# THREADS

ISSUE 2 · FALL 1996

A Modula-3 Newsletter

*Foreword*
## Remembering Geoff Wyant

Bob Sproull, Sun Microsystems Laboratories

*Geoff Wyant, a dear friend of the Modula-3 community and an editor of Threads, died in a tragic plane accident on Father's Day, 16 June 1996. Geoff is survived by his wife and two young children.*

*Bob Sproull, director of Sun Labs East where Geoff last worked, shared a few words with us that eloquently characterize Geoff's spirit. For more information, visit Geoff Wyant's Memorial Page at:*

http://www.cmass.com/people/geoff/.

Geoff was a smart, accomplished computer scientist. Since graduating from Ohio State, he worked at Harris, Apollo Computer (since acquired by Hewlett Packard), Centerline Software, and most recently with a group of about twenty of us at Sun Microsystems Laboratories here in Chelmsford. He worked on hard problems--how to coordinate the activities of many computers operating together in a network--and brought to these tasks both a mathematician's precision and an engineer's desire to build elegant systems. As his career developed, he took on increasingly difficult problems with ever greater personal leadership. This year, he was a principal investigator on a research project seeking to unify diverse databases that are spread around a computer network.

But enough of Geoff's technical accomplishments. Even those of us who can understand his contributions find our memories of him are more personal than technical.

Geoff was an ideal colleague. He was smarter than we are, so we learned a lot from him. He read the technical literature voraciously, and was a fountain of information about obscure projects and technologies. He worked well in teams, but certainly best just sitting with you, explaining something. He was a patient teacher; we have all been his students.

Geoff's standards were high. We will all remember a unique facial expression--eyes wide, his whole face in a silent smile of victory--that he flashed to signal uncovering the subtle flaw or hidden weakness in a technical argument. This usually happened during a seminar, after Geoff himself had asked the penetrating question that made it all clear.

But probably most of all we remember Geoff for his humor. He was deceptively quiet--some might say shy--until he delivered one of his trenchant one-liners. Not so much a witticism as a sharp insight from a different point of view that forced you to think, then howl with laughter. It's difficult to describe his sense of humor: fresh, off-beat, never conventional or repetitious.

We didn't see much of Geoff's family at work, but we knew they came first. On Sun picnics and outings, Carole and Rebecca and Gregory came along, and we all watched Geoff lovingly tend to them and their enjoyment of the event.

Geoff loved his life and his work, and injected some of that spirit into each of us. We will miss him as a colleague and friend. ⦚

*Threads*
## A Modula-3 Newsletter

We are pleased to bring you a second issue of *Threads: A Modula-3 Newsletter.* By publishing this newsletter we hope to establish a forum of discussion and information sharing about Modula-3 and what various organizations--industrial or academic--are doing with Modula-3. We tried to make the articles accessible to both currently active and potential Modula-3 users. We hope to invite those who now use other programming languages give Modula-3 a try, too.

We welcome your ideas and contribution in shaping the future of Threads. We imagine that Threads will change with your input over the next few issues. Please send your comments to threads@cmass.com. You can also view *Threads*, on-line at http://www.cmass.com/threads. ⦚

### *What is Modula-3?*

Modula-3 is a simple and modular programming language, providing facilities for exception handling, concurrency, object-oriented programming, automatic garbage collection, and systems programming without involving the complexities forced by other languages of its class. Modula-3 is both a practical implementation language for large software projects and an excellent teaching language.

A free implementation of Modula-3 is available from Digital Systems Research Center. For more information visit the *Modula-3 Home Page* at http://www.research.digital.com/SRC/modula-3/html/home.html.

*Reactor*, a commercial distributed application development environment based on Modula-3, is available from Critical Mass, Inc. For more information, send e-mail to info@cmass.com. ⦚

### *Editors*

Farshad Nayeri
Critical Mass, Inc.

Allan Heydon
Digital Systems Research Center

Bill Kalsow
Critical Mass, Inc.

Emon Mortazavi
GTE Laboratories, Inc.

*Issue 2*
## Table of Contents

*Foreword*........................................ *1*
**Remembering Geoff Wyant**
Bob Sproull

Geoff Wyant, a dear friend of the Modula-3 community, died last summer in a tragic airplane accident. Bob Sproull writes in memory of Geoff.

*Feature Article*.................................... *2*
**Reactor Goes On-line!**
Farshad Nayeri

Farshad Nayeri introduces *Reactor*, a development environment for building robust distributed applications.

*Continuing Thread* ............................... *5*
**A Reusable Software Double-Buffer**
Allan Heydon and Greg Nelson

In their second article in a series about Juno-2, a constraint-based drawing editor, Allan Heydon and Greg Nelson describe the design and implementation a reusable software double-buffer object in Modula-3.

*How Modula-3 got its spots?*....................... *8*
**Why checked run-time errors are not exceptions?**
Greg Nelson

Ever wonder why the language doesn't map all run-time errors to exceptions?

*Modula-3 in Academia*............................ *8*
**Teaching Computer Science with Modula-3**
Spencer Allain and Farshad Nayeri.

With the debut of the English translation of the Modula-3 introductory textbook, *An Introduction to Programming with Style*, and the new release of *SRC Modula-3*, it may be time for you to consider using Modula-3 for teaching computer science. Find out why!

*Advanced Research Topics*........................ *12*
**Link-Time Optimization for Modula-3**
Mary Fernandez

Mary Fernandez describes her work in implementing high-level programming languages with late binding and shows how Modula-3's features that require late binding can be implemented more efficiently with an optimizing linker. The linker also helps with inlining object-oriented code.

*Feature Article*
# Reactor Goes On-line!

Farshad Nayeri, Critical Mass, Inc.

*This article describes Reactor, a development environment for building robust distributed applications from Critical Mass, Inc.*

*Reactor combines Modula-3's support for developing robust, distributed applications and the web's support for displaying complex, inter-related information to construct a powerful but easy-to-use development environment. This article introduces Reactor, outlines its strengths, and motivates some of the decisions in its development. To learn more, visit the Reactor web page at:*

http://www.cmass.com/ /reactor/

*Farshad Nayeri is the Director of Product Development at Critical Mass, Inc.*

After announcing our plans to produce a programming environment for robust distributed applications last year--back then named *Photon*--we at Critical Mass started its development. Aiming at a quick, incremental upgrade to the existing state-of-the-art Modula-3 system, we planned on first releasing the system for Linux, working our way to supporting other Unix platforms, and eventually moving on to Windows.

A lot has changed since then. Those who have tried the preview release noticed that it supported Windows 95 and NT. Indeed, our current release supports cross-platform development on Unix and Windows systems through a unified programming environment for building distributed applications. With the newly-added incremental garbage collector, cross-platform network objects, native threads and DLL support, Open Database Connectivity (ODBC) interface, and a stable port of the Trestle window system, Reactor's support for Win32 is finally on par with Unix!

## Today's Integrated Development Environments

The trend toward Integrated Development Environments (IDEs), popularized by the advent of GUI-based operating systems such as Windows, has reshaped the programming landscape significantly. Using today's popular IDEs involves traversing a complex (and not always intuitive) maze of windows, menus, and tool bar hierarchies. Developing code in these environments is so complicated that vendors feel compelled to throw in "wizards", "hint screens", and "expert" help systems to guide you through tasks. (If you think using the "integrated" environment is hard, try the command-line interface for their compiler!) Many of these systems lock you into proprietary project formats and user interfaces, making integration with your favorite editor, system utilities, and productivity tools difficult. Finally, to strengthen C++'s weak support for building programs, you may have to invest in additional tools such as separate memory managers, separate bounds checkers, and separate builders.

Unix systems vendors typically take an "open systems" approach: they expect you to purchase each component from a different vendor, ultimately resulting in little or no integration between components. (If the code generated by your CORBA stub generator crashes the new version of your C++ compiler, whose support line do you call?) The net result: you have to work around not only the design and implementation flaws of each product, but also integration problems between

products. The truth is that, despite the sharp rise in complexity of target applications, the general nature of Unix system development at its core has changed little in the past two decades. Due to market and political pressures and the high costs of writing portable, reliable code in C++, many Unix systems vendors today tailor their environments to only a few platforms. If you need to build multi-platform, robust applications, you must base your code on a patchwork of tools and language subsets, or be prepared to roll your own.

Worse yet, if you are targeting both Unix and Windows, you are constrained by the distinct development philosophies, development user interfaces, and configurations. It's no wonder that only a few organizations ship robust, cross-platform programs.

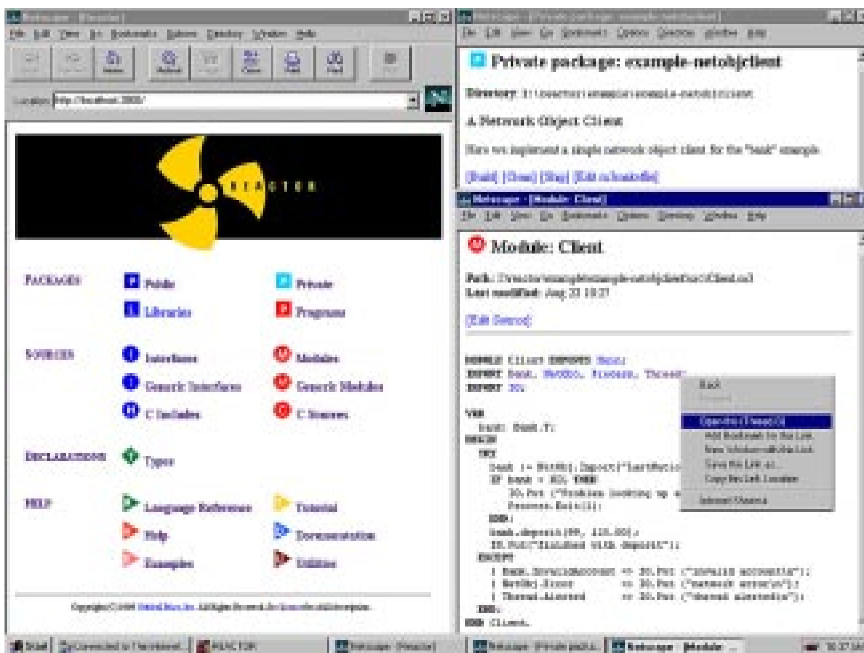



***Figure 1:*** *Reactor User Interface*

## A New Approach

We would like to change all that! In producing Reactor, we aimed to produce a no-nonsense, unified development environment for serious developers whose first and foremost goal is to build robust applications. We realized that to do this, we must:

- *Make the programmers more productive.* This requires raising the lower-bound on the environment and language support for building robust applications significantly. We also have to make it easy for new developers to make the switch to this higher level of productivity.
- *Eliminate unnecessary details and differences.* Unify developer's environment to the degree that a Unix developer feels comfortable developing or shipping with Reactor on Windows. Cross-platform compatibility is a must; porting serious distributed applications from one operating system to another must be easy.
- *Utilize existing technologies.* Avoid locking the user into proprietary, risky, or non-standard technologies as much as possible. Take advantage of investments in existing, relevant infrastructure and standards when they solve users' problems.
- *Practice what we preach.* Utilize the tools ourselves. If we are right, our costs for developing, enhancing, and supporting Reactor on multiple platforms will not be prohibitive. Hence we can pass on some of the cost-savings to the customer.

We noticed quickly that our goals were conflicting. For example, it is hard to use standard technologies if what we are trying to do is to raise the programming standards. In practice, we have had to strike a delicate balance: while we could not always count on standard technologies, we tried to adhere to standards that matter to users. Of course, there are many standards endorsed by various organizations, and there are even more proprietary solutions.

To achieve all these goals, we have had to tread carefully to adhere to standards that are relevant to users and count on non-standard technologies when their advantages significantly outweighed their disadvantages. Using web technology as the user interface for our system allows us to capitalize on an already relevant and growing standard. While not a standard, Modula-3's support for building robust distributed programs continues to be unmatched by existing "standard" technologies such as C++ and CORBA. We believe that the combination of the web-based interface, the clean design of the Modula-3 system, and the extensive portable libraries results in an unprecedented level of support for the programming activities required for building robust distributed applications.

## How Reactor Works

Reactor's IDE is a customized HTTP server that maintains a personal database of your programs and their relationships—all continuously updated. You interact with Reactor using your favorite web browser. Coupled with a new integrated builder, Reactor allows you to browse, build, run, and share programs using the same unified interface on all platforms. While the look of Reactor changes slightly from one web browser or platform to another, its feel and function stay the same no matter where you are. Thanks to Modula-3's support for development of multi-threaded, high-performance servers with long-running activities, the basic system design for Reactor is quite straightforward. At the core of Reactor is a custom server providing access to a virtual namespace of program elements. Each program element has an associated URL. For example, to visit the thread interface, you browse to the

path “/interface/Thread” on a Reactor server. Reactor returns a dynamically marked-up view of the interface complete with syntax highlighting and links to other relevant information. Each program element (for example, a module) has a link to other elements (for example, the types it defines or interfaces it imports). Pre-defined elements provide views to all interfaces, packages, programs, libraries, package collections, types, project descriptions, past compilation results, and documentation. An optional action parameter can be appended at the end of each path; the default action is [view] which displays a node. Other actions, such as [build], [edit], and [run] activate the actions of building packages, editing source files, and running programs. **Figure 2** shows the architecture of Reactor's IDE.

Certainly, the web-based organization makes it easy to traverse large amounts of information about your programs. For some tasks, however, using the web as a medium has not proven easy. For example, displaying dynamic output of a compilation is more difficult since we have less control over the user interface than the traditional IDE. Nonetheless, by taking advantage of threads, we've made it possible for you to leave a build session and re-attach to it later--the compilation proceeds in the background, and the new results are displayed when you revisit the compilation page.

More importantly, the integration with the web means the Reactor user interface works on all platforms, and allows you to share information about your programs with your co-workers easily by sending URLs. Remote programming is also a lot easier. Some users have gone as far as using Reactor to build, browse, and run programs on a Windows/NT system from a browser running on a Unix host.

## A New Compiler, A New Run-time System, and New Libraries

We were on a roll improving the system, so we didn't stop at just a new IDE; we've also made major renovations to the architecture of the compiler, the run-time system, and the libraries.

The new system integrates the builder and the compiler into one process. A single executable named cm3–Critical Mass Modula-3–does all the building and front end work, calling the back end to generate native code. A single configuration file defines the native compiler settings and descriptions of external commands; it is evaluated each time the compiler is invoked so reconfiguring the compiler is a simple matter of changing the file. The compiler keeps track of program elements, deducing dependencies automatically whether or not you use makefiles. **Figure 3** illustrates the new compiler architecture.

The new runtime can produce and link against shared libraries on all supported platforms, including DLLs on Windows. The new incremental garbage collector has already proven to be essential in building response-critical Windows applications for some users. Among other new features, the current release of Reactor includes a safe interface for accessing relational databases via ODBC, cross-platform pickles, and a web toolkit for constructing custom web servers. Last but not least, support for robust distributed programming through the use of cross-platform Network and Stable Objects is one among several hundred facilities included with Reactor.

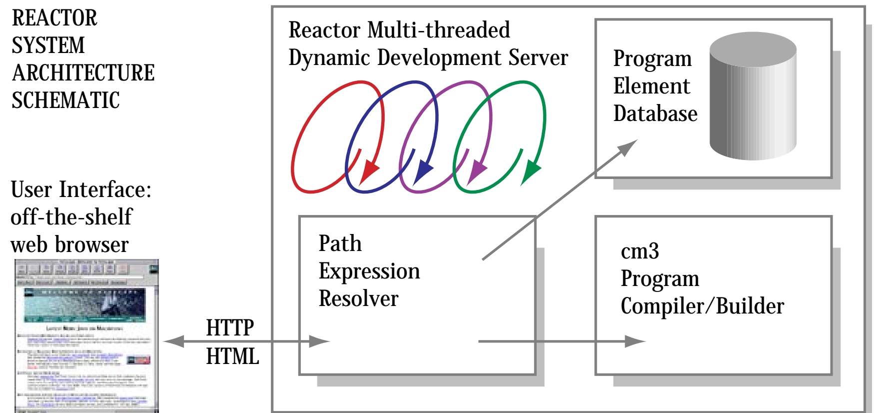



***Figure 2:*** *Reactor User Interface Architecture*

## Future Directions

Our next step is to enhance Reactor's integration with operating systems components such as Microsoft's Active-X (a.k.a. OCX), and to expand the set of available portable libraries for distributed programming.

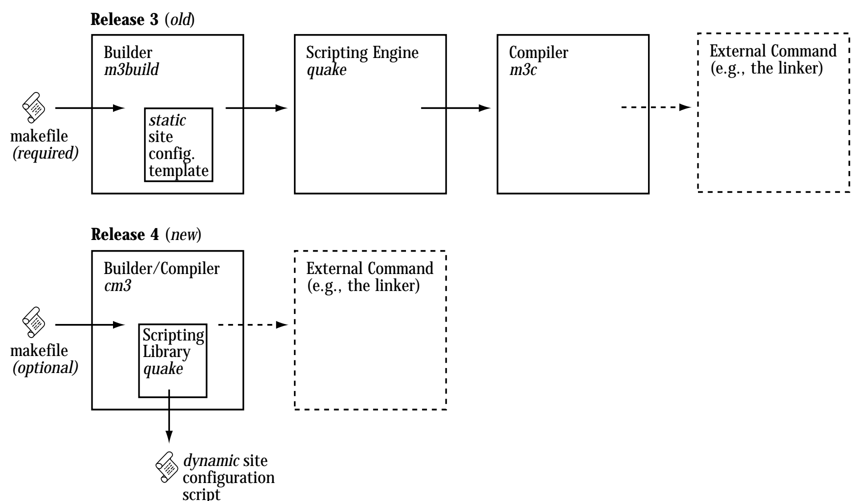



***Figure 3:*** *New Compiler Architecture*

*Continuing Thread*

## A Reusable Software Double-Buffer

Allan Heydon and Greg Nelson
Digital System Research Center

*This is the second in a series of articles about Juno-2, a constraint-based drawing editor. The previous article described the implementation of the Juno-2 user interface. This article describes the design and implementation of the type DblBufferVBT.T, a reusable software double-buffer object.*

*Allan Heydon and Greg Nelson are members of research staff at Digital Systems Research Center.*

Double-buffering is a well-known graphics technique for delaying the effect of graphics operations. It is usually implemented by rendering graphics operations into an off-screen buffer. The contents of the buffer are then copied to the screen in one atomic operation. Ideally, the copy is faster than the time required for the monitor to refresh the screen, so no visual artifacts are visible.

Since there is a performance penalty associated with copying, double-buffering is sometimes implemented in hardware. This article describes a software implementation based on the Trestle window system. In Trestle, a window is an object called a virtual bitmap terminal (VBT) whose behavior is determined by its methods.

In Juno-2, we exploit the effect provided by double-buffering in two ways. First, rendering a drawing can take enough time that the action is perceptible. If the graphics are drawn directly on the screen, the user perceives the rendering as a sequence of graphics operations, rather than as a single update. With a double-buffer, the intermediate painting operations are delayed so that the user sees only the final image. Second, double-buffers are essential for producing smooth animations. An animation is rendered by repeatedly painting the entire window in a background color, such as white, and then painting the next frame of the animation. Without double-buffering, this produces a distracting flickering effect.

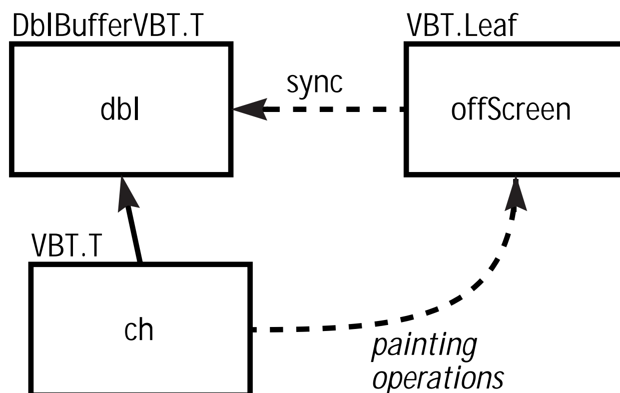



***Figure 1:*** *A DblBufferVBT.T dbl with child ch. The double-buffer creates an off-screen VBT offScreen into which it directs its child's painting operations. The sync method flushes the accumulated painting operations to the on-screen parent.*

The DblBufferVBT interface also provides an extension of standard double-buffering; namely, operations for saving and restoring the double-buffer's contents. This is convenient for producing animations with permanent paint, that is, paint that should be included in all subsequent frames.

### Design

Here is the start of the actual DblBufferVBT interface.

```
INTERFACE DblBufferVBT;
```

A DblBufferVBT.T is a filter that redirects the painting operations of its child to an off-screen buffer, and then updates its screen from the buffer when the child's sync method is invoked. This can be accomplished by calling the VBT.Sync procedure with the child or any of the child's descendants as arguments.

```
IMPORT VBT, Filter;
TYPE T <: Filter.T;
```

The call NEW(DblBufferVBT.T).init(ch) returns a newly initialized double-buffer VBT with child ch.

The child coordinate system of a double-buffer VBT is a translation of its parent's coordinate system. You can compute the translation vector between the parent and child by subtracting the northwest corners of their domains.

A double-buffer VBT v does not forward repaint events to its child; instead, it repaints by copying from the off-screen buffer.

**Figure 1** illustrates the VBTs involved in double-buffering, and the way painting operations flow through the off-screen buffer. The solid arrow represents that ch is the child VBT of the double-buffer dbl.

Next, we describe an extension provided by DblBufferVBT for saving and restoring a double-buffer's contents.

It is common for all of the frames in one scene of an animation to share a common background. Although the client could paint the background afresh on each frame, it would be more efficient and convenient to take a snapshot of the background and restore it at the start of each frame. The rest of the DblBufferVBT interface provides just such a facility.

In addition to its off-screen buffer, a DblBufferVBT.T maintains a saved buffer and provides operations for copying the off-screen buffer to and from the saved buffer. This is convenient for building up a background to be restored on each frame of an animation, for example. The initial content of the saved buffer is a conceptually infinite pixmap of background pixels.

Here are the procedures for saving, restoring, and

clearing the saved buffer:

```
PROCEDURE Save(v: VBT.T);
```

Requires that some proper ancestor of v be a T. Sets the saved buffer of the first such ancestor to be a copy of its off-screen buffer.

```
PROCEDURE Restore(v: VBT.T);
```

Requires that some proper ancestor of v be a T. Sets the off-screen buffer of the first such ancestor to be a copy of its saved buffer.

Save(v) and Restore(v) force all painting operations (paint batches, in Trestle terminology) from v to the relevant off-screen buffer. This will work smoothly if v is the only leaf descendant of the relevant double buffer (i.e., if all splits between them are filters). Otherwise, you may get the wrong answer due to unforced paint batches on other leaf descendants.

```
PROCEDURE ClearSaved(v: VBT.T);
```

Requires that some proper ancestor of v be a T. Clears the saved buffer of the first such ancestor to contain an infinite pixmap of background pixels.

```
END DblBufferVBT.
```

Figure 2 illustrates how the Save and Restore operations copy between the off-screen buffer and the saved buffer.

## Implementation

For efficiency, a DblBufferVBT.T maintains two rectangles named screenDiff and savedDiff.

The screenDiff rectangle is a bounding box of all pixels that differ between the on-screen window and the off-screen buffer. When painting operations are forwarded from the child to the off-screen buffer, a conservative bounding box for the operations is computed, and the screenDiff rectangle is augmented to include the pixels affected by those operations. The sync method copies only the pixels contained in screenDiff, and then sets that rectangle to Rect.Empty. This technique reduces the copying cost by copying a smaller area than the entire window. Of course, if pixels have changed at opposite corners of the window, then almost the entire window will be copied. In our experience, however, the overhead of computing more accurate bounding regions is justified by the savings in copying costs.

The savedDiff rectangle is a bounding box of all pixels that differ between the saved buffer and the off-screen buffer. This rectangle is also augmented to include pixels changed by new painting operations. Both the Save and Restore procedures copy only the pixels contained in the savedDiff rectangle, and then set the rectangle to Rect.Empty.

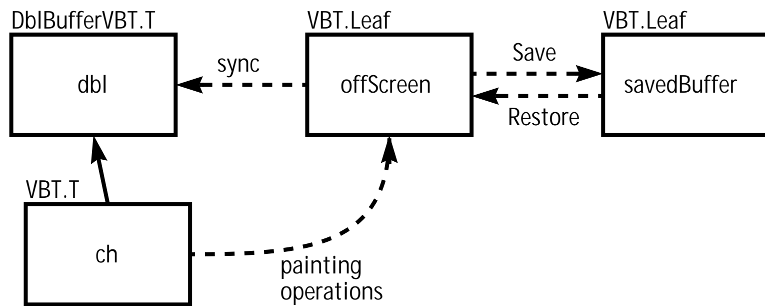



***Figure 2:*** *The Save and Restore operations copy the off-screen buffer to and from the saved buffer.*

## Example

**Figure 3** shows several snapshots of an animation demonstrating a discovery of Rida Farouki: If you walk along the graph of the curve $y = x^4$ carrying a beam that extends one unit in each direction, the inner tip of the beam traces out a star.

This example demonstrates the use of the double-buffer's saved buffer. Each frame can be divided into three parts: the background common to all frames (the black curve and the text), the permanent paint that should be included in all subsequent frames (the path drawing a star), and the ephemeral paint that should only be included in this frame (the beam). Here is the code for the animation:

```
PaintBackground(ch);
VAR start := Time.Now(); t, tLast := 0; BEGIN
  WHILE t < Duration DO
    PaintPermanentPath(ch, t, tLast);
    DblBufferVBT.Save(ch);
    PaintEphemeralBeam(ch, t);
    VBT.Sync(ch);
    DblBufferVBT.Restore(ch);
    tLast := t;
    t := Time.Now() - start
  END;
  PaintPermanentPath(ch, Duration);
  PaintEphemeralBeam(ch, Duration);
  VBT.Sync(ch)
END
```

First, we paint the background common to all frames. Then, we render each frame. After painting the permanent paint, we Save the contents of the double-buffer so that the permanent paint will become part of the background on subsequent frames. After calling Sync to copy the off-screen buffer to the screen, we Restore the background for the next frame. The last frame painted from within the WHILE loop is for a value of t less

than the animation's full duration. So, upon completion of the loop, we paint the animation's final frame.

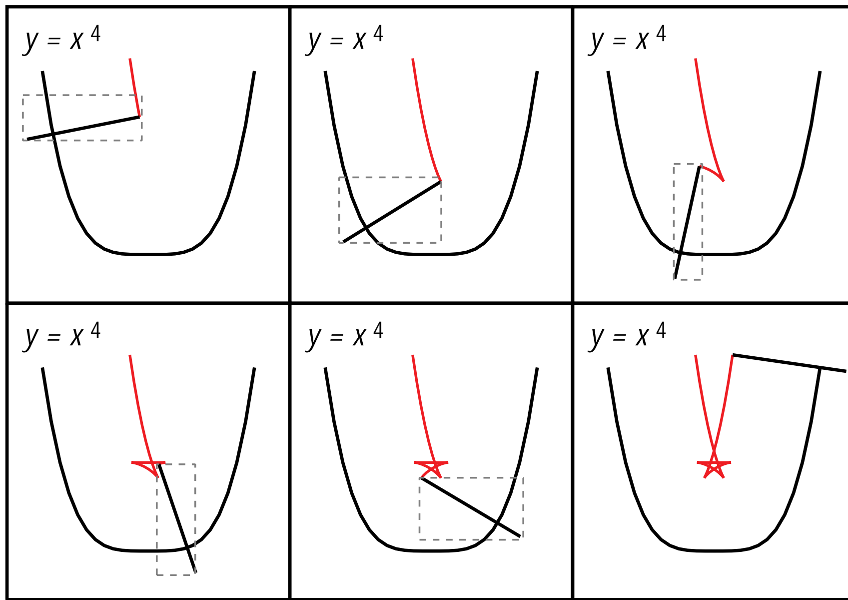



***Figure 3:*** *Several animation snapshots. The dashed rectangle shown in each snapshot is the screenDiff rectangle copied by the sync operation.*

You can see from the size of the dashed screenDiff rectangle that the double-buffer implementation excels at minimizing the number of pixels copied on each frame. In general, the rectangle includes only the pixels for the beam from the previous frame (since those had to be erased), the new piece of the star-drawing path, and the new beam.

## Performance

**Figure 4** shows the performance of the double-buffer while animating a simple filled triangle. These measurements were made on a Digital 3000/600 Alpha workstation equipped with a 175 MHz DECchip 21064 processor, and an 8-bit frame buffer, and running Digital Unix (OSF/1). Each point (*x*,*y*) corresponds to a single frame: *x* is the number of pixels painted for the frame (i.e., the area of the screenDiff rectangle), and *y* is the elapsed time in milliseconds (ms) between that frame and the next. The graph shows that software double-buffering has a fixed cost of about 3 ms per frame, and a marginal cost of 1 ms per 40K pixels per frame.

The data for **Figure 4** were collected while animating a very simple drawing, in which the double-buffer copying costs (0 – 9 ms) dominated the graphics costs (fractions of a ms). The copying overhead is less noticeable in a more typical drawing, where the graphics can require tens or even hundreds of milliseconds. Even so, the relatively steep slope of the line in the figure indicates that the overhead of computing the screenDiff and savedDiff rectangles is worthwhile.

## Conclusions

The hallmark of the double-buffer design is its simplicity. To double-buffer some window, a client simply has to wrap a DblBufferVBT.T around the VBT.T corresponding to the window, and then make calls to VBT.Sync at appropriate times to flush the double-buffer. Hence, DblBufferVBT.T is a reusable class that can be applied on a per-window basis. It is a good example of the extensibility made possible by the object-oriented Trestle design.

The facility provided by the double-buffer's saved buffer is also quite useful. We employ it in Juno-2 for two purposes: for animations with permanent paint and for implementing tools with a text argument. In the latter case, we first save the current drawing into the saved buffer. Then, for each character the user types, we restore the off-screen buffer from the saved buffer, apply the tool's procedure with the text typed up to that point, and then sync the off-screen buffer. Previously, we had to redraw the entire figure on each character; for complicated figures, this could result in an annoying delay between keystrokes.

So far, DblBufferVBTs are used only in Juno and in one other Trestle application: a shared whiteboard. However, they could easily be retrofitted into such applications as the Zeus algorithm animation system and the GraphVBT implementation. It would probably simplify the code in both cases, since those systems are already performing their own double-buffering.

## Acknowledgment

Steve Glassman helped us with the original implementation of the DblBufferVBT interface. ⦚

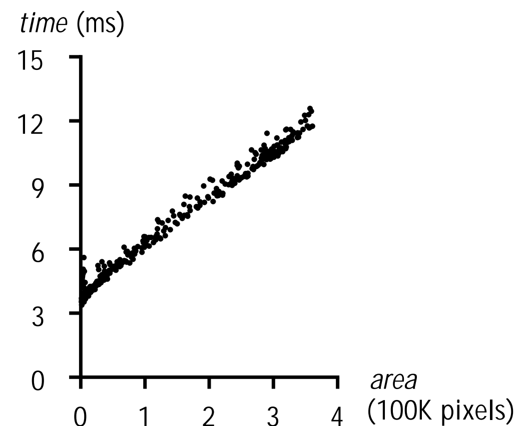



***Figure 4:*** *The per-frame cost of software double-buffer as a function of the number of pixels painted per frame.*

## *How Modula-3 got its spots?*
## Why checked run-time errors are not exceptions?

Greg Nelson, Digital System Research Center

Modula-3 defines a *checked run-time error* as an error that implementations must detect and report at run-time. For example, using an array index that is out of bounds is a checked run-time error.

The method for reporting checked run-time errors is *implementation-dependent*. For example, in a program development environment, the most useful implementation action would probably be to enter a debugger. However, in an operating telephone switch, appropriate actions would more likely include logging the error and restarting the switch software.

Proponents of mapping checked run-time errors to exceptions argue that the language should be changed to require implementations to raise predefined exceptions on checked run-time errors. This would give programmers maximum flexibility to recover from the error in whatever manner is appropriate for the application. But this argument doesn't stand up to scrutiny.

First, when a checked run-time error is detected, the appropriate recovery action almost always requires implementation-dependent actions. Changing the language to provide an implementation-independent way to detect such errors only postpones the problem. For example, there is no implementation-independent way to enter a debugger, to log an error, or to restart a server.

Second, it is important to realize that checked run-time errors can occur in threads that are forked by libraries, as well as in an application's main code path. If errors were mapped to exceptions there would be a need for "TRY" statements to handle these exceptions wherever a thread forked. When a program moved from testing into actual service, and the appropriate error recovery action changed, it would become necessary to modify many "TRY" handlers scattered throughout the program.

A better strategy is to let the implementation determine the error recovery action. ⦚

*← Greg Nelson is a Member of Research Staff at Digital Systems Research Center.*

*Aside from his key role as the editor of Systems Programming with Modula-3, he has contributed to the development of several Modula-3 libraries and systems.*

## *Modula-3 in Academia*
## Teaching Computer Science with Modula-3

Spencer Allain, Raytheon E-Systems
Farshad Nayeri, Critical Mass, Inc.

*With the debut of the English translation of a Modula-3 textbook,* ***An Introduction to Programming with Style****, and the new release of* ***SRC Modula-3****, it's time for you to consider using Modula-3 for teaching.*

*In this article, Spencer Allain and Farshad Nayeri describe some of the reasons why you may choose Modula-3, what people who have used Modula-3 for teaching have to say about it, and where to go to get more information about teaching with Modula-3.*

*Spencer Allain is a software engineer at Raytheon E-Systems, and a graduate student at Geroge Washington University.*

*Farshad Nayeri, an editor of Threads, has been a happy user of SRC Modula-3 for a number of years.*

Many schools are considering newer languages for teaching in various computer science courses. Traditionally the debate has been between C, Pascal, Ada, and Modula-2--with a smattering of Fortran and Scheme to boot.

Academia has slowly begun to adopt some of the new languages; namely those labeled as object-oriented. Today it's accepted that exposing students to many different programming concepts will enhance their understanding, but a college has only a limited amount of time to impress this knowledge upon its students.

In an ideal world, each university would have unlimited funds, enough staff to have an expert in each programming arena, and a body of computer science majors composed of geniuses that would be able to absorb all of the information about every programming concept within the short time-span of an undergraduate degree. The world isn't perfect, and compromises need to be made. The first and foremost tends to be:

What do we wish to use as our core programming language to convey the most information successfully to the majority of the students?

Some universities are fortunate enough to have the funding and the personnel to be able to support two or more core languages, but many simply do not and must make the difficult decision of selecting only one all-purpose language. Even the universities that must choose two languages, still have a difficult decision ahead of them and should not make their decisions lightly, as two poor languages may produce worse results than one good language.

There are a plethora of languages available, and many are quite good for teaching purposes, but what defines an excellent language for learning?

There are many criteria, and they must be tailored to the goals of each individual university. For instance, colleges that emphasize training students well in the languages that appear most often in the want-ads clearly have different objectives than the colleges that focus upon programming theory and utilize the languages that facilitate this learning in lieu of market demands. For universities that are driven by industry demand only, the choice of programming languages is clear--teach the languages listed in the majority of the job offerings: Visual Basic, C or C++. It is the universities with impeccable reputations that have the luxury of being in the second category; students are drawn in by prestige, not

whether industry uses the same languages.

Most universities, however, fall somewhere in between. They attempt to use the appropriate languages, but fall back upon industry standards when decisions boil down to recruiting new students. Finally, they are worried about the impact of the choice of the language in the satisfactory completion of projects and the overall health of the curriculum.

The remainder of this article targets universities who are looking for a fresh alternative to what they teach today, whether it is Pascal, C, or Ada. We encourage you to read on even if you don't fit into this category as we think you'll find many issues of interest, even if you don't end up switching to a new language.

First, some characteristics of a good teaching language:

- It should be clear and straightforward so that the logic of the program is easy to follow from the code.
- It should cover all major issues for which it is being used to convey, in an integrated and coherent fashion. Its design should allow concepts and features to be introduced incrementally, so that novice programmers are not overwhelmed.
- It should facilitate the learning of high-level concepts before requiring the student to deal with all the low-level issues.
- It should be scalable well beyond what will be taught in lectures, as students need room to experiment. There is nothing harder than defending a bad choice by the language designer to a student who is trying to find a better way to do something.
- Experience with the language should be readily applicable to skills in the professional market.
- The implementation of the language must be well-worn, reliable, and well-documented. It must run on modest hardware, and it should be very inexpensive or even free.

There are many possible languages to choose from, but there is one in particular that is strong in all the above areas, yet under-utilized by many schools because they have not heard about it. The language is *Modula-3*.

## What is Modula-3?

Modula-3 is a member of the Pascal/Modula family of languages. Despite its name, Modula-3 is much more than just another successor to Modula-2. The language and its implementation have been stable for the past five years; they have always boasted a nice integration of features that have only recently been realized in other designs, such as C++, Ada95, and Java.

The goal of Modula-3 is to be as simple and safe as it can be while meeting the needs of modern systems programmers. Instead of exploring new untried features, the designers followed proven practice. The language features that depart from previous designs aimed at two important areas: a simpler type system and greater robustness.

Modula-3 retains Modula-2's module system for the most part and most of its Pascal-like syntactic style. It adds objects, exception handling, garbage collection, lightweight processes (also called threads), generics, and the ability to separate safe and unsafe code, all in one integrated whole.

The combination of features in Modula-3 makes it both a wonderful systems programming language and a great teaching language. The central source for finding information about Modula-3 is the Modula-3 Home Page at Digital Systems Research Center:

```
http://www.research.digital.com
    /SRC/modula-3/html/
```

## Why Modula-3 for Teaching?

Modula-3 is well-suited for teaching because it combines simplicity, power, and safety.

Since less time is spent by students and teaching assistants chasing dangling pointers and corrupted data, more time is available for learning the important concepts.

Modula-3 avoids the complexity of legacy languages. The Modula-3 language specification is *fifty* pages long even though Modula-3 is as powerful as C++ and Ada95. However, Modula-3 does not hinder the programmer as Pascal does. The language design is very uniform, and allows the programmer to work at different levels of abstraction easily.

Modula-3 can be used also for serious systems work. For example, the University of Washington has had quite a positive experience in using Modula-3 for building *SPIN*, their extensible operating system, showing that Modula-3 can easily be on or above par with C/C++ for systems programming. In the safe subset, however, Modula-3 works well as a predictable programming language, encouraging the programmer to concentrate on the problem at hand instead of working around language limitations.

In general, as it is a lot easier to concentrate on solving problems by writing good programs in Modula-3, students will have the satisfaction of completing programming projects, which will help motivate them to excel. (See "what do people who use Modula-3 have to say about it?" below) Also, well-designed and well-implemented Modula-3 libraries (more than 2500 modules) serve as great examples for students.

To sum up, by teaching in Modula-3, you can easily demonstrate:

- basic programming, via a Pascal-like syntax
- modules and interfaces

- object-oriented programming
- generics
- multi-threading, concurrency and operating systems issues
- graphical user interfaces
- data structure and algorithm via animation
- distributed and persistent programming concepts

To get a better idea of the academic experience with Modula-3, please see the quotes from an informal survey at the end of this article.

## Addressing Practical Concerns

SRC Modula-3, a freely-available Modula-3 implementation, comes with a large standard library (libm3) providing:

- Text manipulation (automatic garbage collection allows for a real concatenation operator.)
- Generic containers: lists, sequences, tables, sorted lists, sorted tables
- Atoms and symbolic expressions (Lisp-like lists)
- An extensible stream I/O system
- Type-safe binary object transcription (persistent objects)
- Operating system interfaces
- Portable interfaces to the language runtime

All standard libraries are thread-friendly and Modula-3 can readily link with existing C libraries. They come with extensive interface documentation, and equal in quality some of the best commercial tools.

There is also a very large array of stable and well-documented libraries available. Some features are:

- Trestle, a complete multi-threaded user interface layer abstracting differences between Win32 and X
- Network Objects, state-of-the-art distributed object system featuring pure object remote procedure calls and streams
- An extensive algorithm animation system that is used by various universities for teaching data structures and algorithm courses
- Programming and documentation tools
- A custom HTTP server that lets students easily browse the entire Modula-3 library via a web browser

Ports of SRC Modula-3 are available for Windows95/NT, DOS, and just about every flavor of Unix from OSF/1 and AIX to Linux and FreeBSD. There is also a port to OS/2 that is nearing completion. SRC Modula-3 supports fifteen architectures and twenty-five operating systems in all, and the list continues to grow. Most notably, the ports of the language on popular Intel operating systems such as Windows and Linux take advantage of a fast native compiler which compiles Modula-3 files in "Turbo-Pascal" speeds.

When programming in Modula-3, you can easily expect to move your code from one platform to another without a single change in your program sources, giving you much more freedom in setting up your academic computing environment.

And best of all it's *free*!

## What Are Some Universities That Already Use Modula-3?

Here are some universities who use Modula-3 in their curriculum:

- State University of New York at Stony Brook, USA
  CS-I (CSE-114) and CS-II (CSE-214).
- University of Cambridge, UK
  A Modula-3 specific course.
  Modula-3 as well as ML is used in teaching introductory courses.
- Ecole Polytechnique de Montreal, Canada
  Graduate course on Algorithms for CAD
  Undergraduate course on OO programming
- Lehrstuhl fuer Informatik III, Germany
  graduate course in software development
- Royal Institute of Technology, Sweden
  the second course in computer science
- University Klagenfurt, Austria
  SW-1 Programming
  SW-2 Algorithms and Data Structures
- University of Manchester, UK
  Third year undergraduate and masters students
- University of Waterloo, Ontario, Canada
  CS246, Software Abstraction and Specification
  CS241, Foundations of Sequential Programming
  CS340, Data Structures and Algorithms
- University of Massachusetts, USA
  graduate and advanced undergraduate projects
  a target language for a compiler course
- Vassar College, USA
  CS-123, Computer Science II
  CS-235, Programming Languages

## Textbooks and Reference Material

- Laszlo Boeszoermenyi, Carsten Weich, *Prgramming with Modula-3, An introduction to Programming with Style*, Springer Verlag, German edition: ISBN 3-540-57911-7. English translation is now available from Springer Verlag: ISBN 3-540-57912-5.
- Samuel P. Harbison, *Modula-3*, Prentice Hall, ISBN 0-13-596396-6, 1992.
- Joseph Bergin, *Object-Oriented Data Structures in Modula-3* . A draft of a data structures text in Modula-3 for a second course in Computer Science. For more information, contact berginf@pacevm.dac.pace.edu
- Robert Sedgewick, *Algorithms in Modula-3*. Addison-Wesley, ISBN 0-201-53351-0, L.C. QA76.73.M63S43, 1993.
- Greg Nelson (editor), System Programming with Modula-3, Prentice Hall Series in Innovative Technology, ISBN 0-13-590464-1, L.C. QA76.66.S87, 1991.

## What Do People Who Use Modula-3 Have to Say About It?

Recently we performed an informal survey of academic sites who have used and are using Modula-3 for their teaching. Below are some extracts that reflect what the professors had to say about their experiences:

"It is a very good language in many ways, very complete, clear, and fairly straightforward compared with say, C++ or Ada. [...] We used to use Ada, but it was slow (at the time), expensive (at the time), and more complex syntactically. [...] The advantages of Modula-3 are: the clarity and cleanness of the language, the similarity to Pascal, and the decent reference books. It would take a very persuasive argument to switch away from Modula-3"

"Last year we had a programming project course where we recommended the students to use Modula-3, but some project groups chose to use C++ or Borland Pascal. Most of the groups using Modula-3 did complete the course in time, while many of the others did not:

| | |
|---|---|
| Modula-3: | 85% on-time (22 of 26) |
| Borland Pascal: | 67% on-time (2 of 3) |
| C++: | 00% on-time (0 of 4) |

We found the top advantages of Modula-3 to be: its similarity to but without the shortcomings of Pascal, its support for modularization, garbage collection and the standard library."

"Very good. The students adapt to the language very quickly. The top advantages of Modula-3 are that it is a sound and safe language, and can be used for the whole curriculum."

"We have selected Modula-3 in a long process in a group of about ten people consisting of two professors and six to eight assistants. We had a catalogue of criteria and we have considered, in the last selection phase, the following languages: Modula-3, Oberon-2, Eiffel, Ada, C++, Turbo Pascal. We made two different kind of evaluations, Modula-3 won both. Top three advantages were:

- unusually clean definition of structured constructs
- object-orientation and nice concept of subtyping
- threads."

"It's certainly the best of the imperative languages that I have used. It's more general and more uniform than Pascal, and vastly more so than C. It's easy to give clean and consistent explanations of what's happening. The top advantages are: clean and consistent syntax and semantics, excellent support for modules and abstract data types, and type safety."

"Top three advantages of Modula-3 are:

- Coverage: exceptions, objects, concurrency, large-scale programming are all there in an integrated, coherent form. Having learned Modula-3 the students should be able to learn and use almost any other imperative language in a disciplined way
- Clarity: the language encourages good style
- Completeness: the libraries."

"As far as project courses are concerned, I would say that Modula-3 is more or less perfect. The main reasons speaking for Modula-3 are: - The language is very easy to learn. Whatever language(s) the participants have learned before, they get acquainted very quick. Even more advanced features like genericity, exception handling, or concurrency are easy to introduce. Modula-3's compactness allows learners to focus on the concepts instead of language features. - Modula-3 is a modern language. It proved to be a sound basis for the discussion of general programming concepts. - The language supports modularization instead of data types/classes as its main decomposition/ structuring scheme. With respect to teaching design and cooperative work (which is what the oject course is all about), this is probably Modula-3's most valuable feature (compared to C++ or Eiffel). - Last but not least, SRC's compiler & libraries are exceptionally stable and powerful. Whatever the concrete topic of the course is, the Modula-3 system provides very good support. [...]"

"We will not consider switching away from Modula-3 in the near future. For the project as well as for the introductory course, the principle is always concepts first, language second. For the time being, I am convinced that no other language provides a comparable ratio of features and conceptual clarity."

"We used to work with Modula-2 and C++ in the project course before we switched to Modula-3 two years ago. Since then, we started to get real, usable results (much to the satisfaction of the participants)."

"[Our experience with Modula-3 has been] generally positive; covers all aspects of imperative programming that we wish to cover; many students, however, might vote for C++, though this is not a big issue."

"[Students] want to learn the latest buzz language (eg, C++). Students are required to have some programming experience before taking CS-I. Many had Pascal or C. Since Modula-3 is somewhat different (syntax and semantics), they often complained it was difficult to learn. Learning a new language is an important experience for software engineers, so from our [CS department's] perspective this was good."

"The top-most advantage of the system is its clean structure of the language, Modules and Objects together, and the SRC compiler is good and free."-

*Much information is available about Modula-3 on the web. Here we include some of the top-level references:*

*Modula-3 Home Page*
http://www.research.digital.com
/SRC/modula-3/html/home.html

*Modula-3 FAQ*
http://www.vlsi.polymtl.ca/m3/

*Threads:*
*A Modula-3 Newsletter*
href=http://www.cmass.com
/threads/

*Standard Library Spec.*
ftp://gatekeeper.dec.com
/pub/DEC/SRC
/research-reports/SRC-113.ps.Z

*A Modula-3 Bibliography*
http://www.research.digital.com
/SRC/modula-3/html
/concise-bib.html

*Modula-3 newsgroup*
comp.lang.modula3
m3@src.dec.com

*Spencer Allain's*
*Modula-3 Interest Page*
http://mason.gmu.edu
/~sallain/html/modula3.html

*Advanced Research Topics*

## Link-Time Optimization for Modula-3

Mary Fernandez, AT&T Research

Modula-3's modules and interfaces, object types, and type inheritance provide strong support for development of modular and reusable software libraries. Opaque object types, the powerful result of combining these features, guarantee that a client module can be compiled even when the implementation of an imported object type is unavailable. This is often the case when object types are implemented in libraries. For example, the module Symbol can be compiled in the absence of Hash's implementation, even though SymbolTab is derived from HashTab, a type which is implemented in Hash. Opaque types also support upwardly compatible object libraries, (for example, clients of a library do not have to be recompiled when a new implementation of the library is released).

```
INTERFACE Hash;
TYPE HashT = OBJECT
  METHODS
    lookup(key: TEXT): REFANY;
    insert(key: TEXT; value: REFANY);
    delete(key: TEXT)
  END;
  HashTab : HashT
END Hash.
```

```
MODULE Symbol;
FROM Hash IMPORT HashTab;
TYPE SymbolTab = HashTab OBJECT
  level: INTEGER
  OVERRIDES
    insert := Insert
  METHODS
    enterscope();
    exitscope();
  END;
END Symbol.
```

Opaque object types clearly distinguish Modula-3 from C++. C++ reveals objects' implementations in their interfaces, which helps a C++ compiler implement references to objects efficiently. However, this prohibits development of upwardly compatible libraries. C++ users often simulate opaque types with cumbersome programming conventions that are not type-safe nor enforceable by a compiler.

Opaque object types incur runtime costs, however, because they prevent the compiler from having complete information about an object's implementation, such as the types and sizes of its data and the procedure bindings of its methods. Without access to this information, the compiler must implement late binding, that is, generate code that computes the missing information at run time.

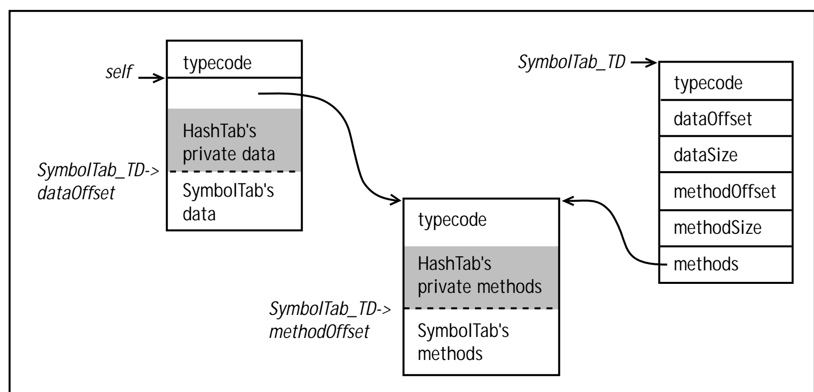
My thesis describes a software approach to implementing high-level programming languages with late binding and shows how Modula-3's features that require late binding can be implemented more efficiently with an optimizing linker. Link-time optimization eliminates the costs of opaque types and reduces the costs of method invocations by finalizing objects' representations at link time. Link-time optimization also permits inlining of methods without compromising program modularity.

*Mary Fernandez is a Senior Member of Technical Staff at AT&T Research.*

### Opportunities for Link-Time Optimization

(Although this article describes opportunities for link-time optimization using the SRC Modula-3 implementation as an example, similar opportunities would exist for other implementations of Modula-3 or for other languages that support late binding.)

Because the representation of opaque types is incomplete at compile time, the runtime system provides a representation of types that describes their complete implementation. At runtime, a type is represented by a type descriptor, which contains the sizes and offsets of the data and methods associated with instances of the type. For example, SymbolTab_TD denotes SymbolTab's type descriptor. The SymbolTab object self is represented by its own data area, and by a pointer to its type descriptor's methods, which are immutable after type initialization. These offsets and method bindings are computed at program startup by the Modula-3 runtime system and are stored in the type descriptors.




The implementations of opaque typing and method invocations incur direct and indirect costs. Direct costs include:

- an extra "fetch-and-add" to compute the address of a field or method and

- an indirect procedure call to invoke a method.

For example, when compiling Symbol, the compiler knows nothing about the structure of HashTab's private fields and methods (shaded) and therefore, cannot compute the offsets to self's SymbolTab fields and methods, i.e., the values of SymbolTab_TD's dataOffset and methodOffset fields. So, to compute the address of self.level, the compiler generates the following C-like code: address_of(self) + SymbolTab_TD –> dataOffset + offset_to(level).

A method invocation includes a similar address computation followed by an indirect procedure call. The compiler implements method invocations as indirect calls to support strong encapsulation and overriding. For example, the invocation self.lookup(key) in Symbol is compiled into an indirect call, because the compiler cannot access lookup's procedure binding in HashTab's private methods. In addition, it cannot determine if lookup will be overridden in a subtype of SymbolTab, and therefore have more than one procedure binding at run time. Implementing method invocations as indirect calls incurs a (necessary) indirect cost, because it prevents inlining and specialization of methods at call sites.

Link time is the earliest time at which a program's entire type hierarchy is known and therefore, the earliest time at which the attributes of type descriptors can be computed. Given a smart linker that can compute and use this information, all expressions involving type descriptors can be simplified at link time. For example, the expression SymbolTab_TD->dataOffset + offset_to(level) is reducible to a constant at link time. A smart linker can also identify those methods bound to a single procedure and can convert their invocations to direct procedure calls.

It is important to note that the code generated to implement these features produces idiomatic expressions in the intermediate code. Although the contents of the idioms (e.g., the values of constants) may vary across targets, the idioms themselves are target-independent and are easily identified given simple information, such as the types of variables, in the intermediate code.

## *mld*: A Retargetable, Optimizing Linker

The system I built to evaluate link-time optimization includes: *mill*, a machine-independent linker format suitable for link-time optimization and code generation; *mlcc*, a C-to-mill compiler; and *mld*, a *mill* linker. mlcc and mld are based on lcc, a retargetable ANSI C compiler and were built by dividing lcc at the interface between its front and back ends. mlcc includes lcc's front end and a new code generator that emits mill code instead of target-dependent object code.

The mill code for a module is a compact binary code that contains call graphs, flow graphs, symbol and type information, and trees of lcc's intermediate instructions that are the executable code. Linking lcc's intermediate code instead of machine-object code simplifies optimization. Recognizing and simplifying idiomatic expressions in object code is difficult, especially on architectures where instructions are reordered by instruction schedulers.

mld performs the functions of a traditional linker, but it processes mill code instead of object code. Unlike traditional linkers, mld applies optimizations before it generates code for a complete executable program. Delayed code generation increases link time, but mld compenstates by using variants of lcc's fast code generators that emit binary instructions.

Optimizers often use program representations that preserve the source language's semantics. The whole program optimizier, for example, transforms an annotated abstract-syntax tree. We chose a low-level representation because it allowed us to determine whether useful link-time optimizations can be applied to a low-level code (they can) and to measure the costs of delaying code generation until link time (they're tolerable). For example, mill files are only 2.5 times larger than unstripped object files generated from the same source, whereas persistent AST representations are usually more than 5 times the source size. Delayed code generation means link time is proportional to the number of instructions generated, but mld's fast code generators help. For example, mld links and emits a large executable with a 1 MB text segment in about fifty seconds on a DEC 5000/240.

To apply link-time optimizations to Modula-3 programs, we use the v2.11 Modula-3 compiler, m3. m3 invokes mlcc, which produces mill files for application modules and for the complete Modula-3 runtime system. At link time, m3 invokes mld.

## *mld*'s Optimizations

*mld* implements late binding using data-driven simplification, which simplifies expressions that refer to variables whose values are constant after linking. mld obtains the bindings between variables and their link-time values from a binding file, which contains assignments of values to global symbols.

For a Modula-3 program, mld executes an initialization procedure similar to the one executed by the Modula-3 runtime system at program startup, but instead of initializing the type hierarchy in memory, it creates a binding file that describes the initialized type hierarchy. For example, part of the link-time binding data for SymbolTab_TD includes:

```
*SymbolTab_TD = {
  typecode = 6;
```

```
dataOffset = 408;
methodOffset= 24;
parent = HashTab_TD;
*methods = [
   4: Hash__Lookup;
   8: Symbol__Insert;
   12: Hash__Delete;
   16: Symbol__Enter;
   20: Symbol__Exit;
];
```

After creating the binding file, mld traverses mill instruction trees, identifies expressions that refer to link-time constants, and simplifies them. mld's expression matcher and simplifier are generated automatically by lburg from concise rules that map mill idioms into simpler mill expressions. Some rules eliminate the fetch-and-add cost of accessing fields and methods; others convert invocations of singly-bound methods to direct procedure calls. mld also applies cross-module inlining of frequently executed methods and procedures.

Binding files are not restricted to type information nor are they dependent on Modula-3. Any write-once data is permissible, such as an array after initialization. mld can use binding information to optimize any mill program, and its expression simplifier has few dependencies on Modula-3.

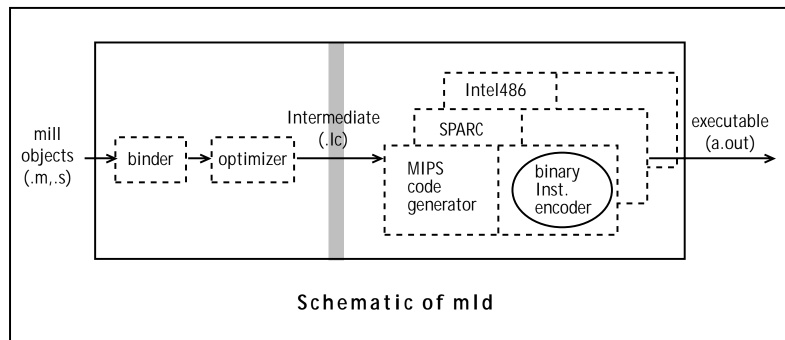
## Results

mld's optimizations are intended for programs that use objects heavily. When applied to six large Modula-3 programs, data-driven simplification reduces total instructions executed by 3-11% and total loads executed by 4-25%. It converts an average 28% of dynamic method invocations to direct calls. These changes result in elapsed-time improvements up to 25%. As expected, programs that use objects most, benefit most. None of our benchmarks use objects heavily, however, so we would expect greater improvements for programs written primarily in an object-oriented style. Link-time code generation dominates mld's execution time, but the optimizations themselves are inexpensive: they increase link time by less than 10%.

## Conclusions

mld's optimizations do not require complex algorithms, and they are inexpensive to apply. We focus on simple techniques that use whole-program information, which is unavailable at compile time. Despite their simplicity, the techniques are effective, even for programs that do not use objects aggressively.

It is possible to apply similar optimizations at compile time, but at the expense of less modular programs and more complex program maintenance. For example, an "optimistic" Modula-3 compiler could avoid the runtime overhead of opaque types by fixing their representations using "hints" about the sizes and structure of imported types. Any changes to an imported type's representation would require recompilation of modules that import the types. Link-time optimization is simpler, because it has complete information, and is more flexible, because it permits optimization of modules in libraries, for which the source is often unavailable. I have described those link-time optimizations that only require the complete type hierarchy and that we have implemented and measured. Other interesting link-time optimizations, such as conversion of heap-allocated to stack-allocated objects and the safe elimination of runtime range and nil-object checks, require intra- and inter-procedural data-flow analysis as well as the complete type hierarchy. My thesis describes these link-time optimizations in detail and discusses their potential effectiveness for Modula-3.



Schematic of mld

*Threads: A Modula-3 Newsletter. Issue 2.*

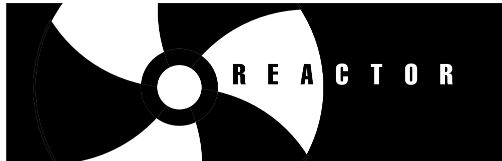
Copyright © 1996 Critical Mass, Inc.
All Rights Reserved.

*Threads* is available via the world-wide web at http://www.cmass.com/threads.

For more information contact:
Critical Mass, Inc.,
1770 Mass. Ave., Cambridge, MA 02140
*telephone* +1 617 354 6277
*e-mail* threads@cmass.com
*web* www.cmass.com

# Are your distributed applications bulletproof?

## They can be with




Reactor is a programming environment designed to support the development of robust, enduring, and distributed applications.

At the center of Reactor is a state-of-the-art object-oriented programming language with built-in support for garbage collection, threads and exceptions.

The environment integrates a full-featured web browser and an easy-to-use builder.

Reactor also comes with a large array of royalty-free run-time libraries for distributed objects, interfaces to X11 and TCP, threads and

9 8 7 7 8 9

Win NT
Win 95
Solaris
SunOS
Linux
HP/UX
SGI Irix
Digital Unix


critical MASS
1770 Massachusetts Avenue
Cambridge, MA 02140 USA
www.cmass.com