches and startup/shutdown of Java threads was offset by the high number of combinations for the (p, q) pairs. This suggested that the network is more scalable to large (p, q) . However, a comprehensive performance analysis depends on many factors, which are outside the scope of this paper.

6. Concluding Remarks

There have been strong arguments and evidence for software component producers to follow the path of hardware producers in engineering complex and high functionality software. To this end, concurrent programming combined with objects orientation is seen to offer a powerful methodology for systems development that emphasizes re-usable components, and rigorous techniques for composing them effectively to meet specifications. In this paper, we have examined this proposition in the context of practical 2-D signal processing, demonstrating the generation of complex moments by a network of basic IIR filters. Within the design, these filters were constructed as software components, which define their functionality and interface, and are available for use in many combinations. Each filter details the methods that it may apply, and the interface necessary to activate such methods in an application environment. This approach to systems engineering is completely at one with the principles and practice that are long established by hardware designers working in the field of VLSI technology.

7. Reference

- [1] Dautray R and Lions JL, *Mathematical Analysis and Numerical Methods for Science and Technology*, Vol 4, Springer 1990.
- [2] Gellersen H et al, *Object-Oriented Application Development*, IEEE Internet Computing, Jan/Feb 1999.
- [3] Gosling J and Arnold K, *The Java Programming Language*, Addison Wesley, Longman, 1996.
- [4] Hoare CA, *Communicating Sequential Processes*, Prentice Hall, 1985.
- [6] Lam KP, *A Component-based Design for Parallel Moment Generators*, *Parallel & Distributed Methods for Image Processing III*, Vol 3817, Proc of SPIE, 1999.
- [9] Rosenchein Jeffrey, and Gilad Zlotkin, *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers*, MIT Press, 1994.
- [10] Smith M, *Java an Object-Oriented Language*, McGraw Hill, 1999.
- [11] SUN site: <http://java.sun.com/products/java-media>