# THREADS

## Fall 1995 - Issue 1

A Modula-3 Newsletter

## Introducing Modula-3 *Threads*

We are pleased to bring you the first issue of *Threads*, a newsletter for the Modula-3 community. 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 drop us a note at threads@cmass.com!


### *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.

An implementation of Modula-3 is available free of charge from Digital Systems Research Center. For more information visit the Modula-3 home page: http://www.research.digital.-com/SRC/modula-3/html/home.html

Editors:
Farshad Nayeri,
GTE Laboratories

Geoff Wyant,
Sun Microsystems Laboratories

Lauren Schmitt,
Critical Mass, Inc.


*Feature Article*
Implementing Juno-2: The User Interface
by Allan Heydon ..... 2


*How Modula-3 got its spots?*
Initialization of Object Types
by Greg Nelson ..... 5


*Modula-3 in Industry*
Photon: An Environment for Building Distributed Applications
by Lauren Schmitt ..... 6


*Modula-3 in Academia*
University of Klagenfurt, Austria
by Laszlo Boeszoermenyi ..... 7


*Advanced Research Topics*
The Whole Program Optimizer
by Amer Diwan ..... 8

*Feature Article*

# Implementing Juno-2: The User Interface

Allan Heydon, heydon@pa.dec.com
Digital System Research Center

This is the first of a series of articles that describe how Greg Nelson and I used the Modula-3 programming language to implement a constraint-based drawing editor called Juno-2. The articles are meant to show how Modula-3 and the rich collection of libraries included with src Modula-3 have been used in a sophisticated application program.

In this article, I'll describe how we used the Formsvbt, vbtkit, and Trestle libraries to implement Juno-2's graphical user interface.

## Overview of Juno-2

Juno-2 is a constraint-based drawing editor intended for the production of precise technical drawings. The main goal of the project was to build a usable constraint-based drawing editor with an extensible constraint definition language. We hoped to build a system to prototype constraint-based interfaces for several different domains from within the application itself.

Constraints allow you to specify locations in your drawing declaratively, rather than computing them imperatively. For example, to draw an equilateral triangle, you first draw an arbitrary triangle and then constrain its sides to have equal length; Juno-2 will adjust the vertices to make the triangle equilateral. Moreover, the constraints are maintained whenever part of the picture is changed, so constraints make it easier to cope with modifications.

Most drawing editors like MacDraw allow you to align objects in various ways, but those alignments are only temporary. By contrast, alignments based on Juno-2 constraints are permanent. This means that alignment is maintained even if an object is moved. Furthermore, constraints can be used to maintain symmetry.

Juno-2 is a double-view editor: it has a *graphics view* that displays a view of the picture as it would appear if printed, and a *text view* that displays a program in the Juno-2 programming language that corresponds to the picture. Editing either view causes the other view to be updated. We use software double-buffering to make dragging in the graphics view appear smooth. Our implementation of a re-usable software double-buffering object will be the subject of a future article.

Juno-2 is written entirely in Modula-3. It is about 26,000 lines of Modula-3 code, plus about 600 lines of Formsvbt form code to describe the user interface (see below) and 6,000 lines of embedded Juno-2 code. Currently, the system is not publicly available, but we hope to make the binaries available soon for various architectures.

## Implementing the Juno-2 User Interface

The Juno-2 user interface is built on top of Trestle, Modula-3's object-oriented window system toolkit. The version of Trestle included with src Modula-3 runs on the X window system; an implementation for Windows/nt and Windows/95 is in progress.

The basic Trestle abstraction is an object called a "virtual bitmap terminal", or vbt. Each vbt represents a share of the keyboard, mouse, and displays. vbts are comparable to the windows and widgets of other window systems.

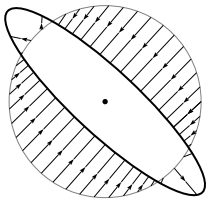
A Trestle application's user interface is structured as a tree of vbts. The vbt at the root corresponds to the top-level application window, and the vbts at the leaves generally correspond to the actual regions on the screen where painting occurs. The internal nodes of the tree are called *split* vbts. They divide their region of the screen between one or more children depending on the class of the split.

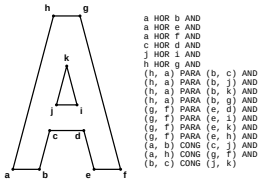
It is quite possible to build a user interface entirely out of Trestle vbts, but then the tree of vbts must be constructed explicitly in Modula-3 code. Since the design of a user interface is often an iterative process, recompiling and relinking the application after each change to the user interface is tedious.

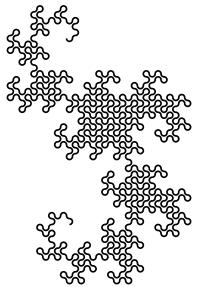
The *FormsVBT* library was designed to address this problem. The library provides procedures that read a declarative user-interface description called a *form* and construct the corresponding tree of vbts. Since the form file is interpreted at run-time, the turnaround time

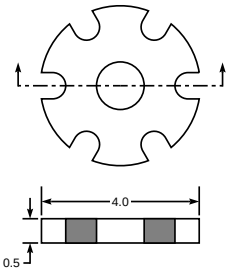
*Allan Heydon is a Member of Research Staff at Digital Systems Research Center. He is currently one of two people responsible for the maintenance of SRC Modula-3.*

*One of Allan's areas of research is in constraint-based drawing with Juno-2 as explored in this series of articles. For more information, see the Juno-2 home page on the web:*
http://www.research.digital.com/SRC/juno-2





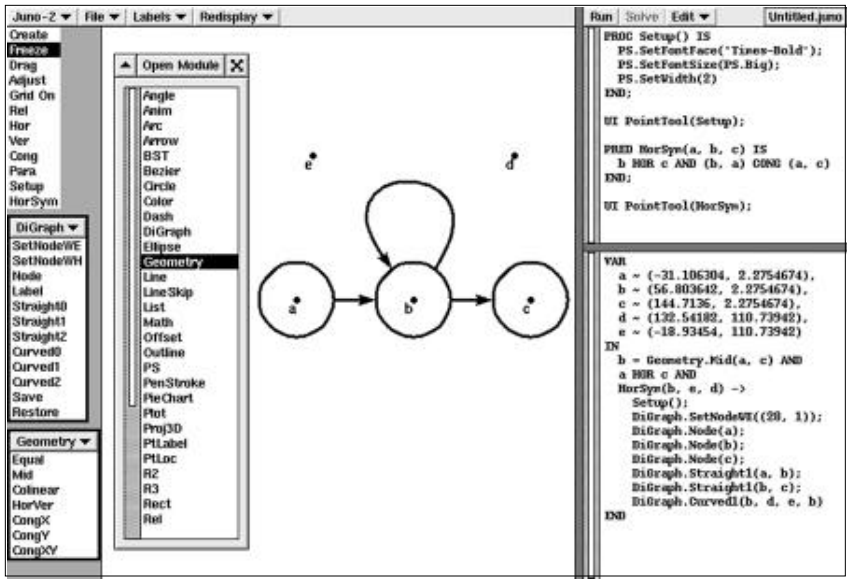


between changes to the user interface is significantly reduced. Moreover, src Modula-3 includes an application called *formsedit* that allows users to edit forms and immediately see how those forms will appear and behave when incorporated into a Modula-3 application. This separates the task of designing the appearance of the user interface from that of implementing its functionality.

The Formsvbt library is built on top of a library called *VBTkit*, which defines many vbt classes, all with a uniform 3D "look and feel." These include the standard buttons, menus, file browsers, and text editors of most windowing toolkits. But vbtkit also includes features like buttons and menu items that automatically pop-up sub-windows without requiring the programmer to write any callback procedures.

Figure 1 shows the Juno-2 user interface. Across the top is a row of pull-down menus. The left sub-window is divided into an area for tool palettes and the main drawing window. The right sub-window contains two text editors. The "Open Module" pop-up window contains a partial list of Juno-2's predefined modules.

*Figure 1*
*The Juno-2 User Interface*




A Formsvbt *form* is a symbolic expression that describes the structure and appearance of a Trestle-based user interface. It is structured as a list of nested *components*. The first element of the list is the component type, and the remaining arguments are either *properties* of the component (name-value pairs) or nested components for its children. For example, here is a stripped-down version of the top-level Juno-2 form (double semicolons begin comment lines):

```
(HTile
  ;; left-hand side
  (VBox
    (HBox
      ;; <<left menu-bar description goes here>>
      Fill)
    (Bar 1)
    (HBox
      (Generic %toolbox)
      (Bar 1)
      (Generic %drawing)))
  ;; right-hand side
  (VBox
    (HBox
      ;; <<right menu-bar description goes here>>
      Fill
      (Border
        (Text %currFile RightAlign
          "Untitled.juno")))
    (Bar 1)
    (VTile
      (TextEdit %currModule)
      (TextEdit %currCmd))))
```

Each component may have a name, as specified by the %name syntax. You can use the names to refer to other components within the form, or to name components from Modula-3 code. For example, you attach a callback procedure to a button by giving the button a name and invoking the FormsVBT.AttachProc procedure to associate the named button with a particular Modula-3 procedure.

In the above form, the symbolic expressions for the menus have been omitted. To give you an idea of what they're like, here's the symbolic expression for the Juno-2 menu on the far left, which has items named About... and Quit.

```
(Menu (MenuLabel "Juno-2")
  (VBox
    (PopMButton (For aboutWindow)
      (TextL "About..."))
    (MButton (Name quit) (TextL "Quit"))
  ))
```

In this form, the component names MenuLabel and TextL are names of user-defined *macros*. Formsvbt does textual substitution of macro occurrences by their instantiated definitions. For example, the expression (TextL <text>) gets

replaced by the expression (Text LeftAlign <text>), which causes the text to be left-aligned rather than centered. Macros make it easy to encapsulate common aspects of appearance in a central place.

Some Formsvbt components provide functionality that eliminates the need to associate callback procedures with them. For example, the PopMButton component in the previous example causes the sub-window named aboutWindow to pop up when that menu item is selected. The form describing the sub-window has a special kind of button called a CloseButton that causes the sub-window to disappear when the button is pressed. In Juno-2, CloseButtons are used to cancel dialogues.

vbtkit (and hence, Formsvbt) includes a wealth of components, many of which have been used in the implementation of Juno-2. For example, when a dialogue box is displayed, the rest of the application becomes passive to mouse clicks and typing. This is easily accomplished by wrapping the entire Juno-2 form in a *Filter* component. Since Filter components have active and passive states, it is a simple matter to switch the filter between its active and passive state by making Modula-3 procedure calls at the appropriate times.

As another example, we wanted to display the Juno-2 logo and a welcome message when the program started up, since there is a short delay while the program reads and compiles the bundled in Juno-2 modules. This was also easily accomplished by wrapping the top-level form in a *TSplit* component. A TSplit is a split that displays exactly one of its children at a time. Here's an example of the top-level form:

```
(TSplit %tsplit (Which startupScreen)
  ;; Juno-2 logo
  (Filter %startupScreen Passive
    (Cursor "XC_watch")
  (VBox
    Fill
    (Pixmap "Juno2Logo.ppm")
    (Glue 10)
    "Welcome to Juno-2! Initializing..."
    Fill))
  ;; Main window
  (Filter %background
    ;; <<main form from above goes here>>
  ))
```

This example illustrates the use of properties. In addition to the use of the Name property, the Which property has been associated with the TSplit component, and the Passive and Cursor properties have been associated with the start-up screen Filter component.

The Which property of the TSplit specifies which child the TSplit should display initially. When the application has finished its initialization work, it calls:

```
FormsVBT.PutInteger(w, "tsplit", 1)
```

to switch the TSplit component in w (the top-level window) to display child 1 (the children of a split are numbered starting from 0).

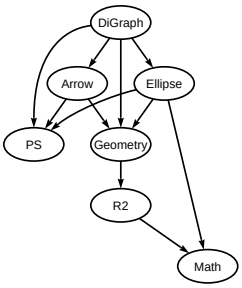
One nice aspect of the Formsvbt design is that properties are inherited. This makes it easy to make fairly global changes to the appearance of a user interface. For example, to change the background color of the entire form to a light grey, you could associate the property (BgColor "LightGrey") with the form's top-level component.

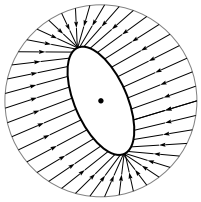
As mentioned before, Formsvbt includes a stand-alone application called *formsedit* for editing and testing forms. Formsedit provides a text editor for editing a form, and a previewer for viewing the instantiation of the form and for testing its behavior. Formsedit's rapid turn-around makes it easy to rapidly prototype different user interfaces.

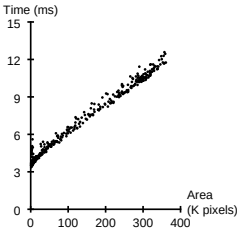
Not every component in your user interface will have a corresponding Formsvbt component. Typically, the "guts" of an application have application-specific vbts. In such cases, you can use a Generic Formsvbt component in your form, and then replace that component by the true vbt at run-time. This technique is used to replace the Juno-2 tool palette and drawing view.

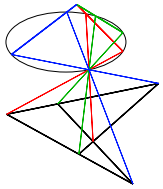
## Conclusions

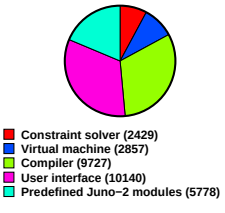
The Modula-3 distribution includes a window toolkit called Trestle, a widget library called vbtkit, a declarative user interface description system called Formsvbt, and a program for interactively constructing user interfaces called formsedit. The vbtkit and Formsvbt libraries, together with the formsedit interface builder, made it easy to prototype and build the Juno-2 user interface. vbtkit provided all the interface components we needed.







Like most modern windowing toolkits, the Trestle and Formsvbt libraries require that applications be structured as an initialization routine and a collection of callback routines. This general approach makes it difficult to build modal dialogues, since by default, the main window remains active when a dialogue sub-window appears. With Formsvbt, modal dialogues can be implemented by wrapping a *Filter* component around the top-level form, as described earlier.

Of course, with all the flexibility that Formsvbt provides, it's quite easy to create forms whose corresponding vbt trees contain hundreds of nodes. The Juno-2 form contains 559 Formsvbt components, which are translated into a total of 731 vbts.

Overall, the vbtkit and Formsvbt libraries provide a robust and flexible environment for constructing graphical user interfaces quickly. The structure and appearance of an interface is described in a simple, declarative language, and its functionality is implemented separately by callback procedures written in Modula-3.

## *How Modula-3 got its spots?* Initialization of Object Types

Greg Nelson, gnleson@src.dec.com
Digital Systems Research Center

*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 various Modula-3 libraries, such as Trestle and Network Objects.*

People often ask why Modula-3 does not distinguish initialization methods. The answer is that we have found it preferable to deal with this issue by a convention instead of a language feature.

The convention is to name one of the methods of a type *init*; this method is responsible for initializing a newly-allocated object of the type. A further convention is that the init method returns the object after initializing it. This is convenient; for example it allows code like this:

```
VAR tbl := NEW(HashTable.T).init(size := 50);
  BEGIN ...
```

instead of like this:

```
VAR tbl := NEW(HashTable.T);
  BEGIN tbl.init(size := 50); ...
```

If a default field value is provided when a type is declared, then that field will be initialized to that value in each newly allocated object. This can often be used to avoid the need for a explicit initialization method. For example, if you want an object to represent stacks of integers of size at most ten, you could declare

```
TYPE
  Stack = OBJECT
    sp := 0;
    data: ARRAY [0..9] OF INTEGER;
  METHODS ... END;
```

The .sp field of a newly-allocated Stack will be initialized to zero automatically. The data field will contain arbitrary values, but these values are irrelevant when sp is zero.

Default field values are frequently used to avoid init methods. For example, a newly-allocated MUTEX is ready to use: there is no need to call any init method. Explicit init methods are generally needed only for types whose initialization depends on parameters supplied by the client.

It is common for the initialization method of a type to call the initialization method of its supertype, as in the following example:

```
TYPE
  A = OBJECT ... METHODS init(...): A; ... END; ...
  AB = A OBJECT METHODS init(...): AB := InitAB;
  ... END

PROCEDURE InitAB(a: AB; ...): AB =
  BEGIN
    EVAL NARROW(a, A).init(...);
    (* initialize extra fields *)
    RETURN a
  END InitAB;
```

Notice that the AB init method is a new method, not an override of the A init method. This is typical: the init method signature for a subtype tends to be specific to that subtype.

Also notice that NARROW(a, A) views a as being of type A; the result is to call A's init method instead of AB's. No run-time check is required by this NARROW. Finally, notice that EVAL is required to discard the result returned by A's init method.

If the language were changed to distinguish initialization methods, the compiler might call them automatically, or it might issue an error if an object were used without being initialized. This might make the language more robust, and is the main argument for dealing with initialization in the language definition instead of by convention. We rejected the argument for several reasons.

First, it is problematic to have the compiler call the init method automatically. For example, in the InitAB procedure shown previously, the arguments to the supertype init method may be arbitrary expressions. They certainly need not be the same as the argument list of InitAB, or a prefix of the argument list, or anything like that. Also, the call to the supertype init method is not always first or last within the subtype init method: it may occur in an intermediate position.

Second, if one checked (either statically or dynamically) that at least one init method has been called for each allocated object before the object is used, the resulting sense of security would be far from complete. After all, guaranteeing that the init method has been called does not guarantee that it has actually initialized every field that will be used.

Third, if careful checking is required for some type, the implementor of the type can easily produce it by defining a boolean field "initialized" with a default value of *false*; setting it to *true* in the initialization method, and test it in the other methods. By hiding the field in the private part of the implementation, the implementor can guarantee that no error in a client program can cause the implementation to compute with an uninitialized object.

## *Modula-3 in Industry*

## Photon: An Environment for Building Distributed Applications

Lauren Schmitt, lps@cmass.com
Critical Mass, Inc.

Most people who have used Modula-3 agree that it provides a great deal of support for today's programming tasks. Garbage collection, thread support, exception handling, and a complete set of robust and well-documented libraries are among the features that make Modula-3 quite suitable for building large-scale robust distributed applications. Unfortunately, most people who may consider using Modula-3 in a commercial setting also agree that, without a commercial implementation and support for Modula-3, it is difficult if not impossible to convince management to allow programmers to use Modula-3.

We have taken the first step toward solving this problem. I am pleased to announce that we at Critical Mass, Inc. have been working on a commercially-supported development environment, named Photon, based on src Modula-3. Photon integrates Modula-3 into a familiar user interface: that of a web browser, allowing the user to readily browse, build, and manage large projects locally or over a network.

We intend to distribute Photon for a low price with an open license to make it easily available to the masses. As part of our efforts, we will also offer commercial support for Photon. We hope Photon will be the first in a family of Modula-3 based products distributed by Critical Mass.

A preview release of Photon will be available before end of 1995 leading to an official release in the first quarter of 1996. Our first target platform for Photon is Linux; we intend to support other platforms as the demand requires. More information regarding Photon will be posted on Modula-3 newsgroup and will be available via our home page, http://www.cmass.com.

*Lauren Schmitt is the founder of Critical Mass, Inc. Critical Mass will be the first commercial provider of the Modula-3 language and accompanying support. For more information visit the Critical Mass web page:*
http://www.cmass.com/

*LaszIo Boeszoermenyi and Carsten Weich teach at University of Klagenfurt, Austria.*

*Their book, "Programming with Modula-3: An Introduction to Programming with Style," ISBN 3-540-57911-7 has been published in German. An English translation will soon follow by Springer-Verlag.*

*For more information on the DOS port, see:* http://www.research.digital.-com/SRC/modula-3/html/m3pc.html.

## *Modula-3 in Academia*

**University of Klagenfurt, Austria**

Laszlo Boeszoermenyi
laszlo@ifi.uni-klu.ac.a

Carsten Weich
carsten@ifi.uni-klu.ac.at

University of Klagenfurt, Austria

### Teaching

We first experimented with Modula-3 as a teaching tool in 1992. In a course on computer networks, students implemented a number of different networking protocols in Modula-3. In 1993 a working group was established to discuss a new programming language to replace Modula-2 as the language for the first programming course. In making this decision, we felt that the most important criteria in picking a language were: traditional structured programming concepts, support for object-orientation, and availability of good compilers for both the Unix and dos environments.

In the final evaluation phase the following programming languages were considered: Eiffel, C++, Oberon-2, Ada, Modula-3 and Turbo-Pascal 6.0. The winner was Modula-3. Consequently, in 1993 we introduced Modula-3 as the first and second courses programming language. Our first experiences have been very promising.

To support the curriculum, we have written an introductory text for programming that uses Modula-3 for its examples. It's called *Programming with Modula-3: An Introduction to Programming with Style* which has appeared in German by Springer-Verlag and is in the process of being translated to English. The main concern of the book is to give a clean and comprehensive introduction to programming for beginners of a computer science study. We start with more traditional programming concepts and move toward advanced topics such as object-oriented programming, parallel & concurrent programming, exception handling, and persistent data techniques. The book also presents a large number of complete examples written in Modula-3.

### DOS Port

At the end of 1992 we made available the first DOS port of Modula-3. The port was made by Klaus Preschern, as a member of the research group of Professor Boeszoermenyi. This system is steadily used by our students, and is available freely for others via the Modula-3 home page.

### Research

The research group of Professor. L. Boeszoermenyi (Karl-Heinz Eder, Andreas Stopper, Carsten Weich) in cooperation with Professor. J. Eder and M. Dobrovnik, are working on the a variety of research projects related to Modula-3.

### PPOST

ppost (parallel persistent object store) is an object store implemented in Modula-3 that keeps all data all the time in main memory of a distributed set of computer nodes. ppost uses two kinds of parallelism: horizontal and vertical.

In *horizontal parallelism*, objects are kept in sets (called *classes*), which can be distributed "horizontally" among a number of worker nodes. With the help of horizontal parallelism, we can store large amounts of data in main memory (several gigabytes and conceivably several terabytes). Data is accessed with the speed of internal memory.

*Vertical parallelism* provides persistence (write to disk) in parallel to normal transaction processing taking place on ppost. All modifications of data are stored in a log file by the *log processor*. With the help of the log information and the backup image of the database, a new actual database image on the disk can be produced by a backup node. As such, disk management is actually parallel to normal transaction processing and does not slow down data processing.

### Parallel and Persistent Sets

General purpose programming languages still consider persistence and parallelism as features of secondary importance. Such features are usually added later with the help of some library modules. This has the disadvantage of the loss of type safety and optimizations of some operations. Typically, programming lan-

guages supporting highly parallel architectures are based on the array model (e.g. Vienna Fortran, Modula-2*, Modula-3*) as the array model is quite adequate for scientific computing. It is, however, inadequate for the manipulation of a large amount of data required for many information systems.

As an alternative, the set model is appropriate to manipulate a large amount of data. It also supports parallelism well; sets are by definition an unordered collection of data, therefore no assumption about the order of processing of the elements needs to be made. We lift the restrictions on sets in Modula-3 and extend the language to introduce typed, polymorphic sets that may contain compatible types. This allows us to handle a large amount of data that maybe physically distributed over a number of processors in a convenient way. Only a few syntactic additions are necessary to allow access to a powerful new feature. The implementation does not have to be heavy-weight either; we provide several alternative implementations of a set for various uses.

### Parallel OO Simulations

Simulation has always been a main research direction of object-orientation. Most work on parallel simulation concentrates on discrete event simulations, which has the disadvantage that it cannot take advantage of parallelism inherently available in the problem easily. With the help of Modula-3 Network Objects we try to utilize this inherent parallelism and provide models which preserve this parallelism as much as possible. Such a model can be more easily mapped to a true parallel architecture.

## *Advanced Research Topics*
## The Whole Program Optimizer

Amer Diwan, diwan@cs.umass.edu
University of Massachusetts at Amherst

As part of our efforts to improve the performance of object-oriented programs, in particular, Modula-3 programs, we have developed the Whole Program Optimizer (wpo). The wpo optimizes multiple modules at once and thus has more information available to it than traditional optimizers that optimize one module at a time. The wpo contains a number of analyses and optimizations that take advantage of the increased information. In the remainder of this article we give examples of the optimizations performed by the wpo, and describes the structure of the wpo. The following example illustrates the kinds of optimizations targeted by the wpo:

```
INTERFACE TU;
TYPE
  Public = OBJECT METHODS f (); END;
  T <: Public;
  U <: T;
PROCEDURE foo (t: T);
END TU;

MODULE TU;
REVEAL T = Public BRANDED OBJECT
    OVERRIDES
      f := Tf;
    END;
REVEAL U = U BRANDED OBJECT
    OVERRIDES
      f := Uf;
    END;
PROCEDURE foo (t: T) =
  BEGIN
    t.f ();
  END;
BEGIN
END TU;

MODULE C;
IMPORT TU;
BEGIN
  TU.foo (NEW (TU.U));
END C;
```

In procedure foo, the method invocation could invoke either procedure Tf or Uf. However, if the wpo has access to all the clients of TU (for example, in module C) it can determine that the concrete type of formal t can only be U and therefore the method invocation can be replaced with a direct call to Uf.

*Amer Diwan is a graduate student and a member of Object Systems Laboratory at the computer science department of the University of Massachusetts at Amherst. Amer's current research interests are in compiler optimizations and language run-time systems.*

*For more information on his project, see:*
http://osl-www.cs.mass.edu/~oos.

Replacing a method invocation by a direct call speeds up the program by removing method lookup overhead and also by enabling other optimizations such as inlining. In this example, this optimization required knowledge of modules other than the one being optimized.

Currently, the wpo is about 5000 lines of Modula- 3 code. Figure 1 illustrates how the wpo fits into a compilation framework.

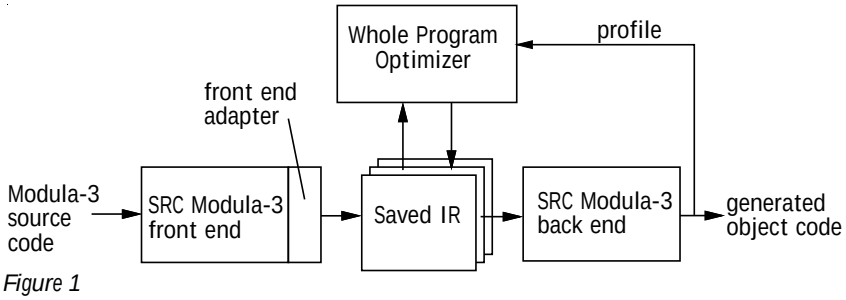



*Figure 1*
*Compilation Framework*

The inputs to the wpo are a collection of code files and an optional profile file. Each *code file* contains code for a Modula-3 module or interface; the code is represented as an annotated abstract syntax tree (ast). The ast is annotated with full type and source level information, including source line numbers. All properties of the source modules that may be useful for optimizations and source-level debugging are maintained. The Modula-3 pickle facility is used to read and write the ast files readily. We have modified the src Modula-3 front end to write the typed ast to pickle files. Each node in the ast has a gen method which generates the stack intermediate representation (by invoking the M3CG.T methods) which is the input to the back end.

We have implemented a collection of analyses and optimizations in the wpo. Currently, the wpo removes the overhead of opaque types, and performs cloning, "if" conversion (similar to transformations in the Self compiler), type analysis for heap allocated data, and interprocedural concrete type inference to remove the overhead of critical method invocations. In addition, the wpo performs some traditional compiler optimizations such as procedure inlining.

An important component of the wpo is the data-flow engine which is implemented as a generic module and solves forward-flow data flow problems for any domain. We use the data-flow engine to implement use-def analysis and type inference.

While the wpo is most effective when the entire program is available to it, it can also be used on subsets of the program. In particular, most of the time we don't optimize the standard libraries. To reduce the memory requirements, the wpo can be run in a mode where it keeps only the modules it needs at any given time in memory rather than the entire program.

We found Modula-3 to be well-suited for the implementation of the wpo. The object model, opaque types, generic modules, an excellent collection of standard libraries, and most importantly garbage collection have all contributed to an implementation that is easy to understand and extend.

We also found Modula-3 programs to be well-suited to optimizations. The small size of the language definition simplifies the design and implementation of optimizers. Also, the isolation of unsafe features allows the optimizer to be more aggressive on safe modules since it can make stronger assumptions about concrete types of variables. In unsafe languages, features such as arbitrary type casts can violate these assumptions and thus cannot be made by the optimizer.

We have recently started experimenting with the wpo. Preliminary results are promising. The analysis alone in wpo are able to convert up to 50% of method invocations to direct calls in our benchmark programs.

You are cordially invited to
a meeting of the
Modula-3 User Group
at OOPSLA'95, in Austin, Texas

This meeting is open to all, especially current users and others interested in learning more about Modula-3! Our meeting will include presentations on what people are doing with Modula-3 in various organizations. **Craig Chambers** & **Jeff Dean** of University of Washington will describe their work on applying advanced **optimizations** to Modula-3 as part of the Vortex project. **Allan Heydon** of DEC Systems Research Center will present his prototype extensible **constraint-based** drawing package written in Juno-2, an **augmented reality** system written using Modula-3 Network Objects. **Lauren Schmitt** of Columbia University will present his prototype Critical Mass Inc will unveil Photon, a **programming environment** written for Modula-3. **Blair MacIntyre** of Columbia University will present his prototype Modula-3 such as: ports to PC OSs (win/NT, win'95, OS/2, Linux), evangelizing, commercial use, academic use, and building distributed applications with Modula-3.



We also hope to cover a variety of topics about Modula-3 and research use, and building

Modula-3 is a simple and modular programming languageproviding facilities for exception handling, concurrency, object-oriented programming, automatic garbage collection, ad systems programming without involving the complexities forced by other languages of its class. Modula-3 is both a practical implementation language. An for large software projects and an excellent teaching language. An implementation of Modula-3 is available free of charge from Digital Systems Research Center.

Please join us!

Organizers:
Farshad Nayeri &
Geoff Wyant

Modula-3

USER GROUP MEETING
OOPSLA '95

**when** & **where** check birds of a feather (BOF) listings

*Introducing*


http://www.cmass.com/

- network objects
- threads
- distributed garbage collection
- partial revelation
- exception handling
- generics

a distributed programming environment for Modula-3.

coming soon from


*You've just written your last destructor!*