Language Support for Extensible Operating Systems

Wilson C. Hsieh, Marc E. Fiuczynski, Charles Garrett,
Stefan Savage, David Becker, and Brian N. Bershad

Abstract

We have identified three areas where language support for operating system extensibility is important: performance, safety, and expressive power. First, an extension language should support pointer-safe casting to avoid unnecessary copying of data. Second, an extension language must allow a caller to isolate untrusted code. Finally, an extension language should provide a vocabulary for describing interfaces, modules, and procedures in a first-class fashion. We present specific examples of these needs, and describe the changes to Modula-3 that we have made for our work in the *SPIN* operating system.

1 Introduction

SPIN is an extensible operating system that allows untrusted applications to extend system services by dynamically linking *extension* code into its kernel [2]. These extensions directly access system services with procedure calls, and system resources via loads and stores. As a result, extensions do not incur costly address space switches when they interact with the kernel.

SPIN relies on language facilities to protect the system from extensions (and extensions from each other). Extensions are written using the safe subset of Modula-3 [6], which enforces type safety. Relying on the language to enforce safe extension behavior is more efficient than relying on hardware mechanisms [1], because much of the protection overhead is paid at compile-time or link-time.

We have added new features to the safe subset of Modula-3, because it does not satisfy our requirements for performance and expressive power. We could satisfy these requirements by using unsafe features of Modula-3, but those features are unacceptable; unsafe features would allow extensions to compromise the safety of *SPIN*. Our new features add these abilities:

- *Pointer-safe casting.* **VIEW** allows data created outside of the language, such as packets coming from the network, disk buffer blocks, and system call arguments, to be given Modula-3 types.
- *Isolation of untrusted code.* **EPHEMERAL** labels procedures that may be terminated at any time. Implicit exceptions allow all runtime errors to be caught. These features allows a caller to isolate itself from untrusted code with which it interacts.
- *First-class code structures.* **INTERFACE_UNIT** and **MODULE_UNIT** allow code to name interfaces and modules, and to pass these names to the *SPIN* extension services. **PROCANY** is a generic type that allows us to manipulate generic procedures in *SPIN*.

Contact information: {whsieh,mef,garrett,savage,becker,bershad}@cs.washington.edu. Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195

The following sections discusses these three new safe abilities. Section 2 describes language support for pointer-safe casting. Section 3 describes support for isolating untrusted code. Section 4 describes support for first-class code structures. Section 5 describes some related language work. Section 6 briefly summarizes our contributions.

2 Pointer-Safe Casting

Operating system code must often interpret an arbitrary “bag of bytes” as a “real” type. For example, a network packet arrives from the hardware as an uninterpreted array of bytes, but networking code must interpret the different fields in a packet as specific data types. For good performance, this interpretation should occur “in place”; that is, an operating system must avoid copying as much as possible, in order to deliver high performance networking [3].

Unsafe languages, such as C, provide the ability to “cast” data from one type to another. The safe subset of Modula-3, however, does not provide any means to cast data. Such a facility is inherently type-unsafe, because it allows a user to subvert the protection afforded by the type system. Unfortunately, we require a casting facility in order to avoid copying data unnecessarily.

Our key observation is that *SPIN* does not require full type safety to maintain system integrity. Extensions need only be “pointer-safe,” which means that pointers are not forgeable. As a result, we can allow casting that does not violate pointer safety. Although pointer-safe casting allows applications to subvert the strict type system, it does not give applications the power to forge any names. Therefore, it does not violate the level of safety that *SPIN* requires in extension code.

We provide pointer-safe casting with a new Modula-3 operator called **VIEW**. **VIEW** can be used to marshal and unmarshal data. **VIEW** can be used to cast data structures to and from an array of bytes, which is used to represent generic data. **VIEW** takes two arguments: a variable, and a destination type. The **VIEW** expression is writable if and only if the variable is writable. There are several restrictions on the type of the variable, and on the destination type. These restrictions depend on the use to which **VIEW** is put:

- To marshal data, **VIEW** can be used to cast a data structure into another type. In the case of networking, the destination type would be an array of bytes. If the data structure is writable, then it cannot contain any pointers. If the data structure is not writable, then it may contain pointers.

- To unmarshal data, **VIEW** can be used to cast a data structure into another type. In the case of networking, the data structure would be an array of bytes, and the destination type would be a packet type. This type cannot contain any pointers.

These restrictions on pointer types prevent the forgery of pointers. Similar restrictions apply to types whose representations have illegal bit patterns (for example, the enumeration [0..5]). Restrictions on these types prevent errors due to illegal bit patterns.

The return value of **VIEW** is the “same” as its first argument, but has a different apparent type. In addition, the size of the input must be at least the size of the target type, which prevents a user from accessing data that is past the end of the input. If the bounds of the array are known statically, the size check is performed at compile-time; otherwise, the check is performed at runtime. Finally, the alignment of the input should agree with that of the target type; if alignment compatability is not known at compile-time, a check is performed at runtime.

The code in Figure 1 illustrates how we use **VIEW** to interpret packets in our networking code [5]. We rely on Modula-3’s **WITH** statement to create an alias, which allows a **VIEW**’ed array to be named without a copy. Inside the body of the **WITH** statement, the variable etherHeader is a well-typed alias for the packet’s

```
MODULE Ether;
IMPORT Mbuf, IP, Arp;
...
PROCEDURE Input(READONLY packet: Mbuf.T) =
  BEGIN
    ...
    (* view Ethernet header using Modula–3 type – no copy. *)
    WITH etherHeader = VIEW(packet.payload, Ether.T) DO
      (* do some processing on the ether packet *)
      END;
    END;
  END Input;
...
BEGIN
END Ether.
```

Figure 1: Use of **VIEW** in networking code

payload. **VIEW** provides the ability to safely interpret external data structures as Modula-3 types, but without having to copy them.

3 Isolating Code Behind Trust Boundaries

In *SPIN*, a caller can invoke untrusted code through a procedure call, which is a convenient and efficient control transfer abstraction. Although modern programming languages provide abstraction mechanisms to hide data behind trust boundaries, they do not provide mechanisms to isolate the execution of code behind trust boundaries. When untrusted code is invoked, two aspects of control isolation become important:

- *Terminating the callee.* A caller sometimes needs to terminate untrusted code that it calls. For example, extensions may be installed as interrupt handlers; if a handler runs for too long, it must be terminated.
- *Isolating the caller.* A caller needs to handle all errors in code that it calls. For example, if an untrusted callee divides by zero, the caller needs the ability to recover from the fault.

The following sections discuss how we achieve these isolation properties within the context of Modula-3.

3.1 Terminating Untrusted Code

When a caller calls untrusted code, it may need to terminate that code at arbitrary points. *SPIN* is a preemptive operating system, so extensions that run forever will not freeze the system. However, non-preemptible parts of the system, such as interrupt handlers and the scheduler, may also call extension code. When extension code runs for too long in such parts of the system, the extension code must be terminated. Because a terminated extension has no means of "cleaning up" after it is killed, arbitrary termination can leave critical data structures in an unknown condition. For example, if termination occurs during memory allocation, the global state of the memory allocator could be left inconsistent.

We solve this problem by classifying code as killable or non-killable. We make this distinction at the procedure level. The invocation of a killable procedure can be killed at any time; the invocation of a non-killable procedure cannot. Killable procedures can only call other killable procedures. Otherwise, the invocation of a killable procedure would not be truly killable. System-provided routines that modify global state, such as the **NEW** memory allocation procedure, are labeled as non-killable.

Our solution in Modula-3 is to provide the **EPHEMERAL** procedure tag. Procedures that are killable include the **EPHEMERAL** tag in their signature. Procedures that cannot be arbitrarily terminated do not use the **EPHEMERAL** tag. When *SPIN* cannot allow preemption, only extensions with a **EPHEMERAL** signature will be executed. For example, *SPIN* will only install **EPHEMERAL** procedures as interrupt handlers.

```
EPHEMERAL PROCEDURE KillMe() =
  BEGIN
    WHILE TRUE DO
      (* infinite loop *)
    END;
  END KillMe;

PROCEDURE CannotKill() =
  VAR NewMemory : REF INTEGER;
  BEGIN
    (* NEW is not EPHEMERAL, so CannotKill cannot be EPHEMERAL *)
    NewMemory = NEW (REF INTEGER);
  END CannotKill;
```

The above code fragment illustrates the distinction between **EPHEMERAL** and non-**EPHEMERAL** procedures. The procedure KillMe is a legal **EPHEMERAL** procedure. As a result, a caller can freely terminate a call to KillMe. Since KillMe can be killed, *SPIN* will allow it to be installed as an interrupt handler. On the other hand, the procedure CannotKill cannot be **EPHEMERAL**, because it calls the memory allocation routine **NEW**, which is not **EPHEMERAL**. Therefore, a caller cannot terminate a call to CannotKill, and *SPIN* will not allow it to be installed as an interrupt handler.

EPHEMERAL does not prevent an application from harming itself. For example, an application could synthesize its own locks, and acquire that lock within **EPHEMERAL** code. If the **EPHEMERAL** code is killed while the lock is held, some other code within that application could deadlock on that lock. We do not expect this to be a severe problem: **EPHEMERAL** serves as a reminder that very little should be done within an **EPHEMERAL** procedure. More importantly, **EPHEMERAL** prevents applications from harming other applications, which is our primary safety goal.

3.2 Isolating Errors In Untrusted Code

When a caller interacts with untrusted code, the caller must be able to handle failures, such as division by zero, in the untrusted code. Modula-3 defines the concept of a "checked runtime error" as those runtime errors that must be detected and reported. The specification leaves open how such errors are reflected back to programs. A typical Modula-3 implementation will halt a program whenever a checked runtime error occurs, which is a poor way to reflect errors in an operating system.

In order to prevent an error in a callee from crashing a caller, we have given runtime errors a representation in the language. All checked runtime errors are reflected back to the offending code as implicitly declared exceptions.[1] That is, these exceptions are all implicitly 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 exception is reflected to any thread that joins with the offender.

```
PROCEDURE Dereference(Pointer: REF INTEGER) =
  VAR info: SpinException.ExceptionInfo;
      value: INTEGER;
  BEGIN
    TRY
      (* dereference a pointer, which could be NIL *)
      value := Pointer^;
    EXCEPT
      SpinException.Exception(info) =>
        IF info.code = SpinException.ExceptionCode.AttemptToDereferenceNIL THEN
          (* handle the error *)
          ...
        END;
    END;
  END Dereference;
```

The above fragment of code 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 Dereference 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 runtime errors.

4 First-Class Code Structures

We want to enable extensions to control linkage. In *SPIN*, user code must have the ability to dynamically link code into the system. As a result, we need to express linkage in the language, which requires that code structures have types and names. In Modula-3 the code structures that are relevant to linkage in *SPIN* are interfaces, modules, and procedures. Currently, Modula-3 does not provide a safe facility for naming the classes of interfaces, modules, and procedures. Moreover, interfaces and modules cannot be treated as values within a program.

We have modified Modula-3 to make code structures first-class. Because the names for code structures are used to control dynamic linkage, they must be unforgeable. In other words, a malicious application should not be able create a name for code to which it does not have access.

We allow programs to name modules and interfaces using two opaque types, RTCode.InterfaceType and RTCode.ModuleType, and two builtin procedures, **INTERFACE_UNIT** and **MODULE_UNIT**. The procedure **INTERFACE_UNIT** takes an interface as an argument, and returns the RTCode.InterfaceType of an interface. An instance of RTCode.InterfaceType uniquely identifies a particular implementation of that interface at runtime.

[1] The exception model of Modula-3 requires that each procedure explicitly state what exceptions it can raise, although a note on page 29 of the Modula-3 book [6] does imply the existence of an implicit exception. A statement on page 12 states that an implementation may reflect checked runtime errors as exceptions. *SPIN* requires that it does so.

The procedure **MODULE_UNIT** returns the RTCode.ModuleType of the calling module, which is used to give a module an unforgeable runtime identifier (**MODULE_UNIT** can only be used inside a module, not inside an interface). The identifier can be used by the module to assert its authority over the implementation of a class of resources, such as types and procedures, which may have exposed interfaces but concealed implementations.

We have also added a new generic procedure type called **PROCANY** to Modula-3, because we have found it necessary to write interfaces that operate on generic procedures. One example of such an interface is that of the *SPIN* dispatcher [7]. The dispatcher is responsible for binding events and their handlers, both of which are represented as procedures. As a result, the dispatcher must be able to accept generic procedures as arguments.

```
INTERFACE RTCode;
(* declare the names as hidden types *)
TYPE ModuleType <: REFANY;
TYPE InterfaceType <: REFANY;
(* returns true iff m exports a function declared in i *)
PROCEDURE ModuleExportsInterface(m: ModuleType; i: InterfaceType): BOOLEAN;
(* return true iff m defines procedure p *)
PROCEDURE ModuleImplementsProcedure(m: ModuleType; p_ref: PROCANY): BOOLEAN;
END RTCode.
```

The above code fragment contains part of the RTCode interface, which demonstrates the use of our new types. This interface contains procedures that can be used to ask questions about the relationships between various code structures. Such questions are necessary to permit linkage operations to be authorized based on these relationships. Without the expressiveness provided by our new types, this interface could not be written in Modula-3.

5 Related Work

The Java [8] language, although it has a syntax that resembles C and C++, is very similar in spirit to Modula-3. Java also restricts casting to be pointer-safe in order to ensure safety.

The Modula-2+epsilon [4] language, an extension to Modula-2+, provided a casting facility similar to **VIEW**. The language uses the notion of "representation-completeness": a type is representation-complete if its representation has no illegal bit patterns.

6 Conclusions

We have described several language services that we have added to Modula-3 in order to support the *SPIN* extensible operating system. Although the changes that we made are specific to Modula-3 and *SPIN*, the issues that forced the changes apply in general to languages for operating system extensions.

Acknowledgments

We thank the rest of the *SPIN* group, without whom this paper could not have been written.

This research was sponsored by the Advanced Research Projects Agency, the National Science Foundation (Grants no. CDA-9123308 and CCR-9200832) and by an equipment grant from Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship. Fiuczynski was partially supported by a National Science Foundation GEE Fellowship.

References

[1] B.N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E.G. Sirer. "Protection is a Software Issue". In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 62–65, Orcas Island, WA, May 4–5, 1995.

[2] 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 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, December 3–6, 1995.

[3] D. D. Clark and D. L. Tennenhouse. "Architectural Considerations for a New Generation of Protocols". In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications 1990*, September 1990.

[4] Digital Equipment Corporation. *"Modula-2+Epsilon Language Specification"*, March 20, 1991.

[5] M. Fiuczynski and B.N. Bershad. "An Extensible Protocol Architecture for Application-Specific Networking". In *Proceedings of 1996 Winter USENIX*, San Diego, CA, January 22-26, 1996.

[6] G. Nelson, editor. *"Systems Programming with Modula-3"*. Series in Innovative Technology. Prentice Hall, Englewood Cliffs, NJ, 1991.

[7] P. Pardyak, S. Savage, and B.N. Bershad. "Language and Runtime Support for the Safe Dynamic Interposition of System Code", November 1996.

[8] Sun Microsystems Computer Corporation. *"The Java Language Specification"*, version 1.0 beta edition, October 30, 1995.