NPE:
A New Modula-2$^+$ Programming Environment

Bob Ayers
John DeTreville
Jim Horning
Bill Kalsow
Ed Sattertwaite

October 22, 1987

Preface

This paper presents an alternative to John Ellis's NPE proposal; it was produced by modifying his document. The modifications are still underway, so watch out. The compiler, linker, and debugger get relatively more attention; editing, browsing, and the like, relatively less. Specifically, the changes from the base document include:

- More vocal support for better code quality, and for good debuggability in the presence of optimizations.
- Support for dynamic replacement. It's very easy to provide, and it's at least a little useful. If it's dangerous, it's also avoidable.
- The integrated editor and browser are disentangled from the remainder of the design. They are not part of this proposal, although the underlying functionality needed for their later implementation remains. Ivy will be supported for editing source programs; other editors will also be allowed. The user interface with System Modelling is up in the air.

1 Introduction

This memo attempts to sketch out the design of NPE, a new Modula-2$^+$ systems programming environment that SRC might build ("NPE" is a temporary name).

This environment would replace the current one in about 18 months and would serve SRC for about 3 to 5 years.

The purpose of this sketch is to increase our understanding of one specific NPE design. A specific sketch helps clarify the costs and benefits. Also, it is easier to understand the consequences of any particular design proposal given a complete design as a reference point.

1.1 Goals

Here are the goals of the NPE design:

- Smooth integration with system modeling.
- Very fast turn-around in the edit-model-compile-link-execute-debug development cycle: 10 seconds from a small editing change to the start of execution. Making interface changes will require more time, but much less than in the current system.
- Programmatic manipulation and generation of Modula-2^+ programs, including a uniform representation of types.
- Better overall runtime performance than the current system: fast procedure calls, faster program startup, other improvements and optimizations in generated code, shared libraries, allowance for a generational copying collector.
- A code generator for both the VAX and load-store machines like the PRISM and TNT.
- Execution of programs created by the NPE on Ultrix.
- Dynamic replacement: the replacement of an existing module or procedure in a running program. This will be used by the debugger; it may also be useful elsewhere.
- Dynamic loading–the addition of a module to a running program that doesn't already include that module–fully integrated with debugging and system modeling.

1.2 Non-goals

The following items are non-goals. The new system may well be built in such a way that the addition of these things later on may be very difficult.

- More than one systems programming language, including Pebble. But prototype language implementations and small languages like Tinylisp may well be built on top of the environment.

- The Unix/C style of small programs invoked very quickly from the shell. The NPE will improve program startup time by using shared libraries, allowing initialized REF-containing data structures created at bind time (like Bags), and perhaps bind-time initialization of RC maps and exception tables. But the basic module-initialization technique of Modula-2^+ will remain.

- Loading of individual procedures; a module appears in a program in its entirety.

- Super-efficient code. We are not planning to implement anything close to a state-of-the-art optimizer; the more straightforward optimizations should be sufficient for SRC's needs. Further, we are making some individual decisions in the runtime architecture that compromise the ultimate conceivable performance (which might not ever be achievable here) in favor of simplicity elsewhere. Examples: an extra memory reference per procedure call; possibly an extra memory reference per TRY scope.

- Shared libraries on Ultrix.

- Highly functional debugging of programs executed on Ultrix.

- Running the new environment itself on Ultrix.

- Highly functional debugging of modules written in C or assembly language.

- Maximally functional debugging of optimized code. It is well known that many desirable optimizations pose significant difficulties for debuggers. For example, implementing tail-recursive calls as jumps can cause inaccurate stack backtraces; elimination of dead variables can also eliminate useful clues to past program behavior. The debugger will be aware of which optimizations have been made, and while it may not always be able to shield their effects from the user, at least it will not lie.

- Configurations produced by binding modules produced by significantly different versions of the Modula-2^+ compiler. System modelling and plentiful cycles make it relatively easy to avoid building such configurations. One of the techniques used to give fast turnaround is to run the compiler during debugging, and life will be much simpler if there is only one version of the

compiler to run. Naturally, it will be possible to debug configurations built entirely with an old compiler by running the old version of the programming environment.

- Structured editing, browsing, querying, and debugging, via an integrated user interface. Greater functionality and integration may come later.
- Modula-3. However, most of the proposed "cleanups," the new type system, and perhaps inline procedures will be adopted, yielding Modula-2^{++}.

1.3 Fast turnaround

In the committee meetings last fall, fast turnaround in the edit-compile-link-execute-debug cycle was the number one or two item mentioned by most of the attendees. A 10-second turnaround for one-line changes would greatly improve our productivity, make us all a lot less grumpy, and make the construction of small and large programs more pleasant.

In our current environment, the slow turnaround is dominated by one factor: the large number of bytes being fondled at each stage of processing. SRC is not peculiar in having large libraries and binaries. It is inevitable that high aspirations (texts, threads, user-interface toolkits, RPC, source-level debugging, etc.) will lead to large binaries. From conversations with people at CMU, Sun, and other institutions using C, I've gathered that they are suffering the same problems of huge libraries and binaries and slow turnaround.

Byte fondling can be reduced using two basic strategies: retaining expensive-to-compute or frequently used information and regenerating less frequently used information on demand. More specifically, the following will reduce fondling:

- A compiler, binder, system modeller, and debugger in one retained address space per development session, sharing a cache of source and object.
- Procedure-at-a-time recompilation.
- A new method of binding (linking) object modules.
- Shared libraries.
- Regeneration of most "symbol table" information on demand during a debugging session.
- Compiled definition modules.

- Fine-grained inter-module dependency analysis to reduce recompilations.
- Dynamic loading of modules to start program execution ("load and go"). A program image ("a.out") normally won't be written into a file during the development cycle.
- Dynamic replacement of modules or procedures in an address space may also be useful in reducing byte-fondling.

2 Runtime architecture

The runtime structure of a Modula-2^+ program is defined by a set of Modula-2^+ interfaces defining the data structures and operations on those data structures. The interfaces will not attempt to "hide information" to any large degree, since there will be only a fixed, small number of clients (the compiler, the debugger, the binder, and profiling tools).

The data structures are represented as REF structures, allocated from collectible storage. Object modules and program images are stored in files by pickling the REF structures.

The same architecture (as defined by the interfaces) will be used on all the target machines.

2.1 Definitions

A Program is the runtime representation of a compiled and bound Modula-2^+ program.

A Module is that part of a Program representing one compiled source module.

An InterfaceRecord is a record containing the procedures and variables exported by an interface. A Module contains a separate InterfaceRecord for each interface it exports. Client Modules are bound to interface implementations by storing a pointer to the InterfaceRecord in the client Module; code in the Module dereferences the pointer to access components of the interface.

An "immediate value" is a value referenced by compiled code that is known at compiled time and can be referenced in the immediate field of an instruction.

A "SourceID" is the unique name of an immutable source managed by the system modeller. (See the System Modelling section.)

A "Procedure" is a one- or two-word value representing a Modula-2^+ procedure value, that is a closure of a procedure and its lexical environment (containing module).

A "RuntimeType" is a structure providing the runtime information needed for REF types, including the typecode and the RC map used by Pickles and the collector.

does this need to change to support Dynk.?

2.2 Modules

A Module contains:

- The SourceIDs of the Module's source.
- An ordered list of the InterfaceRecords exported by the Module. Each InterfaceRecord contains the SourceID of its interface.
- A LinkageRecord containing all the non-immediate values referenced by procedures in the Module but not occurring in any exported InterfaceRecord.
- The code for procedures.
- A list of the runtime types defined by this module, including any equivalences of opaque REF types (needed at bind-time).
- A PC map from PC values (relative to the start of the module) to "exception handlers," special subroutines invoked by the implementation of RAISE. No other static information is needed by RAISE.

Space for top-level variables is allocated in the InterfaceRecords and LinkageRecords; there is no additional indirection. The variables may contain initial values, including initial REF values.

The first n words of a LinkageRecord are for the n InterfaceRecords imported by the module.

As in the current Modula-2^+ implementation, a REF type declared in a module will have a corresponding internal variable allocated in the LinkageRecord or InterfaceRecord for holding the runtime type.

The code for each procedure is a contiguous, non-overlapping vector of bytes, even if procedures are lexically nested. The code is not guaranteed read-only using hardware-provided page protection. The code is position-independent. Intra-module references (including local procedures) will either be indirect through the LinkageRecord or else will use pc-relative addressing (less likely).

The InterfaceRecords, the LinkageRecord, and the code are contiguous within one module.

2.3 Programs

A Program contains:

- The SourceIDs of the program and its system model.
- A map from SourceIDs to Modules contained in the Program.
- A map from typecodes to runtime types.
- A map from virtual address ranges to Modules (a Module occupies contiguous storage).
- The initial starting address.

2.4 Binding

Inter-module binding occurs at several different times:

- When a program image or shared-library image is created.
- During load-and-go startup of programs.
- When a shared library is mapped in at program startup.
- Dynamic loading of new modules during program execution.
- Dynamic replacement of old procedures during program execution. new

In all cases, the binding occurs on in-memory REF structures of the runtime architecture, so a single, simple implementation of Bind suffices.

The Bind operation is:

Bind(Program, Module, list of <SourceID, i>) & ReBind?

This adds a Module to a Program. The ordered list of <SourceID, i> is obtained from the system modeller, and it specifies the InterfaceRecords to be bound to the imported interfaces of the module. <SourceID, i> names the ith InterfaceRecord exported by the module named SourceID.

Bind updates the SourceID→Module map, the Address→Module map, and assigns typecodes to the module's runtime types.

and sorts the list of init calls?

2.5 Debugging maps

The debugger maps addresses to procedures and variables using multiple levels of mapping. The Address→Module map maps the addresses of procedures and variables onto Modules. (Modules are stored contiguously, so this map is very simple.) The debugger then uses the AST for a Module (not contained in a Program) to map an offset in the module onto source procedures and variables. The AST interface is responsible for providing this offset mapping, using information stored or recomputed in the AST alone.

2.6 Calling sequence

Our programs currently spend 25% of their time in the calling sequence. By switching to a minimalist calling sequence we shall get a two- to four-times across-the-board speedup in procedure calls. The same calling sequence will be used for both the VAX and load-store machines like the PRISM.

Most arguments will be passed in registers. Callers must save any live registers before calling a procedure (caller-saves). The stack pointer and the collector transaction queue are the only dedicated registers. The return address is passed as the 0th parameter in a register. Only move, add, and jump instructions will be used in the sequence.

Most procedures will have fixed-size stack frames, using the stack pointer directly as their frame pointer. Frames will not be explicitly linked together; instead the debugger and the exception handler trace back by using PC maps.

Many "leaf" procedures won't even need a stack frame and won't do any memory references in the calling sequence. The few procedures that have dynamically sized frames will store their size in the frame.

Nested procedures will be implemented using traditional static links; the static link for a procedure points at the stack frame representing the lexically enclosing scope, and that frame will contain a static link to its enclosing scope, etc. Static links are passed as implicit parameters.

A "tail-recursive call" (as defined in the literature) is a call that's the final (dynamic) action of a procedure and that yields the same result type as the procedure. Tail calls will be optimized, saving a few more instructions.

The combination of argument passing in the registers, caller-saves, and tail call optimization will be especially effective when programs have a large amount of "locality of arguments;" that is, when the ith argument to a procedure is often passed on unchanged or only slightly changed as the ith argument to a sub-procedure that is often tail recursive.

The code generator will be written with the demands of the calling sequence in mind. It will try to target the evaluation of argument expressions into the appropriate registers, and it will reload values saved across procedure calls only on demand.

Raising exceptions with this sequence will be somewhat faster because RAISE will not have to examine register masks or restore saved registers as it traces back through the stack. Because of caller saves, a TRY handler is guaranteed that on entry all variable values are in known memory locations.

As in the current Modula-2$^+$ implementation, a global table maps PC values onto exception scopes and their handlers, but unlike the current implementation, most of the work will be done in the handlers (Tinylisp currently uses such a scheme). For any PC value, there is at most one exception handler; as an exception propagates back through the stack, each frame's handler is called in turn. The handler decides whether or not it wants to handle the exception (by examining the exception code and executing FINALLY or EXCEPT clauses if appropriate). The handler returns (to Raise) the frame's return address and the address of the previous stack frame, so there don't need to be any global procedure descriptors describing the layouts of stack frames. All told, we should get a speedup in raising exceptions by at least a factor of two compared to the current scheme.

This calling sequence is significantly different from the VAX and the PRISM standards. Calls between C and Modula-2$^+$ will occur through intermediary stubs generated by the Ultrix object file converter.

See John Ellis's message of 3/19/86 on src.mp for a more detailed discussion of calling sequence issues.

2.7 Shared libraries

A shared library is a set of bound modules, relocated to an absolute address and shared among many address spaces running on a machine. To increase the sharing and to avoid discouraging development, there will be many smaller shared libraries, not a single monolithic one. Examples include srclib, RPC, trestle, vbtkit, and sx/tinylisp.

When a program is bound against a shared library, a stub is placed in the program startup code that at startup time makes a call to the operating system to map the library into the program's address space. The library is mapped copy-on-write (the "map" operation is simply a faster method of placing a copy of the library in the program's address space). The library is then bound to the rest of the program using interface-record binding, just as if the library had been placed in the address space by dynamic loading. The location at which the library is placed is not nec-

essarily known at compile or program-image bind time. A section of the address space (say 4%) is reserved for shared libraries.

When a program requests a shared library, it will get the exact version specified in the program's system model. To reap the performance benefits of sharing in the presence of frequently changing libraries, we'll need to apply administrative willpower to periodically rebind programs and test them.

Whence comes the the mapped library? There are two alternatives: the library could be mapped directly from a file, or it could be mapped from a special library segment global to the machine.

Mapping the library directly from a file requires copy-on-write, map-from-file in VM. Each library (srclib, tinylisp, etc.) would be manually assigned a non-overlapping section of the library space, with generous allowances for growth of future versions.

Alternatively, a special address space would be established as a repository for loaded shared libraries. When a program requests a shared library, the operating system would map copy-on-write the appropriate section of the program's address space onto the section of the library space containing the desired library. If the requested library currently isn't loaded in the library space, the operating system would first dynamically load the library from disk. (A small program running in the library space would execute the load and map requests on behalf of the operating system.) This scheme is similar to the one used on Apollos.

But unlike the Apollo design, multiple versions of libraries would be accommodated naturally, since in the library space each library is self-contained and wouldn't be bound with other libraries. For example, "tinylisp" depends on "srclib," but in the library space "tinylisp" isn't bound to "srclib;" the binding between them occurs when copy-on-write copies are mapped and bound into the program's address space.

How do the two mapping methods compare? From the programming environment's viewpoint, map-from-file is simpler and faster. A library is paged in from disk incrementally, on demand, so mapping in a library not currently shared with other programs is faster. And presumably, exec-ing a program could also use map-from-file (reducing the need for shared libraries). On the other hand, it might be easier for the VM folk to implement restricted copy-on-write mapping to another address space rather than general file mapping.

2.8 Object files and images

Modules and Programs are stored in files by pickling them.

A new operation in Pickles, WriteAbsolute, will write a pickle so that it can be

read back into memory at a specific absolute address without any relocation. This will be used to make executable images of Programs and shared library images (if we use the copy-on-write-map-from-file approach).

One complicating factor in producing executable images: The current reference-counting algorithm uses an overflow table to store those few counts that have overflowed the small number of bits in the object header. So a small amount of work might have to be done by stubs at program-load time and library-map time in order to establish these overflow counts.

2.9 Including C and assembler modules

Programmers will be able to hand-code an entire implementation module using "as." Such modules will use the calling sequence of the runtime architecture. A very simple converter will take the .o files generated by "as" and an interface and generate a valid object file. Names in the module will be matched up with the interface using a naming convention, and everything in the data segment will be placed in the linkage record.

We have only a small amount of assembly code that will need to be ported to the new environment. The Nub has a fair amount of such code; but the Nub will probably be the very last program converted to the NPE. The collector can revert to Modula-2^+, since it executes in background anyway. As now, there will also be a small amount of "engine room" code that can't conveniently be expressed in Modula-2^+ proper.

A collection of C modules can be included in a Modula-2^+ program using a stub generator and the .o converter. The stub generator takes the Modula-2^+ interfaces representing the Modula-2^+ to C calls and generates assembly language ("as") stubs that convert from the Modula-2^+ calling sequence to the C sequence. The stubs and all the C modules are then linked using "ld -r" into a single Ultrix .o, and that .o is then run through the .o converter to produce an object file. The use of stubs will make C calls more expensive.

The initial release of the environment (the first 18 months of effort) will provide only very minimal support for including C modules. Programmers will have to write their own stubs by hand. Modula-2^+ routines will be able to call C routines, but not vice versa (except in a few restricted cases using special hand-coded stubs). There will be no debugging support, not even symbolic stack trace-backs (the debugger will show Modula-$2^+ \rightarrow$C calls, but not C$\rightarrow$C calls).

2.10 Why the hard-assed approach to compatibility

Providing general compatibility between different runtime architectures, while feasible, takes a lot of effort. Specifically, converting between calling sequences, allowing for the transaction-queue register, exceptions, threads, interface-record binding, and debugging all will need non-trivial amounts of attention to provide compatibility. To avoid the slippery slope and devote our energies to real research issues, we need to be extremely hard-assed in assessing what is "necessary."

A complicating factor here is The Nameless Thing. If the TNT materializes, *someone* (not this project!) will likely end up porting a C compiler. That means this project would have to deal with three runtime architectures and object file formats: it's own, C on a VAX, and C on a TNT.

Why do we need to include C modules in a Modula-2^+ program? Our current uses of C in Modula-2^+ programs are purgeable without great effort: We can replace "yacc" with "llgen" or Eric's "yaccm," it's trivial to write lexers by hand, and we could hire a starving NA grad student to rewrite Math with a week or two of effort.

The only planned major use of C is to allow our Modula-2^+ programs to use X servers. The client-side library, written in C, will need to be linked in with Modula-2^+ programs. (Aside: Greg and Mark say that the X developers will need to do a fair amount of work before X will be usable by multi-threaded clients.) The X library doesn't do up-calls to its clients.

3 AST: Public interface for manipulating program source

The AST (Abstract Syntax Tree) interface provides structured manipulation of program source at the module level and below. (The system modeller provides operations above the module level.)

We have many applications that do such manipulation already: the compiler, the debugger, the pretty printer, flume, the Tinylisp stub generator. Future applications include: a structure editor, the system remodeller, a definition-use database, source-based profiling and testing tools (for example, verify that a test suite executes all statements), Larch, perhaps a Web-like tool for integrated documentation and source.

By having a single interface that all of these applications used to manipulate source, we would have smaller, more robust tools with greater functionality. In addition, such an interface would make it easier to experiment with new tools.

The danger in using a single interface is inefficiency: Attempting to provide generality or simplicity often conflicts with the desire for utmost efficiency in crit-

ical components like the compiler. In this instance, I think it is possible to reach a workable balance. We'll introduce some inefficiency into the interface but more than make up for it by doing fine-grain dependency analysis and procedure-at-a-time recompilation, greatly reducing the total development-cycle time compared to the present.

The AST interface will be designed for the immediate potential clients (the compiler, the debugger, the pretty printer, and flume). It will not necessarily be compatible with current implementations of those clients. Only after the first design is complete will work on the compiler begin (and of course, there will be refinements or complete redesigns during the course of implementation.)

3.1 The AST

The AST interface will provide a decorated abstract syntax tree representing a module. An abstract syntax tree is simpler than strict parse tree for a particular grammar, but it is close enough to the concrete syntax to allow the source text to be faithfully regenerated. Most clients won't need all the information used by the compiler, so the AST will be accessible after each of these main phases:

- After parsing, with retained comments options. (Pretty printer, a structure editor)
- After name resolution and type checking. Name resolution binds each occurrence of an identifier to its definition, and type checking assigns a type to each name and expression. (System remodeller, source-based profilers)
- After complete code generation.

(A "definition" is any Modula-2^+ construct that introduces a name into a scope: PROCEDURE, VAR, TYPE, IMPORT, FROM, etc.)

Choosing an actual representation will be a tricky balancing act between efficiency, generality, and simplicity. AST nodes will be defined as Modula-2^+ REF records with some fields marked as private to the AST implementation. A standard object-oriented vector-of-procedures will be provided for clients who desire simplicity over efficiency. For example, Type(node) would return the type of any node. Some kind of method-list scheme will provide extensibility for individual clients. The compiler itself will probably short-circuit the object-oriented interface and reference the node records directly. (Unlike many object-oriented applications, the number of distinct node types supported by the interface is fixed.)

There will be no "symbol table" in the traditional sense. Each AST node defining a new lexical scope (module, procedure body, record definition) will have a map from names declared in the scope to the AST nodes representing their definitions. All information about a name is stored in the AST node for the definition. The name and type resolution phase binds each use of a name (leaf identifier nodes) to the corresponding definition node.

Names from imported interfaces will be handled the same way. An imported name will be bound to the AST node representing its definition in the interface. This AST node is obtained from the previously compiled interface.

The AST representation for a type definition will also be used as the runtime representation of types used by the GC, pickles, the debugger, etc. (Of course, the GC will still have its specialized structures inside the type representation.) This approach would facilitate a "runtime type" mechanism like Cedar's that allows more general introspection by programs; but this is not an immediate goal of this proposal.

3.2 Compilation

Compiling a module results in two structures, an AST and a Module that is exactly the runtime representation of a module. Both are REF structures that will be pickled into separate output files stored by the system modeller. Compiled definition modules have an AST only.

Past experience with programming environment projects shows that fully decorated ASTs are very large and that writing and reading them is very expensive. But for our purposes, it isn't necessary to store the entire AST. Instead, the sub-trees representing procedures bodies will be NILed out when the AST is pickled, greatly reducing its size.

When a client such as the debugger needs the AST for a procedure body, the AST interface will regenerate it on the fly (if necessary) by recompiling the procedure. (Each AST node contains an offset into the immutable source file.)

The immutable source files and unpickled object and AST files will be cached in cooperation with the modeller. The compiler and debugger will be packages bound into a single address space sharing the same cache, and that address space will be retained during an entire development session. Thus, fully recompiling an average procedure should take less than a second (Lightspeed Pascal compiles at 200 lines/second).

Normally, the detailed debugging information for individual procedures is neither generated during compilation nor stored in the AST files. Instead, if the debugger needs the PC to AST node map or the PC to variable location map for a

particular procedure, it will regenerate code for the procedure, asking the compiler to retain debugger information. The modeller guarantees that the correct source will be used.

Currently, only a tiny fraction of such debugging information is ever used during the lifetime of a program. By regenerating it on demand, we'll greatly reduce byte fondling, speeding up the overall development cycle and debugging in particular.

3.3 The compiler

The "compiler" is merely a package that implements the AST interface. The compiler is bound into the same address space as the binder, the debugger, and perhaps the system modeller, sharing the cache of source and object. There will be nothing new or fancy about the actual compilation algorithms; what is different is the AST interface, the ability to do procedure-at-a-time recompilation, and the use of pickles to store the ASTs.

The separate, classical procedure-at-a-time optimizations such as common-subexpression elimination and loop-induction-variable simplification may not be implemented, though the compiler design will allow for them. Optimizers are still expensive to build and maintain, and their overall marginal benefit in our systems environment is small; the extra 10-15% speedup probably isn't worth the man-year of effort. And no one has expressed interest in mounting a full-scale research effort in "super compilers" (which typically involves several full-time people).

However, careful attention will be paid to basic code generation, procedure calls, and register allocation (essential for fast procedure calls). Just writing a new code-generator from scratch will help us avoid the code-generation stupidities of the current compiler; we should end up with code quality roughly comparable with the produced by the portable C compiler.

The same code generator will be used for both the VAX and the PRISM. It will generate pseudo-instructions for a load-store machine model, and a machine-specific peephole optimizer will convert the pseudo-instructions into actual VAX or PRISM code. Treating the VAX as a load-store machine instead of attempting to use its complicated addressing modes will not only simplify code generation but probably result in faster code. (See Dick Sites's memo to VAX compiler writers.)

The register allocation will be done on-the-fly during code generation, using a simple priority scheme to choose values that need to be spilled to memory. Actual arguments to procedure calls will be targeted into the desired argument register. Unlike the current compiler, all values will by default be assigned to registers, not stack frames. On-the-fly allocation works well because the number of live values

referenced between procedure calls in systems programs is small, and the caller-saves calling sequence forces all live registers into memory.

No attempt will be made to do inter-procedural register allocation. David Wall's experiments showed that, for his benchmarks of large systems programs on a 56-register machine, interprocedural register allocation (with arguments passed in the registers) was only 10 to 20% better than procedure-at-a-time allocation (with arguments *passed on the stack*). It's likely that much of that improvement was due simply to passing arguments in registers, and that a fast calling sequence and programmer-specified procedure integration will gain most of the benefit of inter-procedural register allocation.

3.4 Compiler versions

Any one instance of the NPE tools assumes that the same version of the compiler was used to compile all the programs being manipulated. The debugger will balk when asked to debug a program compiled with a version of the compiler different from the one it is using. The debugger and AST interface rely heavily on this assumption, since they assume that they can recompile source to recover information such as PC/statement maps.

Thus, when a new version of a compiler is released, an initial flurry of recompilation will be required. This can be alleviated by using midnight cycles and the modeller to recompile commonly used programs, shared libraries, etc. More extensive and careful testing of the compiler, using large test suites, will reduce the need to have frequent releases. (Also, the compiler implementor can declare to the system modeller that a new version of the compiler is the "same" as the old version.)

Of course, when presented with a program corpse built with some old version of the compiler, a programmer can always invoke the corresponding instance of NPE tools; the system modeller guarantees that the old version will be available. There could be a significant time penalty starting up the old tools.

3.5 Fine-grained dependency

A single fine-grain dependency analysis will be used to reduce compilations due to interface changes and to changes within a single module.

A straightforward approach to dependency analysis would be to record in each compiled definition the attributes of external names used in the compilation of that definition. For example, if a procedure P referenced some type T, the full definition of T (including any definitions T depends on) would be recorded in the compiled

P. To see if P needed recompiling, one would compare the recorded definition of T with its current definition.

Greg Nelson has generalized such a scheme in his Juno proposal. But there's no need for such generality here, because there is essentially only one attribute that figures in dependency analysis: the "compile-time value" of an external definition. The compile-time value of a type is its definition, the compile-time value of a constant is its value, the compile-time value of a procedure or variable is its type.

Recording the entire value of an external name in each dependent definition is probably too costly. Instead, some kind of fingerprint or unique id representing the value could be used. The fingerprint would be computed from the AST, not the source text, filtering out changes to whitespace or comments. Alternatively, one could assign a new unique id to a definition each time its AST changes.

The system modeller knows only about coarse-grain dependencies of the form "module M depends on interface I." When I changes, the modeller will ask the Modula-2^+ compiler to recompile M; the compiler will check the top-level definitions of M and recompile only those whose dependencies have in fact changed. Similarly, after the user edits some of the top-level definitions inside M, the compiler will check all the top-level definitions for those whose dependencies have changed.

4 Debugging

The debugger resides in the same retained address space as the compiler, the system modeller, and the other components of the NPE, simplifying its communication with them. As now, the debugger will use the teledebugging protocols to manipulate the address space of the program being debugged. Significant parts of the debugger will be written from scratch as part of the NPE development.

4.1 Data structures

The debugger reads and writes the runtime representation of the program (memory image, thread states, etc.), as guided by the ASTs for the modules. The ASTs provide all available information about a module's definitions and the results of its compilation, including the runtime type representations of all the types in the module. The ASTs do not include information on the results of linking, such as the modules' absolute locations in memory; this information is obtained from the runtime representation.

An AST for a module is generated during compilation and cached by the system

modeller as a derived object. As with all such derived objects, it may be necessary to regenerate an AST on demand by recompiling the module. Further, the ASTs are only skeletons, with procedure bodies and detailed debugging information normally deleted. When the debugger needs detailed information about a procedure, such as the map between PC ranges and expressions, the AST interface may have to recompile that individual procedure to produce the requested information. (For modules in the "working set," the sub-tree in the AST for a procedure may well already exist, in which case only the code generator need be reinvoked to produce the detailed debugging information.)

Given a Program, the runtime representation of a program, the debugger obtains the SourceID of the system model for that program. Also in the Program are maps from PC ranges onto Modules, and from SourceIDs onto Modules. From a Module, the modeller obtains the SourceID for its source (from which the AST can be derived by asking the system modeller). Once the debugger has an AST for a module, detailed information about the contents of the module (such as a map from offsets in the module onto procedures and variables) is obtained by querying the AST interface directly.

4.2 Performance

On average, the debugger will respond much faster than the current implementation. The hierarchical layering of the program representation means that on any one user interaction, a much smaller number of bytes need be examined. The entire "symbol table" for a program is never examined in whole.

When the debugger is first directed towards a Program, it examines a small table and the system model for that Program to discover the contained modules. Then to produce, say, a stack trace, it examines only the skeleton ASTs for the modules referenced by the procedures on the stack. Only when the user requests to see a particular location in a procedure (e.g. to set a break point) will the detailed information be requested for a procedure (typically resulting in a once-only, one- or two-second recompilation of that procedure).

The system modeller will cache the ASTs as derived objects, so usually they won't have to be regenerated; the modeller also caches the in-memory unpickled representation of an AST, so ASTs recently referenced will be even cheaper to access. The by-procedure cacheing of verbose, detailed debugging information such as PC maps reduces the total size of ASTs and makes access to them much faster.

4.3 Functionality

The debugger's functionality will be very similar to Loupe's, with the differences outlined here.

The debugger will be integrated with Ivy to provide at least the features of Loupe's Emacs mode. For example, to set a breakpoint, the user can point directly at the source.

The initial release may or may not contain features such as the monitoring of variables or a data browser, depending on who signs up and their particular ambitions. But in general the uniform interface and the AST structure will make it easier to provide new facilities.

More and better information about the program will be available to the debugger.

Some of the increase in information is due to greater opportunity; the new compiler and AST mechanisms make it possible to describe more accurately the program structure and the results of compilation. For example, single-stepping and breakpoints will be at the level of expressions, not statements (unless optimization is enabled).

Some of the increase in information is by necessity. For example, because of the greater variability of calling sequences, each procedure has associated information residing in the AST describing its stack-frame layout; this allows the debugger to locate the return address.

Some debugging features will be available only if preplanned. For example, additional classes of program tracing may be implemented by the debugger by modifying the program's AST to perform additional runtime actions, then recompiling the AST. After this is done, the program must be restarted from scratch; there is no facility for changing the contents of a running program.

Similarly, in the initial release the only support for debugging programs running under Ultrix is manual insertion of print statements, plus a hexadecimal stack backtrace mechanism called from the program. Later releases may provide better support, especially if it proves easy.

The new compiler may support a number of sophisticated optimizations, which can be optionally invoked. If a procedure is compiled with these optimizations, the operations supported and the information available will be restricted. If necessary, the user can recompile the program turning off all such optimizations, and then restart it. This may be very inconvenient; that's an argument for good debugger support, and dynamic replacement may also help.

The only debugging facility supported for modules not written in Modula-2^+ will be this simple stack backtrace, plus machine-level debugging. (The first release

may not provide a symbolic stack trace for non-Modula-2+ procedures.)

5 Editing

Source programs are represented as text files; Ivy is typically used to edit these files. The compiler takes text files and produces Modules and ASTs.

To provide procedure-at-a-time recompilation, the compiler will provide a "recompile" operation. Given a source file, a set of changes made when editing from an earlier version (e.g., the runs of characters affected), and the results of compiling the earlier version (the Module and the AST), the compiler will avoid unnecessarily duplicating old work. Ivy will support use of this facility.

Ivy will also provide simple facilities for displaying source positions to the user (e.g., for error messages, or when the debugger reaches a breakpoint), and for letting the user select source positions (e.g., for setting a breakpoint).

User interaction with the System Modeller has yet to be addressed; it may be easily separable.

Future structure editors may directly manipulate the AST structure of a Module.

6 Interface to System Modelling

The interface between system modelling and the other NPE components is still quite sketchy. It will get somewhat clearer after the initial system modelling manual is finished.

The system modeller will live in the same address space as the rest of the NPE (compiler, binder, etc.)

6.1 Naming source and derived objects

How do clients of the system modeller name source and derived objects?

To support later use, ASTs and compiled Modules contain references, called "SourceIDs," to their corresponding source modules. These references must not only include the immutable source text but also all the context (imported interfaces, switches, etc.) used to derive those objects, so that individual procedures can be quickly recompiled on demand from the debugger. Thus a SourceID for a derived object can be thought of as a pointer to the exact spot in a system model that produced the object. This "spot" is in fact a function application, applying the source to all of its actual parameters (interfaces and switches). From a SourceID,

modeller clients can obtain just the source text or they can explicitly re-execute the application to get a derived object.

Subtle point: Many different applications could produce the same exact derived object. But the ASTs and the Modules need to contain a SourceID for only one of the applications (the one that originally produced the bits), since by definition all the applications yield the same result.

ASTs also contain references to the ASTs for the imported interfaces. These references are also represented as SourceIDs.

Derived objects (Modules and ASTs) are represented to modeller clients as REFs. The modeller does most of the dirty work of cacheing the pickled representations of those REFs in files.

The load-and-go binder executes in a separate address space and thus will need another way of naming the object modules that live in the programming-environment address space. One method would be to use the REFs themselves as opaque names; the load-and-go binder would do some kind of special cross-address-space calls (e.g. nub calls) to transfer the bits of a Module into its own address space (RPC may be too heavyweight for the load-and-go stub, which must start up in a fraction of a second).

7 Some Performance Issues

Some isses that have come up so far:

7.1 A budget

Here is a guess at a budget for the 10 seconds from the time the user finishes editing a 50-line procedure and says "go" to the point at which the first programmer-supplied line of the main module begins executing. This is just educated guesswork, based on currently Firefly timings; I've tried to be conservative about speeds and sizes, pretending that CVAX doesn't exist.

seconds:

.5 Compilation. Mac products compile 150 to 200 lines per second. Assume we can compile in-memory (with good AST cache hits) at 100 lines/second.

1.5 Constructing the object module and AST and pickling them. Internal pc-relative addresses need to be adjusted for all procedures at this point. Assume there is a 20K object module, a 20K AST (this is probably too big), and that the pickler can pickle at 20 usec/byte (too slow).

2.5 The loading part of "load and go." The Ivy server's object modules consume about 280K; Tinylisp object modules about 330K. So assume a large application has 400K of application-specific code not in libraries, and unpickling goes at 6us byte (25% faster than the current pickler, but assume object modules don't have much fine-grain ref structure).

0.5 Mapping libraries and dirtying 400 module data pages (1ms page).

1.5 Module startup for application and programs. Includes registration of application exception scopes, but most of the work of registering library exception scopes already done. Includes registering of typecodes (noise).

6.5 Subtotal

This leaves 3.5 for all system modeller-related activities. Roy believes that shouldn't be any problem; he will provide a more detailed budget later.

7.2 Can we afford delayed binding?

Of course, interface records impose an extra memory reference on every inter-module procedure and variable reference. Can we afford that?

In our current Modula-2^+ implementation, 25-30% of execution time is spent in the calling sequence, and the average call takes about 20 usecs. An extra memory reference for the indirection will cost 1 usec, so we've increased the cost of a call by 5%, increasing total execution time by at most 1.25%. Not all procedure calls are inter-module calls, so the actual increase will be less. I don't have good estimates of the usage patterns of global variables, but my guess is that the effect on them would be even less than for procedures. (A very simple, specialized common-subexpression analysis could identify multiple references to an interface record pointer.)

So we could afford delayed binding in the current environment. But don't fall into the trap of saying, "Ah, if you're speeding up procedure calls by a factor of two, three, or more in the new environment, then the relative cost of delayed binding will be higher." That's true, but mostly irrelevant. We're mainly interested in a holistic comparison of the current environment with the new one.

Overall, programs will run faster in the new environment due to much faster procedure calls, slightly better non-call code, and reduced demand on VM through shared libraries. We'll be sacrificing a small part of the potential improvement for the benefit of faster turn around and a much simpler implementation. As was

pointed out in the SRC review, one of the ways we can profitably use the increasing number of cycles is to delay binding.

7.3 Page size versus shared libraries, binding, and threads

Compared to the VAX (with a page size of 512 bytes), the larger virtual-memory page sizes of Prism (8K) and TNT (16K) create problems for the NPE. Shared libraries and protection against thread-stack overflow become harder to implement efficiently.

A shared library gets mapped into an address space copy-on-write, bound with the rest of the program, and then its initialization procedures are called. This will dirty data pages of the modules containing the init flags, global mutexes, initial constants which are really global VARs, typecode cells, etc. Further, if the shared library depends on another shared library, then the linkage vectors of the first library will get dirtied by pointers to the interface records of the second library.

The data and code of a module is stored contiguously for each module. So at least one page from each module will be dirtied on startup.

To get an idea of the problem, suppose an application includes the following shared libraries:

srclib tinylisp uilibClient Network windowsClient vbtkit facades

These contain 248 modules whose text + data is larger than 300 bytes. The average size is 6600 bytes. Here's a histogram of the sizes:

size		number of modules
0-	4095:	148
4096-	8191:	41
8192-	12287:	18
12288-	16383:	17
16384-	20479:	5
20480-	24575:	7
24576-	28671:	3
28672-	32767:	2
32768-	36863:	2
36864-	40959:	1
40960-	45055:	1
45056-	49151:	1
53248-	57343:	1
69632-	73727:	1

srclib:	123 modules
tinylisp:	27 modules
uilibClient:	65 modules
windowsClient:	9 modules
vbtkit:	64 modules
facades:	15 modules

To the degree that the page size approaches the average module size, sharing will be reduced. For example, if pages are 1K and one page on average is dirtied for each module, then 1/6 of the total library will be dirtied on startup. But if pages are 8K, then the entire library is dirtied.

But why, you ask, aren't the data and code segments of modules stored in separate contiguous units, as on Unix? Then there would be no fragmentation of dirty pages, and a module with a single global variable (such as its init flag) would not need to dirty an entire page. Unfortunately, separate data and code segments makes fast binding, load and go binding, and shared libraries harder.

In the current proposal, the PC could be used as a base register by procedures in a module to access the linkage record and interface records exported by that module (the linkage record contains all non-immediate, non-compile-time values referenced by the procedure code). The linkage record, the interface records, and the procedure code are stored contiguously at offsets known by the compiler when the module is compiled, and thus PC-relative addressing can be used to reference them. The TNT (and Prism?) will provide 16 or 18 bits of displacement, enough to handle almost all modules. For the rare larger modules, an extra add instruction at procedure entry can be used to establish a base register pointing to the linkage and interface records.

If the data and code segments were separated, then PC-relative addressing couldn't be used by procedures to address the linkage and interface records. Procedures would have to be represented as explicit closures (a code pointer plus a data pointer). This has several disadvantages:

- The calling sequence will have more instructions to dereference a closure, loading the data pointer into a base register and obtaining the code address. The base register will generally be a live register and must be saved across procedure calls. (The Prism sequence is a good example of such a sequence.)

- Binding becomes more more expensive. The closures have to be created when modules are bound together (during program-image binding, shared-library creation, and load-and-go binding). Thus, binding is proportional to the number of procedures plus the number of interface imports, instead of merely the number of interface imports.

- Binding becomes more complicated; no researcher at SRC wants to spend months and months implementing complicated binders. Complicated binding schemes tend not to get implemented (as in Lampson's "Fast Procedure Call").

Another disadvantage of having separate code and data segments is that unpickling a module will be slower. It's pretty clear that the pickler will not run at near-disk speeds unless the objects being unpickled are large. But splitting each module into two separate segments means that the average size of an unpickled segment will be 3K instead of 6K. If we're going for broke to get 10-second turnaround, every little bit helps.

We currently unmap "guard pages" to protect against stack overflow. This requires that thread stacks be allocated in whole pages. My measurements indicate the average stack has about 4K dirty bytes. If the page size is larger than that, say 8K, the physical memory cost of a thread will double. For a machine with 2000 threads (not unreasonable, perhaps, in the next few years), that's 8M of extra, unused allocated VM just for the thread stacks. Not only will this cost physical memory, but it will increase swapping demands. Even inactive threads will have to be swept by the GC, and that extra 4M will have to be paged in each collection.

Explicitly checking for stack overflow on every procedure call increases the overhead of procedure call, which is unacceptable. A fancy compiler could alleviate the problem by eliminating the redundant checks from the global call graph, but conditional paths of execution, recursive calls, dynamically sized actual parameters, procedure variables in object-oriented systems, the large percentage of inter-module calls, and the overall difficulty of doing global optimization lead me to suspect that this is not a viable option in the next few years.

A hardware stack pointer (with an associated bounds) would solve this problem, allowing thread stacks to be allocated in much smaller, contiguous increments and making the dynamic growing of stacks easier. (Using page-protection traps in a safe software system is harder because of the problem of identifying which traps are legitimate stack-extension traps, and which are spurious.)

So what to do?

Virtual memory is used for things other than simple swapping–it provides:

- convenient memory allocation (by segments)
- address space protection
- sharing between address spaces
- thread-stack protection

In addition, Kai Li's and my work with garbage collection and shared virtual memory also take advantage of VM page protection. When evaluating the "ideal" page size, all these uses of VM should be taken into account.

The page size is pretty much decoupled from the size of swapping units. VAX systems already swap in units larger than the VAX 512-byte page.

Talking to Tom R, it seems the size of the page tables needn't be a key issue, even with VAX-sized pages. Provided the hardware doesn't have its own notion of page table format (e.g. it has a TLB that traps to software on misses), run encodings can be used to efficiently represent very large page tables with small or medium page sizes. Even on the VAX, such encodings can be used to eliminate the bugaboo of page-table size (e.g. Mach).

So, from point of view of the programming environment, a page size of .5 or 1K is acceptable, 2K is perhaps barely acceptable, but 4K or larger present problems.

Perhaps the next step (for the TNT) is to discuss the hardware tradeoffs involved in page size.

8 Extensibility

This proposal provides only limited extensibility. Too many other research projects have pursued the holy grail of general extensibility, trying to accommodate multiple languages, arbitrary user interfaces, and ever-expanding tool kits. But none of these projects has succeeded in producing a realistic prototype; often they pursued generality without a basic understanding of which set of tools needed it and why. Indeed, I believe that generality is one of the reasons that these "meta" projects have failed to produce large-scale artifacts.

So this proposal is taking a different tack: Let's list out the functionality of the environment that we really need and know how to implement. Then let's implement it. Repeat several times. Then step back and generalize.

This proposal provides limited extensibility in a few well-defined components:

- The AST interface allows other programs to manipulate and generate programs. For example, someone could easily write a statement profiler by transforming the target program's source, adding statement counters at appropriate places.
- The debugger will most likely continue to have programmability similar to (or exactly the same as) Loupe's lambda language.
- The Ivy text editor is extensible.
- The "macro-operations" of the NPE (controlling the edit-model-compile-link-debug cycle) may well have a programming-in-the-small interface to Tinylisp or the shell.

However, there won't be a general mechanism to add arbitrary new components to the NPE.

Will this discourage experimentation on the part of other members of SRC not formally connected to the project? If you accept the premise that general extensibility will likely cause the project to fail, then the lack of extensibility will certainly encourage experimentation (because otherwise there won't be a project to experiment with).

SRCers can use the Taos/Nub model of experimentation for the NPE. Neither Taos nor the Nub provide extensibility. And yet several SRCers (and at least one non-SRCer, Kai Li) have experimented by copying packages and building private versions. These people always have the option of going back to the NPE implementors and negotiating changes. For example, Andy Hisgen will likely be negotiating the changes necessary for the Distributed File System experiments.

One of the major reasons for avoiding extensibility is to make the design of the system simpler. If a component has only one or two clients, then it need not worry about the general case. And when necessary, we can "cheat" in the interfaces. But the lack of extensibility is independent of the desire for clean interfaces in the NPE implementation. As researchers, clean interfaces will help us understand the resulting structure of the NPE.

9 Planning

The basic language tools, including the compiler, the binder, and much of the debugger, will have to be built from scratch. The past experiences of many different language projects show that, when requirements change dramatically, it is almost always better in the long run to build anew rather than hack an old implementation. We don't want the design of our new environment to be needlessly constrained by the historical artifacts of 13-year-old Unix technology.

The Modula-2^+ language must be frozen in its current state during the construction of the environment; tracking a moving language design greatly increases the cost of implementation. (And starting out with a freshly minted, "frozen" Modula-3 would be almost as bad–as with all language design, we would end up debugging the design after 9 months or so of experience.) Most of the minor Modula-3 "cleanups," the simplification of the type system, and perhaps inline procedures will be incorporated, but none of the true extensions will be incorporated.

Two researchers and two staff members working full-time for 18 months should be able to implement the minimal set of tools needed to replace the current environment. A possible division of labor is:

- Ivy changes, pretty printer, definition-use database, integration with system modeller user interface.
- The AST interface and compiler. (I would sign up for this.)
- Binding, dynamic loading, shared libraries, the debugger.

Since tight integration with system modelling is an important goal of this proposal, careful planning with the system modelling people is crucial. However, implementation of the core tools (the compiler and the runtime architecture) is not closely tied to having a working modeller until the later stages of the project. For example, the AST interface relies on having immutable file storage supplied by the modeller, but implementation and testing of the AST interface can use mutable Unix files.

Some functionality will have to be added to pickles. The performance of pickles will probably have to be improved to reach its original goal of reading REF structures at disk speeds. But this proposal isn't closely tied to absolute pickle performance. Pickles already read and write large coarse-grained REF structures at near-disk speeds. The more pickles are improved, the finer the grain of structure we can use in the design.

This proposal relies heavily on REF structures, so we'll want a highly optimized allocator and collector, though we could live with the current one during development. Given that we're implementing a compiler from scratch, the amount of work needed to finish the fast reference-counting design and tune the allocator is small and not very risky, though someone would have to implement trace-and-sweep. However, the compiler and the runtime will be written to allow for a generational copying collector, which has the potential to be cheaper, but is a riskier proposition.

The OS people will have to provide some sort of copy-on-write for shared libraries, and provide some minimal support for managing the table of mapped-in libraries.

Someone in RPC land will have to rewrite "flume" using the AST interface.

The new environment will coexist with the current one for many months. Since the new environment will require very few changes to the operating system, the same machine can run programs from both environments at the same time. The lab can convert slowly at its own leisure. Taos and the Nub would probably be among last programs converted. On average, each member of the lab would probably spend a few weeks converting, but those weeks of effort would be distributed over a longer period.