# THREADS

Modula-3 Systems Journal • Issue 3 • Fall 1997

## *Threads*
## *The Modula-3 Systems Journal*

We are pleased to bring you the third issue of *Threads, The Modula-3 Systems Journal*. By publishing *Threads* we hope to establish a forum for discussion about Modula-3 and about what various industrial and academic organizations are doing with Modula-3. The articles are intended to be 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 contributions 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. 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/

Introductory information about Modula-3 is also available from the Modula-3 Web Resource at:

http://www.m3.org

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

http://www.cmass.com/reactor

## *Issue 3*
## Table of Contents

*Threads, The Modula-3 Systems Journal is a publication of Critical Mass, Inc. For more information, contact:*
*Critical Mass, Inc., 225R Concord Ave., Cambridge, MA 02138 USA, tel. +1 617 354 6277. email: threads@cmass.com.*

*Feature Article*

# Critical Mass JVM: Modula-3 Befriends Java

*Farshad Nayeri, Critical Mass, Inc.*
*Blair MacIntyre, Columbia University*

*JVM, Critical Mass's implementation of the Java Virtual Machine, is written entirely in Modula-3.*

*In this article, Farshad Nayeri and Blair MacIntyre describe how they used JVM to build a Java-extensible serial port controller in Modula-3.*

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

*Blair MacIntyre is a graduate student at Columbia University.*

The success of Java has popularized many of the concepts that Modula-3 has been promoting for years, particularly the combination of garbage collection, exception handling, objects, and threads. Given the similarities between the two languages, and the strength of Modula-3 as a systems programming language, it is possible to exploit this common run-time infrastructure and build a Java virtual machine (JavaVM) implementation in Modula-3. Such an implementation will be better-structured than the equivalent C-based implementation since much of the hardest work, such as multi-threaded garbage collection, has already been done.

This is exactly how *Critical Mass JVM* was conceived. Adding to the impressive list of systems built in Modula-3, such as operating systems, network object systems, windowing systems, compilers, internet distribution and commerce systems, we can now point to Critical Mass's implementation of the JavaVM as another example of the power of Modula-3 for building systems that work.

## What is Critical Mass JVM?

JVM is a clean-room implementation of the Java Virtual Machine: it was based entirely on the JavaVM specification and other publicly available information, not on existing JavaVM implementations. The Critical Mass JVM is class-level compatible with Sun's JavaVM.

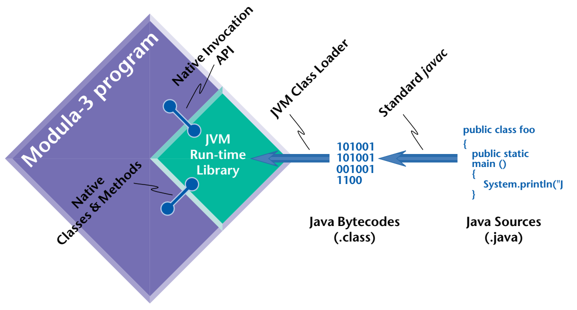
What's unique about JVM is that it's written entirely in Modula-3, and as such it carries many of the benefits of Modula-3's mature, well-designed infrastructure. In JVM, a single run-time system supports both Java and Modula-3, providing an unparalleled level of integration between the two languages.

Like most other Modula-3 packages, JVM is implemented as a "component" (or in old-fashioned terms, as a reusable library.) Therefore, you can easily integrate JVM within new or existing applications written in any language, including Modula-3, C, or C++. In addition, JVM is designed so that it can be easily reconfigured and extended. The internal interfaces to JVM are available to the integrator, allowing his application to be integrated more tightly with JVM. JVM makes for a great migration path to Java, as well as a nice testbed for extending, enhancing, and researching the JavaVM. If you like both Java and Modula-3, then JVM is a great way of mixing them together!

## Java *a la* Modula-3

Java and Modula-3 are sufficiently similar that it makes sense for them to share a common runtime, instead of pushing for a strictly "layered" architecture. You can imagine that the same runtime might support both Java and Modula-3 programs using the same garbage collector, the same threads, and the same exception handling system. The trick is to reconcile the differences in the runtime support so that both sides can use this shared runtime. (This turns out to be easier to imagine than to implement--the devil is in the details.)

Integration of the runtime systems is exactly how Critical Mass JVM was built. In the process of building JVM, portions of the Critical Mass Modula-3 runtime were redesigned to add support for some useful extensions, such as mapping runtime errors into exceptions, unicode characters, and dynamic loading. These features were used in turn to implement the JVM. This article won't go into the Modula-3 runtime extensions; they are likely to be the subject of a future *Threads* article.



The resulting system is quite useful for building large, intricate systems that require tight integration with Java. Indeed, Critical Mass JVM uses an identical layout for the fields of its objects as Critical Mass Modula-3, allowing you to assign values to fields of Java objects from your Modula-3 code. That is, given a Java object, it is possible to write a Modula-3 object definition that allows you to access the fields of the Java object *directly* from Modula-3. Methods, on the other hand, are not mapped between languages automatically, because the object models of Java and Modula-3 are sufficiently different that such a mapping either would have to unify both object models, or it would have to support a subset of each language. Neither one of these solutions seem useful. However, you can bind Modula-3 procedures as native Java methods via calls to the JVM runtime, allowing Modula-3 code to be called directly from Java. Conversely, Java methods can be called from Modula-3 by name.

## JVM Example: Extending JVM with serial port access

As an example, we now show how JVM can be extended in Modula-3 to support serial port access from Java. For this we will use the portable Modula-3 SerialPort interface, which is distributed as part of the Reactor system:

```
INTERFACE SerialPort;
IMPORT OSError, Pathname, Terminal;
```

SerialPort.T represents a serial communications device. It is a subtype of File.T that can be read or written. Various configuration details of the underlying device, such as its baud rate and parity, can be read or written.

```
TYPE
  T <: Public;
  Public = Terminal.T OBJECT METHODS
    get_config (): Config
        RAISES {OSError.E};
    set_config (READONLY config: Config)
        RAISES {OSError.E};
  END;
```

Given an open serial port s, s.get_config() returns the current configuration of the underlying device. s.set_config(cfg) configures the underlying device to correspond to cfg.

```
TYPE
  Config = RECORD
    baud_rate : BaudRate;
    data_bits : DataBits;
    stop_bits : StopBits;
    parity    : Parity;
    DTR_mode  : DTR;
    RTS_mode  : RTS;
  END;

TYPE
  BaudRate = {BR75, BR110, BR300, BR600, BR1200,
              BR2400, BR4800, BR9600,
              BR14400, BR19200, BR38400,
              BR56000, BR128000, BR256000};
  DataBits  = {DB8, DB7, DB6, DB5};
  StopBits  = {SB1, SB15, SB2};
  Parity    = {None, Odd, Even, Mark, Space};
  DTR       = {Disabled, Enabled, Handshake};
  RTS       = {Disabled, Enabled, Handshake,
              Toggle};

CONST
  InitialConfig = Config {
      BaudRate.BR9600, DataBits.DB8,
      StopBits.SB1, Parity.None,
      DTR.Disabled, RTS.Disabled
  };

PROCEDURE Open (p: Pathname.T): T
                RAISES {OSError.E};
```

This procedure opens the serial device named p and returns a SerialPort.T that can read and write the device. The initial configuration of the result is set to InitialConfig.

```
END SerialPort.
```

Note that this Modula-3 serial port interface itself has nothing to do with Java; it provides portable serial port access to Modula-3 programs on Windows and Unix. A SerialPort.T is a subtype of File.T; this is a nice property because you can mix and match serial ports with other Modula-3 libraries. For example, you can create a standard reader or writer for a serial port.

## Interfacing to Java via JVM

Next, we designed a Java package that provides functionality similar to the Modula-3 SerialPort interface in Java. Despite the similarities, note the number of differences between the structure of these two "interfaces":

```
package cmass;

// SerialPort is a JVM native file descriptor
// that maps to a serial port of the system
// where JVM runs

public class SerialPort {

/* File Descriptor - handle to the open file */
  private java.lang.io.FileDescriptor fd;

/* Opens the specified serial port. */
  private native void open(String name)
    throws java.io.IOException;

/* Closes the serial port. */
  private native void close()
    throws java.io.IOException;

/**
* Creates a SerialPort file with the specified system
* dependent file name. Throws FileNotFoundException
* if the file is not found.
*/
  public SerialPort(String name) throws java.io.FileNotFoun-
dException {
    try {
      this.fd = new FileDescriptor();
      this.open(name);
    } catch (IOException e) {
      throw new java.io.
        FileNotFoundException(name);
    }
  }

/* Constants for configuring the serial port. */
/* The baud rate. */
  public static final short BR75 = 0;
  public static final short BR110 = 1;
  public static final short BR300 = 2;
  public static final short BR600 = 3;
  public static final short BR1200 = 4;
  public static final short BR2400 = 5;
  public static final short BR4800 = 6;
  public static final short BR9600 = 7;
  public static final short BR14400 = 8;
  public static final short BR19200 = 9;
  public static final short BR38400 = 10;
  public static final short BR56000 = 11;
  public static final short BR128000 = 12;
  public static final short BR256000 = 13;

/* the number of data bits */
  public static final short DB8 = 0;
  public static final short DB7 = 1;
  public static final short DB6 = 2;
  public static final short DB5 = 3;

/* the number of stop bits */
  public static final short SB1 = 0;
  public static final short SB15 = 1;
  public static final short SB2 = 2;

/* the parity */
  public static final short PARITY_NONE = 0;
  public static final short PARITY_EVEN = 1;
```

```
    public static final short PARTIY_ODD = 2;
    public static final short PARITY_MARK = 3;
    public static final short PARITY_SPACE = 4;

/* DTR line control */
    public static final short DTR_DISABLED = 0;
    public static final short DTR_ENABLED = 1;
    public static final short DTR_HANDSHAKE = 2;

/* RTS line control */
    public static final short RTS_DISABLED = 0;
    public static final short RTS_ENABELD = 1;
    public static final short RTS_HANDSHAKE = 2;
    public static final short RTS_TOGGLE = 3;

/**
* Sets the configuration of the serial port. The
* array parameter contains the following elements:
* [baud_rate, data_bits, stop_bits, parity,
* DTR_mode, RTS_mode, timeout readInterval,
* timeout readMultiplier, timeout readConstant].
*/
    public native void setConfig(short[] cfg);

/*
* Gets the current configuration of the serial port.
* The array parameter will contain the following
* elements: [baud_rate, data_bits, stop_bits,
* parity, DTR_mode, RTS_mode,
* timeout readInterval, timeout readMultiplier,
* timeout readConstant].
*/
    public native void getConfig(short[] cfg);

/**
* Returns the opaque file descriptor object
* associated with this stream.
*/
    public final FileDescriptor getFD()
            throws IOException {
        if (this.fd == null) throw new IOException();
        return this.fd;
    }

/* Cleans up if the user forgets to close it. */
    protected synchronized void finalize()
            throws IOException {
        if (this.fd != null) this.close();
    }

} // class SerialPort
```

This class simply exposes the serial port features to Java. You can compile this file with a standard Java compiler (such as Sun's javac.) Of course, the resulting class file expects the Java runtime to include native implementations for the methods defined in this class. If you were to run this with an ordinary JavaVM without the serial port extensions, it would not work. So, the next step is to extend the JVM to include support for objects of this class.

Note the private "fd" field of this object, which is of type FileDescriptor. Ideally, we would like to do the Java equivalent of subtyping File.T, namely having our SerialPort class extend java.lang.io.FileDescriptor, but this is not possible because the Java Language Specification declares FileDescriptor to be final (meaning that this class cannot be extended.) Even if JVM allowed FileDescriptor to be extended, javac, the Java source-to-bytecode compiler would not allow the code for cmass.SerialPort to compile. So, we end up with the more cumbersome (and less efficient) approach of including a FileDescriptor field in our object. (There is precedence for this approach in the Java libraries that implement TCP/IP sockets, which should also be subtypes of FileDescriptor.) The contrast of the two serial port "interfaces" also illustrates some useful constructs missing from Java, e.g., an enumeration type.

## Implementing native Java methods in Modula-3

To implement the native methods, we create a JVM_SerialPort module in Modula-3:

```
INTERFACE JVM_SerialPort;
IMPORT java, JVM_File;
CONST     ClassName = "cmass.SerialPort";
TYPE
   T = java.object BRANDED ClassName OBJECT
      fd: JVM_File.FileDescriptor := NIL;
   END;
END JVM_SerialPort.
```

JVM_SerialPort.T is a mirror image of cmass.SerialPort. Note how its fd field's type is the mirror image of java.lang.io.FileDescriptor, JVM_File.FileDescriptor.

Finally, we implement the JVM_SerialPort module.

```
MODULE JVM_SerialPort;
IMPORT SerialPort, JVM, JVM_Interp, java,
           JVM_Error, AtomList, OSError, JVM_String,
           JVM_ArrayClass;
```

First, create an array of method descriptions. For each method, declare the class name ("cmass.SerialPort"), the method name, its Java type, and the name of the Modula-3 procedure implementing it. The somewhat cryptic Java types are a legacy of Sun's JavaVM implementation, and are documented in the Java Virtual Machine Specification. (We leave as an exercise to the reader to decrypt them! Hint: see the method signatures of the cmass.SerialPort class.)

```
CONST  NativeMethods = ARRAY OF JVM.MDesc {
   JVM.MDesc { ClassName, "open",
      "(Ljava.lang.String;)V", Open},
   JVM.MDesc { ClassName, "setConfig",
      "([S)V", SetConfig},
   JVM.MDesc { ClassName, "getConfig",
      "([S)V", GetConfig},
   JVM.MDesc { ClassName, "close", "()V",
      Close}
};
```

The next procedure, Open implements the open call, which is declared to take a pathname as its argument. To access its parameters, Open pops them off the JVM stack. Here, there is only one argument to be popped off, plus the implicit self argument. We take advantage of Modula-3's automatic narrow by declaring self to be of type T, thereby allowing us to refer to self.fd.

```
PROCEDURE Open (VAR env: JVM.Env;
                         m: JVM.Method)
                           RAISES {java.failure} =
VAR
   path : java.string := JVM_Interp.PopObj (env);
   self : T := JVM_Interp.PopObj (env);
   name := JVM_String.ToText (path);
BEGIN
   TRY
      self.fd.file := SerialPort.Open (name);
   EXCEPT OSError.E (err) => IOError (env, err)
   END;
END Open;
```

(For simplicity, Open doesn't check the validity of its arguments. We skip over the implementation of Close, since it is similar to Open.)

You may wonder what happens if you want to access fd concurrently from Modula-3 and Java. The answer is that, just as if you were building two components in pure Java or Modula-3, the two sides will have to synchronize with each other. In particular, you can use the usual per-object locks available in Java. To access the object from Modula-3, you will have to lock the object via the call JVM_Thread.Lock.

Next, consider the native implementation of the setConfig method, which takes a configuration array. Note how we can mix and match Java data structures with Modula-3 ones. (We skip getConfig's implementation, since it is similar to setConfig.)

```
CONST ConfigArraySize = 9;

PROCEDURE SetConfig (VAR env: JVM.Env;
                     <*UNUSED*> m: JVM.Method)
                     RAISES {java.failure} =
VAR
   x : java.short_array := JVM_Interp.PopObj (env);
   self: T := JVM_Interp.PopObj (env);
   cfg : SerialPort.Config;
BEGIN
   IF NUMBER(x.elts^) # ConfigArraySize THEN
      ArgError(env, "illegal array size");
   END;
   cfg.baud_rate := VAL(x.elts[0],
                              SerialPort.BaudRate);
   cfg.data_bits := VAL(x.elts[1],
                             SerialPort.DataBits);
   cfg.stop_bits := VAL(x.elts[2],
                             SerialPort.StopBits);
   cfg.parity := VAL(x.elts[3],SerialPort.Parity);
   cfg.DTR_mode := VAL(x.elts[4],SerialPort.DTR);
   cfg.RTS_mode := VAL(x.elts[5],SerialPort.RTS);
   cfg.timeouts.readInterval := x.elts[6];
   cfg.timeouts.readMultiplier := x.elts[7];
   cfg.timeouts.readConstant := x.elts[8];

   TRY
      TYPECASE self.fd.file OF
      | SerialPort.T (port) =>
         port.set_config(cfg);
      ELSE
         ArgError(env, "non-serial port file");
      END;
   EXCEPT OSError.E (err) => IOError (env, err)
   END;
END SetConfig;
```

The procedure IOError raises a java.failure exception from Modula-3, passing enough information so that meaningful backtrace information can be produced in Java (including Modula-3 line numbers in Java's backtraces!) The procedure ArgError is similar to IOError, except that it raises an exception of a particular type, namely a standard IllegalArgument exception.

Finally, the main body of the module registers the native methods using JVM.RegisterNativeMethods, which conveniently takes an array, and the class name "cmass.SerialPort" is registered to map to JVM_SerialPort.T.

```
BEGIN
   JVM.RegisterNativeMethods (NativeMethods);
   JVM.RegisterNativeType (ClassName, TYPECODE (T));
END JVM_SerialPort.
```

As you can see, it is quite simple to extend JVM with Modula-3 code. The nice part is that in your extension, you can take full advantage of Modula-3 features, such as exceptions, garbage collection and threads. These features are unavailable inside other JavaVM implementations; there you drop to ground zero when you start to write native methods, foregoing all access to these features.

Note that, as with Modula-3's <*EXTERNAL*> pragma, it is the programmer's responsibility to ensure that the mapping from Java to Modula-3 is correct. The programmer is still aided by the strong typing of both languages. This responsibility is a small price to pay for the tight degree of integration provided by JVM.

## The results

Once we have built the new JVM with serial port support, we are ready to test it. Extending JVM with serial port access enables a new class of applications. Here is an example use of a Java class that allows us to control an LCD display that is connected to the host computer via a serial line interface (implemented as the SLI class, which is not described here):

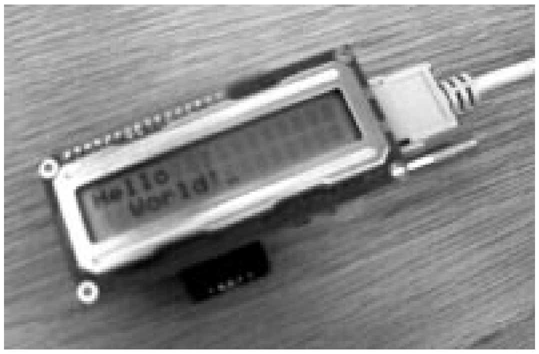
```
import java.io.*;
import cmass.*;

class app {
   public static void main (String argv[]) {
      try {
         SLI sli = new SLI("/dev/cua1");
         PrintStream ps = sli.ps;
         ps.print("\fHello\r  World!");
         ps.flush();
         Thread.sleep(1000000);
      } catch (InterruptedException e) {
         System.err.println("sleep failed");
      } catch (java.io.FileNotFoundException n) {
         System.err.println("serial port not found");
      } catch (java.io.IOException e) {
         System.err.println("I/O error");
      }
   }
}
```

And here is the result:

## Conclusion

JVM, an extensible Java Virtual Machine, is implemented in Modula-3. JVM makes it easy to extend Java functionality using Modula-3 (or C), and to load Java classes as part of Modula-3 programs. JVM also implements the native methods of the core language packages for Java, so it is easy to interoperate with



most Java programs. Finally, JVM can be extended easily to accommodate systems programming needs. More information about JVM can be found at http://www.cmass.com/jvm.

*Advanced Topics*

## Low-Level Systems Programming with Modula-3

*Marc E. Fiuczynski, University of Washington*
*Wilson C. Hsieh, University of Utah*
*Emin Gün Sirer, University of Washington*
*Przemyslaw Pardyak, University of Washington*
*Brian N. Bershad, University of Washington*

*The SPIN group at the University of Washington has been using Modula-3 since 1995 as both the kernel and extension language for their extensible operating system. In this article, they describe the set of additions they have made to extend Modula-3's support for low-level systems programming.*

*For more information on SPIN, visit:* http://www.cs.washington.edu/research/projects/spin/www

Since 1995 we have used Modula-3 to develop operating system services for a kernel called SPIN at the University of Washington [1]. SPIN is an extensible kernel that allows untrusted applications to extend system services by dynamically linking in *extensions* [4]. The SPIN kernel provides in-kernel threads, virtual memory, and device management as its core set of services, and higher-level operating system services are implemented by extensions. To date we have implemented a variety of extensions in Modula-3, including user-level threads, a TCP/IP protocol stack, transaction services for databases, a log-structured file system (LFS), NFS and HTTP servers, a Unix emulation library that is binary compatible with Digital UNIX and FreeBSD, and many other services. The kernel itself is also implemented in Modula-3.

We have found Modula-3's type safety, as well as its data hiding properties, threads, generic interfaces and object support, to be effective in developing a high-performance, modular system [5]. While building SPIN, we have learned that low-level system programming requires explicit language constructs to interact with the outside world. Some examples of interactions with entities outside the domain of a Modula-3 program are interfacing with other languages, manipulation of the underlying hardware, and interpreting data from devices. Modula-3 greatly facilitated the rapid construction of SPIN and its extensions, but there were a few additions and modifications we needed to make to the language and run-time to support low-level programming.

In this article we describe our changes to support low-level systems programming, with Modula-3 as follows:

- *Interfacing with foreign code and devices.* Since our kernel interfaces with hardware device drivers and services implemented in C, we improved Modula-3's capability to interface with other languages, and added a run-time feature to enable sharing of garbage collected data.
- *Implicit exceptions.* We added support to turn checked run-time errors and other system faults into language exceptions. This feature is important in an extensible system like SPIN, as it enables services to isolate themselves from errors caused by other subsystems.
- *Type-safe casting.* We added language support for type-safe casting. We needed this facility because operating system services often operate on data created outside of the language, such as packets

coming from the network, disk buffer blocks, and system call arguments.

The rest of this article addresses each of the above points.

## Interfacing with foreign code and devices

Our kernel relies on low-level platform-specific services that we borrow from Digital UNIX and FreeBSD. These services are implemented in C and highlight some of the difficulties of interfacing with other languages and devices. There are three aspects to these difficulties: argument passing between Modula-3 and foreign code (including issues related to Modula-3's garbage-collected heap), calling procedures across the language boundary, and the alignment of shared data. The rest of this section addresses these issues.

### Passing data between Modula-3 and foreign code

Passing non-reference arguments between a foreign language and Modula-3 simply requires the declaration of matching types. Matching up reference types is complicated, as Modula-3 treats references to data allocated on the traced (i.e., garbage-collected) and untraced heaps differently. The traced heap is automatically managed in ways that are not compatible with sharing of memory between safe and unsafe languages. For example, the DEC SRC reference implementation uses a copying garbage collector. Consequently, if a Modula-3 program passes a traced reference to C, and C stores the reference in its own untraced heap, the collector might relocate or collect the object, leaving C's reference dangling. Conversely, if a C program passes an untraced reference to Modula-3, and Modula-3 treats the reference as traced, the collector will fail.

To pass reference types from a foreign language (typically C, although occasionally assembly language) to Modula-3 we follow one of two strategies on the Modula-3 side. We either declare the *arguments* as untraced references to records, or we declare them as call-by-reference arguments using Modula-3's VAR parameter-passing mode (for which the compiler generates code that automatically dereferences the parameter).

Most Modula-3 programs define data structures in the traced heap, which simplifies interface design and eliminates storage leaks and dangling pointers. However, passing a traced reference outside the domain of a Modula-3 program (such as a foreign language or a device) requires that we register the referent with the collector. Otherwise, the collector may relocate or prematurely deallocate it. For instance, a network buffer that is allocated in the traced heap cannot be passed to a network device, since the pointer to the buffer that is kept in the device is invisible to the collector and hence will not be updated if the buffer is relocated. For this reason, we introduced the notion of a *strong reference*, which registers an object allocated from the traced heap with the collector as temporarily uncollectible and immovable. In this way, an object can be "strong ref'd" before it is passed out and "un-strong ref'd" when and if it is safe to do so.

Finally, we added the CVAR parameter-passing mode to allow Modula-3 procedures to call C procedures more efficiently. In C, call-by-reference is implemented by passing a pointer to the appropriate parameter. Passing a value of NIL as the address of an argument is often used to denote a special case by C programs. It is not possible to preserve such semantics using VAR, the Modula-3 mechanism for call-by-reference, because an argument passed as a VAR parameter must be a designator (and has its address taken automatically).

```
UNSAFE INTERFACE Ccalls
(*   The C declaration for the CallByReference
     procedures is void CBR(int *out); *)

<* EXTERNAL "CBR" *> PROCEDURE
CallByReference1(CVAR out: INTEGER);
(*   CallByReference1 can be called with NIL
     as an argument: CallByReference1(NIL); *)

<* EXTERNAL "CBR" *> PROCEDURE
CallByReference2(out: UNTRACED REF INTEGER);
END Ccalls.
```

**Figure 1:** Declaration using the CVAR call-by-reference mode.

The code fragment in **Figure 1** illustrates how CVAR can be used. CVAR is restricted to EXTERNAL declarations. The CVAR parameter out is similar to a VAR parameter; it differs in that NIL can be explicitly passed as the value for out. If NIL is passed to CallByReference1, the C formal parameter out is bound to NIL. If an INTEGER designator is passed to CallByReference1, CVAR is equivalent to VAR, and out is bound to the address of that designator.

Without CVAR, we would have to declare an external function as CallByReference2. However, using a REF INTEGER does not capture the semantics of the parameter-passing mode for out, and would require the caller to use the unsafe ADDRESS operator. For these reasons, we decided to add the CVAR parameter mode.

### Cross language procedure invocation

Calling out from Modula-3 is facilitated by the existing EXTERNAL pragma, which allows an interface to describe a function written in a foreign language such as C. However, this pragma is legal in both safe and unsafe interfaces, implying that a safe module could import a safe interface that provided direct access to a C function or data structure of arbitrary type. Since the type of the function or data structure may in fact be specified by the C implementation, Modula-3 cannot enforce type safety of safe modules that use EXTERNAL. Consequently, we modified the compiler so that the EXTERNAL pragma can only be used within unsafe interfaces. The programmer is thus forced to assert safety by wrapping a safe interface around the unsafe one containing uses of EXTER-

NAL.

Calling Modula-3 procedures from foreign languages is more complex, as the DEC SRC implementation does not directly support this direction. Rather, the conventional approach is for the programmer to initialize procedure variables that are visible to the foreign language. We use this technique, but also modified the front-end of the compiler to generate C header files for Modula-3 interface files. This way, procedures exported via a Modula-3 interface can be called directly from C using "Module dot method" syntax.

## Specifying alignment

The Modula-3 language does not provide facilities for guiding the compiler's alignment decisions. However, alignment is a real concern, especially when interacting with hardware devices or with code in another language. For example, a C compiler may make certain alignment decisions about data structures, or a memory-mapped device may require a specific memory layout for a data structure. Therefore we need the ability to specify alignment for Modula-3 data structures. In order to give the programmer control over the alignment of data structures, we have added an ALIGNED FOR construct (analogous to BITS FOR) to Modula-3. The type ALIGNED n FOR T, where n is an INTEGER, specifies a subtype of T that has an alignment of n bits, subject to the following constraints:

- n must be a multiple of the type's inherent alignment.
- Aligning T to n bits must not require any padding.

The first restriction preserves natural alignment. The second restriction is a design decision that requires the programmer to explicitly include extra padding when necessary. Together, these restrictions mean that ALIGNED FOR can be used only to boost the alignment of a type.

```
(* The corresponding C declaration:
   struct UdpHeaderC {
      int sport: 16; int dport: 16;
      int len: 16; int check: 16 };
*)

TYPE UdpHeaderM3 = RECORD
   sport: BITS 16 FOR [0..16_ffff];
   dport: BITS 16 FOR [0..16_ffff];
   len : BITS 16 FOR [0..16_ffff];
   check: BITS 16 FOR [0..16_ffff];
END;

TYPE UdpHeaderM3C = ALIGNED 32 FOR RECORD
   sport: BITS 16 FOR [0..16_ffff];
   dport: BITS 16 FOR [0..16_ffff];
   len : BITS 16 FOR [0..16_ffff];
   check: BITS 16 FOR [0..16_ffff];
END;
```

**Figure 2:** Declarations of Modula-3 records using different alignment constraints. Using the ALIGNED FOR constructor enables the programmer to specify the alignment requirements of the data structure.

**Figure 2** illustrates the alignment problems that can arise when interacting with C. One might expect (as we initially did) that a UdpHeaderM3 could be aligned on any 16-bit boundary. In fact, the DEC SRC M3 compiler boosts the alignment of every UdpHeaderM3 to the natural word size of the architecture for efficiency (which is 32-bit for x86 and 64-bit for the Alpha). If the Modula-3 compiler chooses an alignment for UdpHeaderM3 that is more strict than the alignment chosen by the C compiler for UdpHeaderC, then an alignment exception can occur if C code passes a reference to a UdpHeaderC to Modula-3 code. Similarly, if the Modula-3 compiler chooses an alignment for UdpHeaderM3 that is less strict than the alignment chosen by the C compiler for UdpHeaderC, then an alignment exception can occur if Modula-3 code passes a REF UdpHeaderM3 to C code. To avoid these problems, we use ALIGNED FOR as shown in the declaration of the UdpHeaderM3C type of **Figure 2** above.

In addition to adding ALIGNED FOR, we have turned off the automatic alignment boosting that occurs in our M3 compiler. This design decision may seem counter-intuitive, because the programmer must deal with alignment when necessary. However, such alignment issues only arise when dealing with data structures that contain only sub-word-sized types. A programmer who uses such types is already taking performance issues into consideration, and dealing with alignment is not much of an additional burden.

## Implicit exceptions

Modula-3 defines the concept of a *checked run-time error* as a run-time error that must be detected and reported. The specification leaves open how such errors are reflected back to programs, although a statement on page 12 of the Modula-3 book [3] states that an implementation may reflect checked run-time errors as exceptions. Most Modula-3 implementations halt a program whenever a checked run-time error occurs. This is a poor way to reflect errors in an operating system.

The exception model of Modula-3 requires that each procedure explicitly state what exceptions it can raise, although a note on page 29 [3] does imply the existence of *implicit exceptions.* In *SPIN* we require that our Modula-3 implementation reflect all checked run-time errors back to the offending code as implicitly declared exceptions. Such exceptions are raised by every procedure. Unhandled exceptions are subsumed by our model of implicit exceptions, in that the "unhandled exception" exception is another implicit exception, and thus can be caught. If an implicit exception is not caught, the offending thread is stopped at its failure point, and the unhandled exception may be caught by another thread.

```
IMPORT SpinException;
PROCEDURE NilDeref(pointer: REF INTEGER) =
   VAR value : INTEGER; BEGIN
      TRY
      (* dereference a pointer, which could be NIL *)
         value := Pointer^;
```

```
    EXCEPT
    | SpinException.Exception(info) =>
      IF info.code = SpinException.ExceptionCode.Attempt-
ToDereferenceNIL THEN
        (* handle the error *)
        ...
      END;
    END;
END NilDeref;
```

**Figure 3:** An example use of catching a checked run-time error as a language exception.

The code fragment in **Figure 3** illustrates how implicit exceptions are used. In the code, the exception *SpinException.Exception* represents all of the implicit exceptions. In this simple example, the procedure *NilDeref* dereferences a NIL pointer; our implicit exception model allows this error to be caught. Although this example is overly simplified (since the test for a NIL pointer could be done explicitly), it illustrates how implicit exceptions allow programs to explicitly catch checked run-time errors.

*SPIN* also reflects other system faults as implicit exceptions to the offending code. The programmer may catch arithmetic faults (e.g., divide by zero), virtual memory related faults (e.g., protection faults or accessing an invalid address), and unaligned accesses (for processors such as the Alpha). The ability to catch system faults as exceptions enables the construction of more robust systems by guarding against errors. In practice, we have found this capability very useful. For example, TCP services in our HTTP server continued operating even when unrelated UDP code on the packet input path erroneously dereferenced NIL.

## Type-safe casting

Operating system code (such as networking, file system, and system call code) must often interpret "raw bytes" as a "real" type. For good performance, this interpretation should occur without having to copy the raw bytes to a designator, yet all type-safe languages lack an efficient and expressive way to cast data *safely*. Although most languages provide unsafe features that enable casting, we found it necessary for extensibility to write system code using only type-safe language features. Consequently, we developed a type-safe cast operator for Modula-3 called VIEW.

VIEW allows a programmer to reinterpret a piece of memory as a different type. The code in **Figure 4** illustrates how we use VIEW to interpret packets in *SPIN* [2]. Inside the body of the WITH statement, the variable ipHeader is a typed alias for the packet's header. Both the ByteFilter and ViewFilter procedures have the same function, but the former is more obtuse, difficult to maintain, and likely to contain errors. VIEW enables programmers to write code that is simpler to understand and that executes more efficiently than using explicit byte manipulations.

```
IMPORT Ip, Word;
CONST SourceAddr = 16_805f02DE;
(* IP address of www-spin.cs.washington.edu *)
TYPE Packet = ARRAY [0..255] OF Byte;

PROCEDURE ByteFilter(READONLY m: Packet) : BOOLEAN =
(* type-safe, but readable; inefficient on machines
  without byte operations *)
  BEGIN
    RETURN m.packet[9] = Ip.protocol.UDP AND
      Word.And (m.packet[0], 16_F) =
                Ip.version4 AND
      Word.Or(Word.LeftShift(m.packet[15], 24),
      Word.Or(Word.LeftShift(m.packet[14], 16),
      Word.Or(Word.LeftShift(m.packet[13], 8),
      m.packet[12]))) = SourceAddr;
END ByteFilter;

PROCEDURE ViewFilter(READONLY m: Packet) : BOOLEAN =
(* type-safe, readable, and efficient *)
  BEGIN
    WITH ipHeader = VIEW(m, Ip.T) DO
    (* no copying *)
      RETURN ipHeader.protocol = Ip.protocol.UDP AND
        ipHeader.version = Ip.version4 AND
        ipHeader.sourceAddr = SourceAddr;
    END;
END ViewFilter;
```

**Figure 4:** Implementation of packet filters using byte operations and VIEW. The filters accept unfragmented UDP/IP packets with a particular source address. The WITH operator creates an alias to the ipHeader using a VIEW expression.

Our casting operator is correct for two reasons. First, VIEW does not create any illegal representations. Second, for any visible type *T*, a client can allocate its own instances of *T* and set *T*'s fields to any legal value. Therefore, there can be no harm in allowing a client to "create" new instances of *T* via casting instead of allocation. More precisely, VIEW allows a programmer to cast a designator (an expression that denotes a memory location) to another type. We develop a notion of equivalence between designators of different types, such that casting between equivalent designators is type-safe. We can cast a designator from a type $T_1$ to another type $T_2$ so long as the following conditions are met:

- Every bit pattern that represents a legal value in $T_1$ represents a legal value in $T_2$.
- Every bit pattern that represents a legal value in $T_2$ represents a legal value in $T_1$. If the designator is not writable, this condition can be relaxed.
- The alignment of the designator satisfies the alignment required for $T_2$. It may be necessary to check this condition at run-time in order not to reject all designators whose alignment may not match.
- The types $T_1$ and $T_2$ must exist in the same register sets. For example, we disallow casting between floating point and integer designators, because the casting would not preserve sharing.

We define two types to be *representation-equivalent* if they satisfy the first two conditions, which can be determined at compile-time. Trivially, any type *T* is representation-equivalent to itself. For other pairs of types, we must compare the sets of legal values for the types, where *legality* is defined as follows:

- For a base type, the language defines the set of legal values.
- For a record or array type, the set of legal values is the cross-product of the legal values of the types of the fields.
- For a pointer type (which can be an abstract data type or an object type), the set of legal values is distinct from all other pointer types. In other words, given a pointer type *T*, only designators of type *T* can be cast to *T*.

## Summary

We have used the features described in this article to do low-level systems programming with Modula-3 in the context of the SPIN operating system. Our experience with SPIN indicates that the safety of Modula-3, combined with our additions to support low-level systems programming, make the language an ideal choice for systems programming. Not only does Modula-3 prevent most common programming errors by virtue of its type safety, it offers a variety of powerful tools that allow the programmer to tackle a range of systems programming tasks.

## References

[1] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. "Extensibility, Safety and Performance in the SPIN Operating System," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Copper Mountain, CO, December 1995.* http://www.cs.washington.edu/research/projects/spin/www/papers/SOSP95/sosp95.ps

[2] M.E. Fiuczynski and B.N. Bershad. "An Extensible Protocol Architecture for Application-Specific Networking," in *Proceedings of the 1996 Winter USENIX Conference, San Diego, CA, January 1996.* http://www.cs.washington.edu/research/projects/spin/www/papers/Usenix96/extprotarch.ps

[3] G. Nelson, editor. *System Programming in Modula-3*, Prentice Hall, 1991.

[4] E.G. Sirer, M.E. Fiuczynski, P. Pardyak, and B.N. Bershad. "Safe Dynamic Linking in an Extensible Operating System," in *The First Workshop on Compiler Support for Systems Software, February 1996.* http://www.cs.washington.edu/research/projects/spin/www/papers/WCS/domain.ps

[5] E.G. Sirer, S. Savage, P. Pardyak, and B.N. Bershad. "Writing an Operating System in Modula-3," in *The First Workshop on Compiler Support for Systems Software, February 1996.* http://www.cs.washington.edu/research/projects/spin/www/papers/WCS/m3os.ps

[6] W.C. Hsieh, M.E. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B.N. Bershad. "Language Support for Extensible Systems," in *The First Workshop on Compiler Support for Systems Software, February 1996.* http://www.cs.washington.edu/research/projects/spin/www/papers/WCS/language.ps

## *Continuing Thread*
## An Object-Oriented Implementation of Unification Closure

*Allan Heydon, Digital Systems Research Center*

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

*This article describes the design of three object types used in the Juno-2 constraint solver. It illustrates the use of objects, subclassing, method overriding, and partial revelation in Modula-3.*

*Allan Heydon is a member of the research staff at the Digital Systems Research Center.*

### Introduction

Juno-2 is a constraint-based drawing editor implemented in Modula-3 [1]. It uses a double-view interface: the figure in the graphical view is produced by running the program in the textual view. You can edit your figure through either view, and Juno-2 maintains the correspondence between them.

The programming language used in Juno-2's textual view [2] is based on Dijkstra's guarded command language [3], but with provisions for solving constraints. Its value space is the smallest set closed under the formation of ordered pairs and containing real numbers, texts, and the special value NIL. In other words, the data structures of Juno-2 programs are similar to those of LISP.

A constraint in Juno-2 is a predicate on program variables. Solving a constraint is equivalent to finding values for the variables that make the predicate true.

The inclusion of ordered pairs in the value space means that the Juno-2 solver must support constraints on the structure of values. For example, if v is a fixed list of elements, the following command can be used to initialize the local variables a0 and a1 to the first two elements of v:

```
VAR a0, a1, tail IN
  v = (a0, (a1, tail)) ->
    (* code using a0 and a1 goes here *)
END
```

Ordered pairs have the property that if the pairs (a0, b0) and (a1, b1) are asserted to be equal, then the equalities a0 = a1 and b0 = b1 must also hold. (The converse is also true, but there is no need for the constraint solver to infer equalities of pairs.) The Juno-2 solver works by maintaining an equivalence relation on known values and the unknowns being solved for. When two values are asserted to be equal, their equivalence classes are merged. The equivalence relation is said to be *unification closed* if all equalities of pairs have been propagated to equalities on their children.

Juno-2 uses unification [4] to solve constraints on pairs. For example, imagine that the variable v in the above constraint has the following fixed value:

```
v = (0, (1, (2, (3, NIL))))
```

By one unification, the constraint on v would engender the following two equivalences:

```
a0 = 0
(a1, tail) = (1, (2, (3, NIL)))
```

The latter of these two equivalences would then engender the following equivalences:

```
a1 = 1
tail = (2, (3, NIL))
```

Together, these equivalences solve the constraint.

There is an extra wrinkle in the unification process due to value types. At run-time, each equivalence class may have a known type. If two equivalence classes being merged have different known types, then the constraint solver can immediately report failure. If only one class has a known type, that type can be propagated to the other class as a side-effect of unifying the two values. Another failure case that must be detected is a cycle in the closure, such as would be caused by the constraint x = (x, 1).

Juno-2's constraint solver uses three abstract classes to propagate equalities and to perform unification: Equiv.Elt, Egraph.Node, and JunoSolve.Var. An Equiv.Elt represents an element of an equivalence class; the constraint solver uses the *Equiv* interface to maintain the equivalence relation on Juno-2 values. The solver uses instances of Egraph.Node to represent the functional expressions appearing in constraints. Finally, the solver uses instances of JunoSolve.Var to represent knowns (i.e, constants) and unknowns in constraints. As we will see, the three classes are related by subtyping, although the subtyping in one case is revealed in an implementation module rather than an interface.

The design of these classes illustrates the use of subclassing, partial revelation, and method overriding in Modula-3. The rest of this article describes the three classes and their interfaces. There is also a short section on the implementation.

## Design

An equivalence graph, or *Egraph*, is a graph together with an equivalence relation [5]. The graph can be used to represent functional expressions, and the equivalence relation can be used to represent known equalities between nodes of the graph. For example, **Figure 1** shows the Egraph for the constraint CAR(x) = 2 * y. In this graph, the names x0 and x1 have been introduced for the first (CAR) and second (CDR) elements of the pair x, respectively.

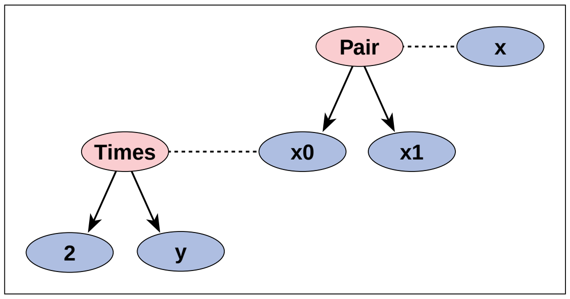


**Figure 1:** The Egraph for the constraint CAR(x) = 2 * y. In this graph, the dashed lines denote nodes in the same equivalence class.

The basis for an Egraph is an equivalence relation on its nodes. Juno-2 uses the following *Equiv* interface to maintain the equivalence relation:

```
INTERFACE Equiv;
EXCEPTION Forbidden;
TYPE
  Elt <: Public;
  Public = OBJECT METHODS
    init(): Elt;
    find(): Elt;
    union(y: Elt): Elt RAISES {Forbidden};
  END;
```

For those readers unfamiliar with Modula-3, these declarations introduce the types Elt and Public. The syntax Elt <: Public declares the type Elt to be a partially opaque subtype of the type Public. Public is declared to have no data fields and three methods. Here is the rest of the interface.

> *An Equiv.Elt is an element of an equivalence relation. The statement NEW(Equiv.Elt).init() produces a new element in an equivalence class by itself.*
>
> *The call x.find() returns the distinguished representative, or root, of x's equivalence class.*
>
> *The call x.union(y) combines the equivalence classes represented by x and y, and returns the representative of the new class. After the call x.union(y), x.find() = y.find(). It is a checked run-time error for either x or y not to be the root of its equivalence class. The result of x.union(y) is guaranteed to be either x or y.*
>
> *The default union method never raises the Forbidden exception, but it may be useful for subtypes overriding that method to raise Forbidden in the event that the operation is deemed illegal by the subtype.*

```
END Equiv.
```

The expressions in an Egraph are function applications. For simplicity, Juno-2 uses a more concrete representation of function applications than the abstract one depicted in **Figure 1**: the function application $f(a_1, \ldots, a_n)$ is represented by a linked list of length $n+1$ in which the function name *f* is the first element of the list, and the arguments are the remaining elements. For example, **Figure 2** shows the representation of the pair expression (x, 2+y).

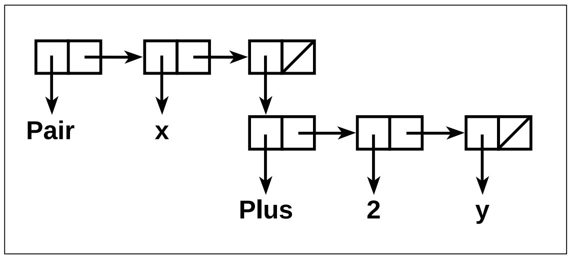


**Figure 2:** The Egraph representation of the expression (x, 2+y).

The *Egraph* interface reveals that an Egraph node is actually a subtype of an Equiv.Elt:

```
INTERFACE Egraph;
IMPORT Equiv;
TYPE
```

```
  Node <: Public;
  Public = Equiv.Elt OBJECT
    head, tail: Node := NIL;
  METHODS
    init(): Node;
  END;
```

*An Egraph.Node is a node of an oriented, directed graph in which every node has out-degree at most 2 and on which there is an equivalence relation.*

*The statement NEW(Egraph.Node, head := a, tail := b).init() evaluates to a new node in its own equivalence class with children initialized to the values a and b. The head and tail fields should not be written after initialization.*

```
END Egraph.
```

The third class of interest is used to represent constants and variables in constraints. These are both represented by the partially opaque Var type in the *JunoSolve* interface. In this interface, RTVal.T is the type of a Juno-2 run-time value.

```
INTERFACE JunoSolve;
IMPORT RTVal;
TYPE
  Var <: Public;
  Public = Private OBJECT
    known: BOOLEAN;
    val: RTVal.T;
  END;
  Private <: ROOT;
```

*If x is of type Var, then x.known indicates if the variable denoted by x is known (i.e., fixed) or unknown. If x.known, then x.val is a legal Juno value. If NOT x.known but x.val # NIL, then x.val is a hint for the initial value of x.*

```
PROCEDURE New(known := FALSE; val: RTVal.T := NIL): Var;
```

*Return a new, valid Var with the given field values.*

The *JunoSolve* interface also includes procedures for creating new constraints and a procedure P for solving a constraint system, but these procedures are irrelevant to the current discussion.

Notice in the above declarations that both the prefix and the suffix of JunoSolve.Var are opaque. The prefix is opaque because the type Private is declared to be an opaque subtype of the root object type ROOT. The suffix is opaque because Var is declared to be an opaque subtype of the type Public. The *JunoSolve* implementation reveals that a JunoSolve.Var is actually a subtype of an Egraph.Node.

```
MODULE JunoSolve;
IMPORT Equiv, Egraph, RTVal, ...;

REVEAL
  Private = Egraph.Node BRANDED "JS.Private" OBJECT
    pair: EC;
  OVERRIDES
    union := Union;
  END;
  Var = Public BRANDED "JunoSolve.Var" OBJECT
  type: Type;
```

*Other fields private to the implementation declared here...*

```
  END;
TYPE
  EC = Private;
```

*The type "EC" represents an equivalence class with an extra field to record if the class contains a pair expression. We declare the name "EC" simply to serve as a more meaningful synonym for the type "Private" in the implementation.*

```
Type = { Any, Pair, Num, Text, Null };
```

*There is a flat partial order on types, with "Type.Any" as bottom.*

The head and tail fields of a JunoSolve.Var are always NIL, but by virtue of being an Equiv.Elt, a JunoSolve.Var is also part of the equivalence relation.

When a node is the representative of its equivalence class and denotes an ordered pair, its pair field is non-NIL and points to an Egraph list of length 3 in which the first element of the list is the node denoting the *Pair* function and the next two elements are the pair's CAR and CDR.

Notice that JunoSolve.Private is revealed to override the Equiv.Elt.union method with the local procedure JunoSolve.Union. Its implementation is sketched below.

## Implementation

Space limits do not permit a complete description of the unification closure implementation. Here, we summarize three of its aspects.

- The implementation of the JunoSolve.Union procedure first calls the Egraph.Node.union method. It then performs extra work required by the Juno-2 solver. For example, it compares the type fields of the two nodes being merged. If they have incompatible types, it raises the exception Equiv.Forbidden. If one is of type JunoSolve.Type.Any and the other is not, the known type is propagated to the representative of the new equivalence class.
- Unification closure is implemented by keeping a queue of pending unifications. First, the constraints are processed and each expression is added to the Egraph. The left- and right-hand sides of each equality constraint are added to the queue. Then, so long as the queue is non-empty, the first two elements in the queue are removed from the queue and unified using the union method. The implementation of the JunoSolve.Union procedure has the side-effect of adding elements to the queue whenever two pairs in different equivalence classes are unified. The time complexity of our implementation is $O(n)$, where $n$ is the size of the Egraph.
- The *Equiv* interface is implemented using the well-known Union-Find algorithm. To evaluate which implementation would be best for Juno-2, we wrote an animation for visualizing the performance of several different Union-Find implementations. By instrumenting Juno-2 to print the sequence of union and find operations it executed, we discovered that Juno-2's solver tended to do many more *find* operations than *union* operations.

We therefore chose to use the QuickFind implementation, in which each equivalence class is represented by a tree of depth at most 1. Each element has a pointer to the representative element of its equivalence class. The *find* operation thus takes constant time. The *union* operation works by making all of the elements of the smaller class point to the root of the larger class. The implementation is straightforward.

## Conclusions

The use of multiple object types, subclassing, and method overriding have led to a modular, compact implementation of unification closure in the Juno-2 constraint solver. Modula-3's opaque types and partial revelation facility allowed us to write interfaces that were clear, simple and reusable.

## Acknowledgments

The work described in this article is joint with Greg Nelson. Thanks to Greg, Bill Weihl, and Marc Najork for their comments on earlier drafts of this article.

## References

[1] Allan Heydon and Greg Nelson. The Juno-2 Constraint-Based Drawing Editor, SRC Research Report 131a, December, 1994. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-131a.html

[2] Greg Nelson and Allan Heydon. Juno-2 Language Definition, SRC Technical Note 1997-009, June 30, 1997. http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1997-009.html

[3] Edsger W. Dijkstra. A Discipline of Programming, Prentice-Hall, Inc., 1976.

[4] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle," in *Journal of the ACM*, Vol. 12, No. 1, pgs 23-41, January, 1965.

[5] Greg Nelson. Techniques for Program Verification, Technical Report CSL-81-10, Xerox Palo Alto Research Center, June, 1981. http://www.parc.xerox.com/parc-go.html

## *Modula-3 in Academia* Department of Computer Science, III Aachen University of Technology, Germany

*Peter Klein, Lehrstuhl für Informatik III*

The Department of Computer Science III is one of the thirteen departments comprising the computer science group at the Aachen University of Technology. Started by Professor M. Nagl, the main research topics of the department are methods, languages, and tools for software engineering.

Like many other computer science departments, we favored Modula-2 in the eighties as the main programming language for teaching and project implementation. In the early nineties, our dissatisfaction with Modula-2 grew because of its missing support for modern programming concepts like objects, genericity, garbage collection, and exception handling. On the practical side, it was also becoming increasingly difficult to find a Modula-2 environment that was suitable for our implementations (approaching 500K lines of code at that time.)

It took us some time to reach agreement on a replacement for Modula-2. Although Modula-3 seemed a natural choice, our experience with Modula-2 suggested using a more well-known programming language. As such, different subprojects started out in C, C++, and Modula-3, which allowed us to recognize that the compactness and clarity of Modula-3 allowed us to produce more robust and reusable code. This factor proved to be an essential point in our academic setting: with a high fluctuation of rather inexperienced programmers, a major part of the implementation of our systems are done by student workers or in the course of diploma theses.

## Project Courses

Since 1993, we have used Modula-3 successfully in a series of project courses for graduate students. These projects introduce different aspects of software development including project planning, supervision, design, and cooperative implementation of small but usable software systems. Central ideas of software design and object-oriented implementation are presented and discussed together with the concepts of Modula-3, which is also used as the implementation language. The success of these projects has been a strong argument for Modula-3 as a teaching and development language since it clearly enables a continually changing group of students to learn the language, its concepts, and its libraries as well as to produce sensible results collaboratively and within a short time.

Some of the project courses resulted in usable software, including *m2tom3* and *readthis.*

### *m2tom3*: migrating Modula-2 programs

The *m2tom3* system helps with migration of Modula-

2 programs to Modula-3. It consists of two parts:

- A conversion program that creates a Modula-3 source with the same semantics as a Modula-2 source while retaining the original's look and feel as much as possible.
- A base library that tries to emulate the Modula-2 standard library using the Modula-3 standard library. Written to support our own transition to Modula-3, we have successfully used it to convert most of our software. Noticeably, the next release of the *PROGRES* system, an implementation of a specification language based on graph grammars, will be Modula-3-based.

### *readthis*: network access to hierarchies

The *readthis* system is a simple, Network Object-based server for accessing hierarchies of text files. Demo applications contained in the distribution are a very simple news/blackboard system called *mininews* and a problem management system called *bugtrack* (comparable to *gnats*/*tkGnats*). For more information on *readthis* visit http://www-i3.informatik.rwth-aachen.de/research/readthis/. *readthis* can be freely download from ftp://ftp-i3.informatik.rwth-aachen.de/pub/Modula-3-Contrib/readthis.

We have also been engaging in a number of longer-term research projects, which are described below.

## Research Projects:

### *GRAS*: graph-oriented database system

*GRAS* is a database system that supports application domains such as software engineering or computer integrated manufacturing. These systems are usually highly interactive so they deal with rather complex object structures. For the realization of these systems, a nonstandard database system is needed to efficiently handle different types of coarse- and fine-grained objects (documents and paragraphs), hierarchical and non-hierarchical relations between objects (composition links and cross references), and attributes of rather different sizes (chapter numbers and bitmaps). Furthermore, this database system should support incremental computation of derived data, undo/redo of data modifications, error recovery from system crashes, and version control mechanisms.

*GRAS* is a graph-oriented database system with a multi-client/multi-server architecture. From the first prototype in 1985, gradually improving versions have been used in different software engineering projects. Currently, a version automatically derived from a Modula-2 implementation using *m2tom3* is shipped in binary form with the *PROGRES* release. A new implementation called *GRAS-3* written directly in Modula-3 will be released around the end of 1997.

### *TXL-3*: transformational programming

*TXL-3* is an implementation of the *TXL* language in Modula-3. *TXL* is a programming language for doing transformational programming, originally designed by James R. Cordy, Queen's University at Kingston, Canada. The basic paradigm of *TXL* involves transforming input to output using a set of transformation rules that describe by example how different parts of the input are to be changed into output. *TXL* has been used to transform C, C++, Cobol, and Modula-3 programs. Each *TXL* program defines its own context-free grammar according to how the input is to be broken into parts, and rules are constrained to preserve grammatical structure in order to guarantee a well-formed result.

### *adt*: design-centered development

The *adt* project is building a design-centered development environment for both large-scale and small-scale programming.

The core of the adt prototype is an editor for a software architecture design language. Existing Modula-3 source code can be analyzed to visualize its architectural structure, and code skeletons can be generated from an architecture description.

A major goal of *adt* is to assist the programmer by providing him with an architectural view of his system without requiring the use of special tools. So, we made it quite easy to adapt *adt* not only to use it with any editor or compiler, but also to integrate it with other tools for revision control, browsing, e-mailing/conferencing, problem reporting, or any other tools needed in everyday work. In order to keep the participating documents consistent while allowing different tools to modify them, *adt* supports incremental updates between an architecture description and source code, so that changes in the architecture can be propagated into existing Modula-3 sources and the architectural plan can be mapped to source modifications.

An early demo of adt was built for a project course in 1994. A complete new implementation making use of the GRAS-3 database and TXL-3 is currently under construction and will, apart from the architecture language mentioned above, also feature extensions for concurrent and distributed systems and interaction diagrams comparable to the collaboration diagrams in Unified Modeling Language. A release is planned for early next year.

*Aachen University of Technology, in Aachen, Germany has been using Modula-3 successfully for a number of academic and research projects since 1993. Peter Klein, a member of the CS Department III at Aachen University of Technology, reports on some of their experiences.*

*m2tom3 is freely available at* ftp://ftp-i3.informatik.rwth-aachen.de/pub/Modula-3-Contrib/m2tom3/

*For more information about GRAS, visit:* http://www-i3.informatik.rwth-aachen.de/research/gras/

*For more information about PROGRES, visit:* http://www-i3.informatik.rwth-aachen.de/research/progres/

*TXL-3 is freely available at URL:* ftp://ftp-i3.informatik.rwth-aachen.de/pub/Modula-3-Contrib/txl-3/

# Are your distributed applications bulletproof?

They can be with

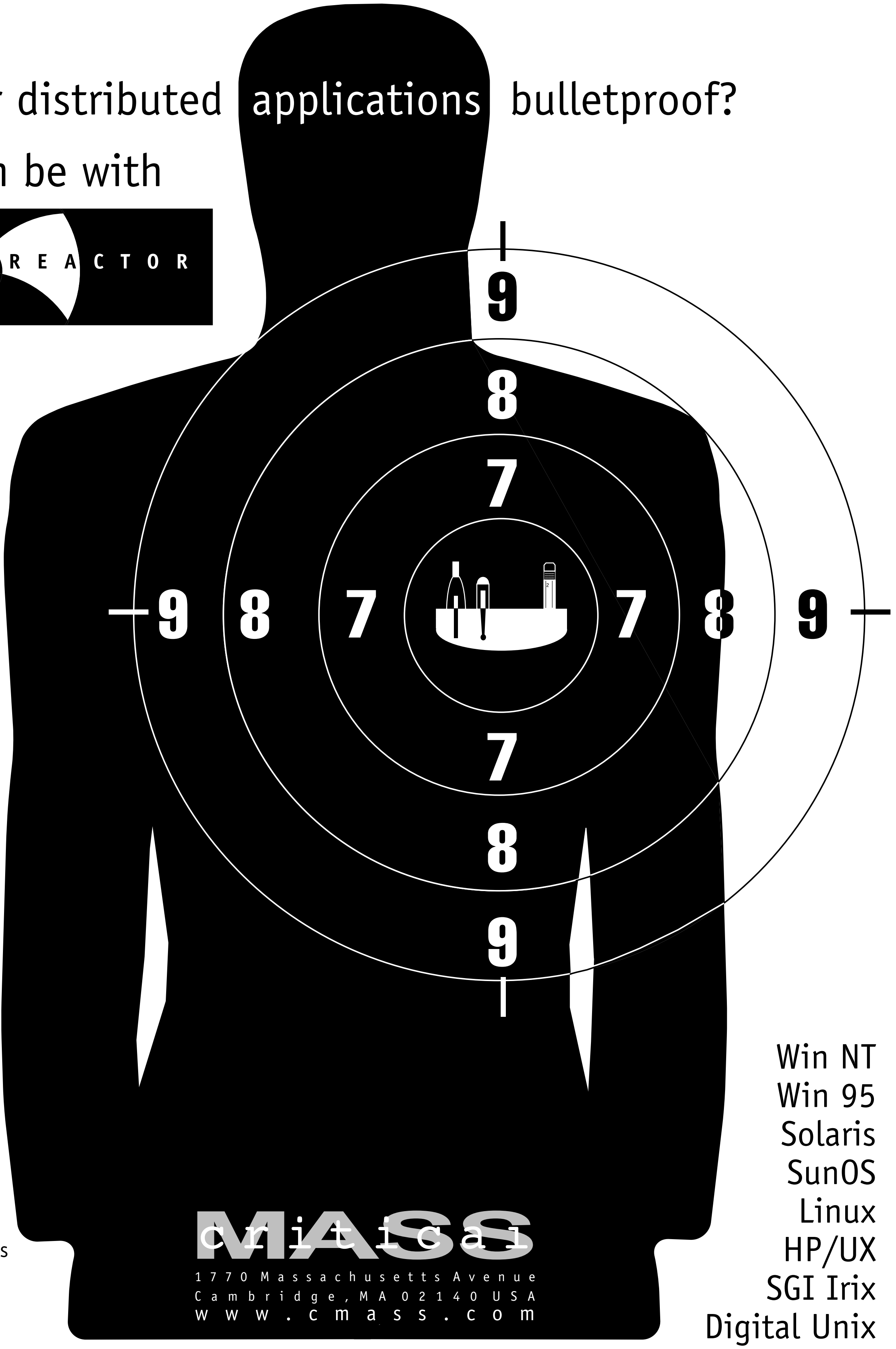
REACTOR

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 www access.



critical MASS

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

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