About This Issue

The title of this issue of *Computer Music Journal*, "Synthesis and Transformation," refers to the focus of the feature articles on the basic tasks of musical activity. Earl Dumour's interview with Arthur Roberts (shown on the cover) gives this pioneer of the field a chance to discuss the early uses of computers in his own idiosyncratic musical applications. He describes his musical oeuvre as well as more practical issues of his life as a physicist and a composer.

The Machine Tongues article introduces the technology of software sound synthesis—the traditional "technology of computer music"—and compares three popular C-language-based packages: Csound, cmusic, and cmix. It is intended for neophyte readers as an introduction and for the computer-literate as a comparison, critique, and benchmark suite. Peter Comerford's article on the simulation of organ timbres details the various techniques used in historical and current instruments and presents his Bradford Musical Instrument Simulator. He describes the foundations of past electroacoustic organs and proposes his digital architecture as a powerful and flexible processor.

The use of graphical modifications of images derived from sounds via the Gabor (wavelet) transform in an analysis-resynthesis process is introduced in the article by Daniel Arfib and Nathalie Delprat. This technique has obvious applications in the construction of musical signal visualization and processing tools. Bridget Baird, Donald Blevins, and Noel Zahler write about intelligent automatic accompaniment systems, derive an improved algorithm for "score following" in computer performers, and discuss the implementation they have developed. They present the background and literature of "synthetic performer" systems and then synthesize several known techniques, mixed with their own musical insights, into a new model.

The Reviews section is very lively, with multipart reviews of last year's International Workshop on Models and Representations of Musical Signals in Capri and the 1992 International Computer Music Conference in San Jose along with other event and publication topics. Two User's Reports—one hardware and one software—are followed by a lengthy Product Announcements section.

Editor's Notes

Putting Max in Perspective

[Peter Desain and Henkjan Honing, well known to *Computer Music Journal* readers, wrote a lengthy letter on the strengths and weaknesses of the popular Max music programming tool (see the Product Announcement in *Computer Music Journal* 15:1 and the article by Miller Puckette in *Computer Music Journal* 15:3). Robert Rowe of New York University and Brad Garton of Columbia University, coeditors of the International Computer Music Association's *Array* newsletter, collected and edited the original stimulus and several replies for their publication. We reproduce it here with minor edits as a "guest editorial," with thanks to the authors and editors.]

The following letter, originally titled *The Mins of Max*, was written by Peter Desain and Henkjan Honing to initiate a discussion about Max, in the hope that we might discover what Max does well and how it might be improved. They succeeded admirably in fomenting a discussion; their original text is ornamented with comments from Roger Dannenberg, Dean Jacobs, Cort Lippe and Zack Settel, Stephen Travis Pope, and Miller Puckette (the author of Max). Another response, from George Lewis, addresses many of the same issues in another light and is included at the end of the other letters.

The Mins of Max

For some time we have been surprised at the success of Max and the enthusiasm of its users. At the 1992 International Computer Music Conference (ICMC), for example, we noticed a very uncritical attitude toward

the Max program, which seemed to be mentioned in every second paper. Max is a data-flow language with a graphical interface that lets one manipulate the flow of data (numbers, symbols, and lists) through "a patch." It supplies a large collection of primitive modules that have one or more inputs and/or outputs. A user makes connections between these modules graphically. The data flow through these connections and are processed by the modules. Certain special inputs control when a module's output is generated. Configurations of connected modules can be combined into a new module (called a "patcher"). The directness of the graphic metaphor (connecting modules with "patch cords") contributes to Max's attractiveness.

The use of Max in our community is now pervasive; it even replaces general programming languages in many courses on interactive composition. Indeed, it is a wonderful new platform for interactive computer music and brings truly programmable tools to the stage of live interactive performances. It is the exchangeability of these patches between users that makes Max such a powerful tool. But if one asks users how they use it—be it for composition, in classrooms, or on stage—what practical and musical problems they run into, and how they solve them, quite a different story emerges. At first most users shy away from admitting having problems with Max and instead blame themselves for not being able to achieve certain goals. Later they recognize that Max may have important shortcomings. This is illustrated by the different opinions of users, ranging from "Max is programming with about the friendliest face anyone could put on it" (Carter Scholz in *Keyboard*), to "Max's main strength is in transforming one kind of signal into another; it is essentially a kind of super-mapper" (Michael Pelz-Sherman at the 1992 ICMC), to "Max is a low-level program disguised as a high-level language" (George Lewis at the 1992 ICMC). We will outline our critique here, doing so in as bold and clear a fashion as possible—hoping that the designers, vendors, users, and advocates are provoked to respond.

Peter Desain and Henkjan Honing
Nijmegen and Amsterdam,
The Netherlands

The term "data-flow" is not a good description of Max except for the naive intuitive model it implies. Data-flow computers and languages are based on the idea that (parallel) computations can be synchronized by data. For example, an addition operator would wait for its two operands before adding them. Note that this is not at all how Max works. I will not argue that Max should work this way, but since it doesn't, one should not call it a data-flow language. I think "patch language" or "visual programming language" are better terms. Since Max is not one of the data-flow languages, it lacks many of their nice properties. Interestingly, most signal processing, and certainly DSP based on unit generators, is essentially data-flow computation. Even the Max extensions for signal processing on the IRCAM Signal Processing Workstation follow the data-flow paradigm, resulting in a mismatch between the event-based Max operators and the signal/data-flow-based Max operators. The boxes and lines look the same, but the underlying semantics are quite different. This is a manifestation of some of the problems that Desain and Honing describe.

Roger Dannenberg
Pittsburgh, Pennsylvania USA

The underlying computational paradigm of Max seems to be one of its primary novel features and is a subject worthy of discussion in its own right. As several people have pointed out, Max is not a pure data-flow language in that (1) there is state embedded in the objects in the network and (2) there is no buffering of values at the inlets to objects, hence values can be overwritten and lost. I would add that, despite statements in the manuals and documentation, Max is not an object-oriented language. While certain predefined objects, such as table, do respond to a variety of different messages, the vast majority simply respond to raw values in their inlets. Perhaps more importantly, user-defined objects do not contain an explicit semantic notion of methods.

To me, Max seems to be an imperative language based on timed streams of values flowing through a static network of objects. A wide variety of [Max] objects can be usefully viewed as being generators of streams, while others can be viewed as filtering and manipulating those streams. Others can be viewed as absorbers of streams. There is a list-to-stream converter and a stream-to-list converter. Finally, the inlets of objects and some other special objects provide storage within a network.

Dean Jacobs
Los Angeles, California USA

I have been impressed with Max since the days when it was called Patcher; it's a wonderfully useful, easy-to-learn tool for teaching, performance, and experimentation. I must, however, take issue with the characterization of Max as an "object-oriented graphical programming language" as it has been referred to by its author and by its marketer OpCode Systems, Inc. I find this an unfortunate statement that muddies the issues and misleads many users. There is a wide consensus within the OOP commu-

nity (ranging from the adherents of Smalltalk to those of ADA) that there are several defining features of OO languages: (1) object encapsulation, (2) inheritance or refinement, and (3) polymorphism. There is a widespread misconception that the icons used, for example, to represent files and applications in direct-manipulation user interfaces represent "objects" and that these systems therefore constitute OOP. As present in Max, the encapsulation of "objects" in the form of Max units is a side effect of the differentiation between types and variables—a feature of structured programming—and has very little to do with OOP. The ability to "compose" icons into "sub-patches," building hierarchical configurations, is analogous to the construction of subroutines in a structured, imperative programming language such as C, and has nothing to do with inheritance in OOP languages. We could all be a little more clear in references to the different styles of programming languages—imperative, data-flow, message-passing, logic-based, etc.

Stephen Travis Pope
Palo Alto, California USA

I haven't found that people have been uncritical about Max, as Peter Desain and Henkjan Honing say; on the contrary, I never saw anyone use it who didn't make at least three major complaints. This is probably an unavoidable situation; in general, the more something does, the more one would like it to do in addition. Put another way, the larger a thing gets, the more trade-offs are likely to be involved in its design. In thinking about Max, I always sought reliability first, simplicity second, expressivity of the language third, and what I call "computer science issues" almost not at all.

Miller Puckette
Paris, France

Max Is Not a Language for Music

Max is claimed to be a music system or musical language, but the only idea of music that Max has is in the (unnecessarily) low-level form of MIDI messages. It does not provide primitives (e.g., notes, chords, or ornaments) or control structures (e.g., repeat or slow down) that may be familiar or useful to musicians. But what is even worse, Max does not provide an easy means to create these constructs that might be useful in a composition or performance within the language itself. Even in the most unconventional music, organizational and structural aspects are essential. Max lacks them.

Peter Desain and Henkjan Honing

In contrast to Desain and Honing, I find the claim that "Max is not a language for music" a feature. I'd rather search for and implement my own elements of musical structure than pick them from a menu. The danger of building in musical concepts is that they can preempt other ways of thinking and structuring.

Roger Dannenberg

I believe that when we are discussing Max, or any other programming language for music, it is important to distinguish between issues associated with the underlying computational paradigm in general (e.g., functional or imperative) and those associated with the particular "library" of musical tools provided. There is no doubt that Max provides fewer high-level musical tools than most music programming languages, such as HMSL or FORMES. A more fundamental question is whether Max allows one to build such tools. No one has convincingly argued that it is difficult to implement some particular high-level feature in Max for some particular reason.

Dean Jacobs

The remark that "Max is not a language for music" comes obviously from someone who believes that a language for making music with a computer needs to offer "musical primitives" to the user. Our belief is that any embedded "musical knowledge" or musical primitives may tend to bind us to preconceived notions about what someone else thinks music is or should be. One of the strong points of Max, from our point of view, is that it is general enough (and stupid enough) to allow us to create our own musical world, without any dependence on another's definitions or preconceptions. The point raised below by George Lewis concerning the "genericness" of interactive works produced in Max touches on a major issue in art-making in general—one can confront and examine the nature of tools used in the art-making process, and the extent to which the means of production actually influences the produced object itself. As computer music composers, we would prefer to be confronted with this issue, and we consider it an important dimension of our work. Rather than consume drive-in-fast-food-art-making products that tend to mask the nature of the tool used (a computer), we prefer the open invitation which Max offers to the user to deal directly with the production tool.

Cort Lippe and Zack Settel
Paris, France

One thing Max was not intended to do was operate on a musical level. In my own experience using "musically knowledgeable" tools, the tool always has an opinion about music that is different from mine. I would object if

my piano started "helping" me interpret the music I was trying to play. Why would I want the computer to do differently? So criticisms such as "Max is not a language for music" or "Max has no idea of [musical] time" are irrelevant to the way that I use Max. I think of Max as a musical instrument, not a musical assistant. Max occupies a niche in the ecology of music-making tools that is similar to that which a shell program (e.g., "sh" or "csh") occupies in UNIX. It's probably possible to write a shell program to, for example, find the *n*th prime number. I wouldn't do it that way—I'd use C. I wouldn't use Max to do it either—I'd write a C external function and link it into Max. Either Max or the shell might be a very suitable environment for invoking this program, though. Max or the shell are good ways for fitting things together. Max does this in a context different from the shell; the programs are often smaller (as in "+"), and real-time response is essential. It proved necessary to give Max a slightly stronger notion of data structures than the shell has. But does Max know about music? No. Does the shell? No. Should they? No.

Miller Puckette

Any modeling activity (such as describing any kind of activity within a computer program) can be based on limiting assumptions or enabling assumptions. One can indeed look on the piano as a "helper"—it has very concrete musical knowledge implicit in its tuning and user interface, and its unique but very limited timbre and envelope have a strong effect on the kind of music one makes with it. The role of the tool in the form result should be well established by now, and the myth of "generic" or "uncoloring" tools must once and for all be laid to rest. We can then focus on the kinds of assumptions Max uses in its models of the musical domain, and how these are better or worse than the (limiting or enabling) assumptions found in the design of other music software. I for one would always vote for some abstraction in music-related systems, even if it's not exactly the one I think with.

Stephen Travis Pope

Computation Time and Order

Computation time becomes involved in the problem of timing; if one datum needs a bit more computation than another, it will arrive later at the point at which they are combined. This can break the algorithm, forcing the user to rely on the workarounds provided (a special resynchronizing module). Even if an algorithm works well, a tiny change such as adding or deleting a calculation step may break it. Since computation routes can change dynamically (based on actual incoming values), there is no guarantee that patches will not break on new input. This means unpredictable and thus unacceptable results; reliable real-time behavior cannot be guaranteed. Furthermore, because of the hidden internal state of the computation (which inputs have been handled and which have not), a break in the middle can leave the system in a state from which no restart is possible. We observed that users were quite willing to close and reopen patches and even restart the application to work around this problem. The graphical-data-flow user interface suggests that one can abstract from details of control and concentrate on the meaningful flow of data instead, as is the case with modern data-flow languages. Those who try to program reliably timed patches will find themselves adding number boxes all over the place to check when data have arrived where. It has to be admitted that to design data-flow languages for the asynchronous case is not trivial, but techniques exist (like time-stamping data) that make managing the flow of control a task for the system, not the user. The Max "bang" rules are far from that.

Peter Desain and Henkjan Honing

There seems to be a general impression that [Max's] paradigm is difficult to use in and of itself. Peter and Henkjan argue that there are often problems with the times at which data arrive at operators; "race conditions" make the behavior of programs unpredictable. They also complain about programs that can enter indeterminate states in the middle of a performance. George Lewis complains below about the circuit-like nature of Max, whereby values flow to various points in the network simultaneously. I believe that these problems are a fundamental part of concurrent programming, at least if we remain within the easily implementable imperative framework. We are willing to accept these problems in general because concurrent programming allows us to avoid having to concern ourselves with the arbitrary sequencing of processes. We cannot expect the language to shield us from these problems—it must give us enough rope to hang ourselves or it will not be expressive enough. It is our job as programmers to use the language well, to have Edsger Dijkstra's "software discipline." Of course, all of this is not to say that Max could not be made easier to use. A number of basic semantic irregularities could be fixed to make the language more uniform. On a more fundamental level, it should be possible to incorporate more functional/data-flow concepts.

Dean Jacobs

"Even if an algorithm works well, a tiny change...may break it." This may be true for a novice (as with any programming language), but Max has the potential to be entirely deterministic. As experienced Max programmers, we do not have trouble with time order in our patches. We specify explicitly the order of all events. It should be pointed out, however, that in the Opcode version of Max, there is a potentially confusing and dangerous mechanism that acts automatically on ambiguities in order; graphic positioning of connections between objects on the screen will determine the message-passing order between these objects. A programmer who does not explicitly specify order can inadvertently reorder events by simply reorganizing the graphics of a Max patch!

Cort Lippe and Zack Settel

Desain and Honing's comments about "computation time and order" and their general conclusion that it's impossible to write reliable Max patches are wrong. There is no compute-time dependency determining how a patch runs. It is possible, even easy, to make a patch whose result is undefined. (If any outlet sends more than one nondelayed message to the same object, the order in which the messages arrive is undefined in general; sometimes one can predict the result by looking at the spatial layout of the patch, but it should always be regarded as an error unless the order in which the messages are received doesn't matter.) It sounds easy to think of a design that would automatically find the "right" order in which to send messages, but after thinking about that for several years, I haven't come up with one yet. Until somebody does, the "order problem" will stay.

Miller Puckette

Max Is Not a Programming Language

Most of the capabilities of modern programming languages for data and control abstraction are absent from Max. But one has to keep in mind that not all abstractions can be expressed well graphically. Nested loops, variables, references, match patterns, computation history, and recursion are among the constructs with which the conversational mode, in the form of a formal language, can deal much better. The one type of abstraction supported is the encapsulation of a patch (a subroutine) into a new module. Some other types of abstraction that can be easily expressed graphically are not supported. Think, for example, of the "bank-of-..."abstraction that would make a bank out of any module (without manual copying) and convert inputs and outputs into vectorized data. Max supports further extensibility in the form of procedures written in the C language, instead of the ability to abstract from the primitives in the language itself.

One often sees that the number of primitive objects needed to deal with special cases grows rapidly in languages that lack appropriate abstraction mechanisms; in Max this number is already above 100. Furthermore, because Max is advertised as a language with an object-oriented flavor, one would expect a versatile system of data types (classes) and polymorphic modules (objects), which adapt their processing to the type of incoming data and their own identity. In reality the data types reliably passed between modules and easily dispatched are few and low-level.

Peter Desain and Henkjan Honing

Max appears to be weak when viewed as a programming language, yet a good programming language is the very thing I need in order to build my own abstractions and support my own concepts. I can't imagine writing the kinds of interactive pieces I write using Max. I think my central criticism (and one that echoes Desain, Honing, and Lewis) is that Max does not scale well. It is fine for prototyping small systems and is great for control panels, but it becomes unmanageable in large systems. Does anyone claim Max is good for large systems?

Roger Dannenberg

While we agree that creating complex data structures and structuring large and complex programs in Max is difficult if not pointless, we would like to stress one point that we do not consider an "escape hatch"—one may program additions to the Max environment using C. The external code resource platform provides a programmer with access to a large number of Max kernel functions, including scheduling, memory allocation, message-passing, and object functions. Using this platform, it is possible to combine the power of C with the convenience and speed of Max. In such a case, "high-level" things happen in C, lower-level things and graphics happen in Max, and development time can be quite rapid.

Cort Lippe and Zack Settel

The fact that Max allows the user to "draw" connections between units in order to define data-flow is one of its most compelling features, and makes for great demonstrations. On the other hand, even the most cursory reading of the literature of graphical programming languages (e.g., the proceedings of the IEEE Conferences on Visual Languages or the various books on visual programming) will show the difference between the use of visual cues for connection, item identity, function, and state. In Max,

one must still read the labels of icons and must know the order of operands of units in order to be able to "read" a diagram. I would contend that this falls far short of being a "graphical programming" front end. Max is extremely useful for teachers and performers in configuring parameter maps between real-time input data and MIDI commands. There are, however, only very limited models of data types, quite rudimentary facilities for functional extension, and no real data or behavior abstraction mechanisms—the defining characteristics of programming languages. The above points being made, I believe it is fairer to characterize Max as a "data-flow-oriented configuration tool for MIDI data or command maps," rather than as an "OO graphical programming language."

Stephen Travis Pope

A few professional musicians exist, such as George Lewis, who can find the time to go more deeply into computers than the "shell" level mentioned above. If you wish your computer to be more than just a musical instrument—if you want it to be an improvisation partner, for instance—you need a programming language. One thing people in this situation might want to do is write Max external C procedures. The advantage to this is that, once you get the procedure working, you can easily change the way in which you use it, for instance to adapt to the peccadilloes of a new pitch-to-MIDI converter, a different effects box, etc. People (such as Lewis) who have done concerts with live electronics will know what I'm talking about.

Miller Puckette

Max Does Not Use Graphics Well

Instead of the neat, old-fashioned block diagrams [e.g., of Music V–style instruments] that we used to see in articles (e.g., in the older ICMC proceedings), now awkward-looking Max patches are often presented—no different symbols for modules, no different line types for different signal types, and a mess of wires. Even if the mess can be cleaned up, users often have to spend more time cleaning up the patch than creating it because the graphical editor does not support state-of-the-art consistency maintenance when moving modules, multiple moves, etc. This is not just an "outside" peripheral issue. Consistent user-interface design often reflects clean internal design of a system.

Peter Desain and Henkjan Honing

Max seems to be especially suited for rapid prototyping of interfaces. Graphical input and feedback is extremely valuable in developing and monitoring real-time interactive programs, and the tools provided by Max are excellent. The need for number boxes may indicate a problem, but the availability of number boxes is an important feature. Max is also excellent for interfaces between hardware or software components, where careful control and mapping of parameters is often required.

Roger Dannenberg

The other novel feature of Max is, of course, the graphical nature of the language. On the positive side, this feature makes it extremely easy to develop the user interface associated with a given application. The alternative, rummaging around in various Macintosh widget libraries, is, at best, tedious for experienced users and, at worst, impossible for naive users. I find GUI-level debugging to be quite pleasant; it is great to be able to stick in a number box and watch values flowing through some point in a program, rather than inserting print statements or setting break points. On the negative side, as many have mentioned, it doesn't scale well; large programs tend to become heaps of spaghetti unless one spends inordinate amounts of time laying them out. I think the ultimate system of this nature should have a Max-like top level that is cleanly and simply interfaced to a more traditional textual language underneath. Note that current Max C-externals are too painful to use in this way. I would like to be able to click on a "code" object and have a window with editable C open up. This starts to sound somewhat like the NextStep environment for developing Objective C programs.

Dean Jacobs

Advanced users in areas as complex as computer music need high-level, powerful, scalable software tools for their work. This should entail graphical as well as text-based programming utilities, graphical user interface builders, and rapid-turnaround language systems. Integrated tools exist for several related domains (e.g., digital signal processing or computer graphics), that incorporate graphical programming in the data-flow style with text-based interpreted or incrementally compiled languages. Modern systems based on abstract but simple programming languages such as Smalltalk-80 or Lisp, together with integrated libraries and programming environments, can render moot the debate over shell versus "real" language versus graphical front end.

Stephen Travis Pope

Conclusion

The main purpose of this letter is to see more acknowledgment of the drawbacks of the simple types of data-flow languages for music, and stimulate more research to improve or find other ways of realizing the important needs that Max only partly addresses. Many good solutions to the problems mentioned are already available in the fields of parallel computation, data-flow, synchronization of data streams, time-tagging protocols, and from the areas of knowledge representation for musical objects and their structural relations. They are not basically incompatible with Miller Puckette's and others' vision of a simple and versatile programming environment for interactive computer music. We hope to have provoked some response to our criticism, but also to have comforted some users that undebuggability and unreliable results are not their own fault.

Peter Desain and Henkjan Honing

I think Desain and Honing's conclusion could have been a little more balanced. Yes, we can and should strive to understand and overcome the limitations of Max in future systems. At the same time, we should try to understand the elements of Max that make it so successful and retain them in future systems. The work done at CNMAT using Max as a front end to workstations on a network is a good example. We should strive for this kind of synthesis of ideas and features.

Roger Dannenberg

A number of people have suggested that the reason Max has become so popular is that it allows relatively naive users to get simple things working rapidly. There is probably some merit to this claim; learning C together with a MIDI library and a GUI library takes substantially more effort. But that is not the whole story—other perfectly reasonable alternatives, such as Mark Coniglio's *Interactor*, have not attracted as much attention. As George Lewis suggests below, we must look beyond the purely technical issues to examine the social context in which all of this takes place. For inspiration, let us look at programming languages in general. I would conservatively estimate that, at this point in time, 80 percent of the running programs in the world are written in five languages: Fortran, COBOL, PL/1, Ada, and C. What do these languages have in common, besides the fact that they almost universally sacrifice conceptual elegance for efficiency? COBOL and Ada were developed and supported by the U. S. Department of Defense, Fortran and COBOL were developed and supported by IBM, and C was developed and supported by AT&T. We must conclude, then, that the success of a language has less to do with its merits than with who backs it. In the relatively tiny world of computer music, a language that was developed at IRCAM, is named after Max Mathews, is extensively used at CNMAT, and is supported by Opcode has heavy backing.

Dean Jacobs

It's always useful to debug one's patch *before* going on stage with it; this would be true of any real-time performance environment. It's good if your environment acts the same way on stage as it did beforehand. Max can't guarantee this, but it bends over backwards to try to keep it that way.

This leaves Max open to many criticisms, both facile and not, relating to its simple-mindedness and incompleteness. Surely Max is not the last music program anyone will propose, either of the practical variety or of the "musically intelligent" variety. But Max is here now—use it, have fun with it, make music with it.

Miller Puckette

Postscriptum

Our original letter was a collection of thoughts that arose when observing users of Max—we are not expert users ourselves. It is good to read in the various responses how expert users solve the problems we identified. To our taste, however, many of the solutions still have more of the nature of clever workarounds than of fundamental solutions. We will clarify and summarize our points briefly below.

Plumbing is a wonderful metaphor, whether one uses it (for example) for connecting processes with UNIX pipes, or applications in the Apple MIDI manager, but it is only viable if one can forget about what is going on inside the pipes. This is easy to design in synchronous data-flow languages, and in the asynchronous case when the control can be made consistently data- or demand-driven. Both of these cases do not apply to Max-like languages, and it is indeed difficult to design a transparent flow of control. Max basically gives up and lets the user explicitly control the computation time and order by the graphical layout and choice of which data item will "bang" the control. In this way data-flow and control-flow are mixed-up. In most cases it is perfectly clear from which computation step an item stems, and with which items it has to be combined, even if the items followed computation routes of different length. In our vision a patch compiler would be able

to figure these things out, and "banging" would no longer be needed. We want composers to be able to concentrate on musical time (which is complicated enough), and not to confound the issue with computation time and order.

A second main issue is the limited availability of data types in Max—and the dynamic nature of its typing mechanisms, which punish a user who tries to define higher-level data constructs with slower programs. We agree that one does not particularly want to be restricted by someone else's idea of the primitive data types needed for music, but on the other hand, music is unique in the intricate way in which multiple overlapping structural descriptions are needed, vertical versus horizontal structures are relevant, ambiguous analyses are common, score versus performance information must be represented, etc. If a language only supplies low-level, quite technical data types, the extra effort needed to represent and work with the kind of structure and relations that resemble (however shallow) what performers and composers deal with (e.g., time) will be enormous. The way George Lewis approached the subject is quite enlightening—it is indeed frightening to see the technical "bang" construct of the language reflected so directly in compositions, based on one event's triggering another. The main escape route often cited to us is the possibility to incorporate large complex modules written in C into Max, and indeed such openness is state-of-the-art in language design. But would application interconnection in a MIDI manager fashion not supply that functionality in an easier way?

There is one domain in which we can wholeheartedly applaud Max—its widespread use as a platform makes a real contribution to mutual collaboration and sharing of work in our community. We learned a lot from the responses we have received to our original letter and hope that placing them together in *Array* and *Computer Music Journal* will give valuable insights to readers as well.

Peter Desain and Henkjan Honing

Max in a Musical Context

I thought I'd take some time to respond to Peter and Henkjan about their Max criticisms, most of which I feel are well taken. The only thing I have to add to their rather thorough technical analysis is my annoyance with bugs that arise from the fact that the position of objects on the screen has an effect on the data-flow and timing. The main thrust of their essay, I feel, is to find out why Max has been so widely adopted of late. To shed some light on this phenomenon, it is necessary to move past the purely technical issues (since Miller and David Zicarelli certainly are aware of these shortcomings, and I am hardly a superwizard programmer myself). We must examine the social and cultural environment in which much computer music is created.

At the demonstration of the IRCAM/Ariel ISPW at the 1992 ICMC, I asked my friend Miller Puckette about Max style. In a very thoughtful answer to my question, Miller agreed that structured Max style was difficult to achieve with programs of any complexity. I feel that the main reason for the problems of structure in the code is that Max is evidently based on an analog synthesizer metaphor. One can regard the analog synthesizer as a quasiparallel structure, in which outputs and inputs seem to be available at any time, and simultaneously. There needn't be a single signal-flow path, and you can have feedback between arbitrary points. The main problem with analog synthesizers is that you know what, but you don't know when. It is OK for voltages to behave this way, but not for a computer program that is handling musical data of any complexity. Data-flow and signal flow are two different animals; while watching someone create analog synthesizer patches, I often feel that if a program were structured in a similar fashion, it would either never work, or be too hard to debug.

In Max, the use of the analog metaphor provides enhanced "user friendliness," but brings along with it many of the structural problems associated with the analog synthesizer. In particular, the proliferation of wires to arbitrary points is a serious violation of process modularity. You can basically have data jump to wherever you want, without any control over when they should jump. It is just as difficult to manage such programs now as it was before Dijkstra started writing about computer programming.

The modularity problems led me to say that "Max is a low-level program disguised as a high-level language." I did say this to Michael Pelz-Sherman, and I also mentioned it to Miller during a visit to IRCAM in May 1992. Replying to my query about how he reconciled the analog synthesizer approach with structured programming, Miller basically said that Max was excellent for quick prototyping, while for the heavy stuff it was better to use the C escape hatch. Miller had not foreseen that so many people would use Max as a primary platform for programming, never even bothering to extend the language using C. Although the analog synthesizer metaphor indeed lies at the root of many of Max's technical difficulties, we must look elsewhere to discover why, despite a host of daunting difficulties associated with its successful use, Max has become quite popular.

Not having attended an ICMC since 1986 in Den Haag, I was surprised to discover that "interaction" in computer music has moved from being considered the province of kooks and charlatans (I'm proud to have been one of those), to a position where composers now feel obliged to "go interactive" in order to stay abreast of newer developments in the field. I can tell you that many composers who had long been working in interactive media found sweet irony in Stephen Pope's comments in a recent *Computer Music Journal* Editor's Notes about tape music becoming "marginalized."

Max lets users do simple things in a simple way, at modest cost; one can "go interactive" for $300 or so. However, the relative novelty of "interaction" in some areas of the computer music community means that very few of the newly interactive composers have had the time to investigate strategies already created by other composers, or to construct anything more complex than rudimentary mental models of how the interactions should be structured musically.

The "interaction" in many of the Max compositions I have heard lately takes place by means of the "trigger"—an analog term, ironically enough. Typically, the occurrence of a low-level MIDI event triggers the playback of some precomposed material, or perhaps a signal processing routine. That these amoeba- or roachlike automata have passed for serious interactive work in recent computer music could (uncharitably, to be sure) be simply deemed a testament to the low level of the current thinking about musical interaction.

Such an assertion may not be without merit in some cases; however, it does not account for the popularity of this kind of "interaction" in present-day computer music. Rather, recent fashion in European art music has driven composers to assert a necessary concomitance of composer control with musical structure. This has important implications for what is produced musically. A real "interactive" entity, a mammal for instance, exhibits complex behavior that cannot be simply tied to a set of controlling "triggers." Moreover, the structures present at the animal's inputs (senses) are processed in quite a complex, multidirectional fashion. Often the animal's output (behavior) is not immediately traceable to any particular input event. The number of triggers needed to fully control every sonic movement of an "interactive" composition of the complexity of a housefly would already be quite high; perhaps hundreds of triggers would be needed, far too many to be manipulated at once by anyone.

The traditional solution to this problem, particularly for those composers who had "gone interactive" prior to the introduction of Max, was to build autonomous, high-level input-parsing structures and musical behavior into the composition. The composer therewith relinquishes some degree of low-level control over every single bloop and bleep in order to obtain more complex macrostructural behavior from the total musical system. The output of such entities might be influenced by input, but not entirely driven by it.

Building such structures into a musical composition, however, would for many composers be the same as allowing performer choice or improvisation—it would violate the composer-control laws. Faced with the alternative of building and manipulating hundreds of triggers, one can see right away why so many Max-based compositions have featured such primitive, hot-button interactions—they are simply easier to assert control over.

Since behind the considerable expressive power of Max is the full power of the C programming language, I don't want to blame Max for the failure of many of its users to create interesting and complex models for their "interactive" works. Rather, what is at fault is the widely held military style, the "hear-and-obey" metaphor that, in much recent computer music, passes for interaction with live musicians. This is ultimately a set of cultural imperatives taking musically expressive form—in itself, an entirely natural occurrence.

For example, perhaps the preponderance of papers presented at the 1992 ICMC that purported to explore improvisation (particularly "jazz" improvisation) actually appeared to treat the medium, not as a primary creative medium, but as a kind of easily tackled baby problem in musical cognition. The goal in most cases was not to explore the possibilities in improvised musical forms, strategies, or heuristics, but to provide insights into musical structure that could later be used to do "serious" composition.

This ultimately unproductive attitude would have been considered rather insulting to the heritage of those for whom improvisation is a primary creative medium—had any such persons been present in even modest numbers among the ICMC participants. In a more culturally diverse musical and social environment, this kind of view would at least be disputed. The social, cultural, and gender isolation of the computer music fraternity (for that is what it is), however, combined with the lack of critical discourse, leaves such questions untouched.

George Lewis
Chicago, Illinois USA