

The Interaction of Access Control and Object Orientation in Extensible Systems

Wilson C. Hsieh
Dept. of Computer Science
Univ. of Utah
wilson@cs.utah.edu

Przemyslaw Pardyak
Dept. of Computer Science and Engineering
Univ. of Washington
pardy@cs.washington.edu

Marc E. Fiuczynski
Dept. of Computer Science and Engineering
Univ. of Washington
mef@cs.washington.edu

Charles Garrett
Reservoir Labs, Inc.
garrett@reservoir.com

Brian N. Bershad
Dept. of Computer Science and Engineering
Univ. of Washington
bershad@cs.washington.edu

Abstract

In this paper we describe how object-oriented language design interacts with access control in extensible systems, based on our experience in building the SPIN extensible operating system. Several modern extensible systems, such as Java-enabled web browsers and SPIN, use object-oriented languages for extensibility. These systems allow extension programs written in their languages (Java and Modula-3, respectively) to be linked in at run time. This paper presents a case study of the object-oriented language design issues that we encountered in building SPIN. First, we describe how access control in SPIN is affected by the language design choices made in Modula-3, and how we changed Modula-3 to satisfy our access control requirements. Second, we compare the access control mechanisms we chose in SPIN, which are mostly link-time, with those in Java, which are mostly compile-time.

Keywords: extensible systems, object orientation, classes, access control, type safety.

1: Introduction

Java-enabled Web browsers [13] and the *SPIN* operating system [3] are two examples of modern extensible systems, which allow untrusted users to directly link extension code into them at run time. Both systems use object-oriented extension languages: Java [1] and Modula-3 [21], respectively. The systems rely on the type safety of their extension languages to guarantee memory safety [2], and they depend on access control to protect code and data from unprivileged users. In this paper we describe what we have learned about using an object-oriented language, Modula-3, as the extension language in *SPIN*: that access control over name spaces requires access control over class operations.

Access control mechanisms allow the system to decide what principals may access which resources in a system. In an extensible system, trusted clients must be granted access to certain system resources, whereas untrusted clients must be denied access to them. For example, a user should have access to the cryptographic key in his Web browser, but an untrusted applet should not. In this paper we focus on the effects of object-oriented language design on access control, and on the mechanisms that can be used to control access to classes.

In this paper we make the following contributions:

- We describe how object-oriented language design interacts with access control in extensible systems, based on our experience with *SPIN*. *SPIN* is an extensible system that uses name space control to encapsulate extensions. We explain how language design choices in Modula-3 affected our ability to provide access control in *SPIN*, and how we had to add access controls over classes in Modula-3. These lessons show that if the language is not explicitly designed for an extensible system, the use of that language may lead to access control problems over classes.
- We describe how we modified Modula-3 for *SPIN* in order to deal with the access control problems that we encountered. These modifications were subject to the real-life constraint that we had to remain as backward-compatible as possible with the Modula-3 libraries that we use.
- We provide examples of how these problems are dealt with in Java, a language that was explicitly designed as an extension language. We compare our solutions in *SPIN* to those in Java, and also compare the different language and system design choices that were made in the two systems. This comparison is performed in the context of the *SPIN* protection model, which uses access control over name spaces to protect resources.

Section 2 provides some background on extensible systems in general, and *SPIN* and Java in particular. Section 3 compares the access control mechanisms over classes that we provide in *SPIN* to the mechanisms provided in Java. Finally, Section 4 describes related work and summarizes our conclusions.

2: Background

Type safety has been used as the basis for protection in many extensible systems, including Cedar [25], *SPIN* [3], Thor [14], Inferno [9], Juice [11], and Java [12]. In these systems, extension code can be directly loaded and linked with the system, which enables low-overhead access to the system's data and code. That is, extensions can invoke services through direct procedure calls (instead of through system calls), and can directly access data. We use two such systems, Java and *SPIN*, as examples in the rest of the paper.

2.1: Java

Java has become widely used for writing portable Web applications. At a high level, Java and *SPIN* are similar in design. Systems that support Java allow users to download, link, and run untrusted extensions that are called *applets* or *servlets*. Java programs are shipped as Java Virtual Machine bytecodes [13], and can be written in any language that compiles

to those bytecodes, including the Java programming language. A bytecode verifier is used to verify the type safety of applets at load time.

Java is a pure class-based, object-oriented language. The class is both the unit of program structure, as well as the unit of linkage. Linkage is accomplished during loading, which is performed by a *class loader*. The class loader loads a class, and the Java Virtual Machine links the class into the system.

Because extension code is downloaded as high-level bytecodes, some semantic checks can occur when code is linked. For example, bytecodes are type-checked when they are linked into the system. As a result, the distinction between “compile-time” and “link-time” is not a matter of the point in time at which the checks occur: some of the compile-time checks on the Java language can (and do) occur at link-time. The primary distinction between our use of “compile-time” and “link-time” checks is that the latter can depend on the identity of the principal that is requesting linkage: compile-time restrictions cannot.

It should be noted that package-level access control is weak in Java. Any class can be declared to be in any package (except for pre-defined system packages, such as `java.lang.*`). Therefore, package-level access controls (through the use of **public** and **protected**) do not provide any real protection.

Wallach et al. [27] have explored several protection models within Java. One of the models that they examine (name space management) is very similar to the model used in *SPIN*. They compare such a model against two other models: capability-based protection, and stack introspection. They show that each model has its own advantages and disadvantages, and that none is clearly superior. In this paper we explore the specific language design issues that are involved with using the name space model.

2.2: *SPIN* and Modula-3

SPIN [3] is an extensible operating system that allows untrusted applications to safely extend system services. Extension code in *SPIN* must be written in the type-safe subset of Modula-3 [21]. Applications, which run in their own address spaces, can dynamically link extension code into the kernel’s address space. Applications can also install upcalls to extension code through event-based dynamic interposition [23].

In Modula-3 the primary program structuring units are interfaces and modules. Interfaces and modules are compile-time units. In order to provide a flexible linkage and access control to code, *SPIN* provides *domains* [24], which are name spaces of program symbols. A domain is a collection of interfaces and modules that is created at dynamic link-time. The most important features of domains for this paper are the following: a domain is a name space, and linkage against a domain is controlled by an access control list.

SPIN uses access control over domains to protect the system from extensions, and extensions from each other. Extensions, which are themselves loaded as domains, are only allowed to link against domains for which they have the appropriate permission. Linkage means the resolution of undefined symbols in a domain, such as type names and interface names. In *SPIN*, every extension must acquire permission to link against D_{spin} , the public system domain that exports all of *SPIN*’s public interfaces. If an extension wants to use symbols in any other domain, it must link against that domain. After an extension has linked against a domain, it is allowed to access any of the symbols that have been resolved from that domain. This protection model is less flexible than capability-based systems, but involves fewer run-time checks.

Class Operations	Access Control Mechanisms	
	Compile-time	Link-time
Definition	unique abstract types	check identity
Subtype or subclass	final classes or methods	check identity
Member access	public, private, protected	multiple interfaces
Instantiation	constructors	multiple interfaces
Method extraction	super restrict extraction to subclasses	check identity

Table 1. Compile-time vs. link-time access control mechanisms over class operations. By “check identity”, we mean that the linker checks the identity of whoever is linking against the controlled code.

Extension code in *SPIN* is downloaded in binary form. *SPIN* assumes that digital signatures can be used to ensure that binaries come from a trusted source, as in the Inferno system [9]. In the future, it may be possible to use proof-carrying code [20] to ensure that binaries are type-safe. In either case, a dynamic check is necessary to ensure that compile-time restrictions (such as type safety) are obeyed by object code.

3: Class operations

We describe a framework for reasoning about access control for object-oriented extension languages. This framework is based on access control over class operations. In typical, statically typed, class-oriented languages, such as Java and Modula-3, there are five operations that can be performed on classes:

- **Definition.** A class can be created.
- **Subclassing.** A class can be subclassed.
- **Member access.** An instance of a class can have its methods and fields accessed by clients.
- **Instantiation.** An instance of a class can be created.
- **Method extraction.** Some languages allow methods to be extracted from classes as procedures.

Each of these operations must have access controls on them, in order to prevent untrusted extension code from performing illegal operations. In each of the following sections, we explain why the operation must have access control, how access control is (or is not) provided in Modula-3 and Java, how the designs of Modula-3 and Java interact with access control, and any changes we made to Modula-3 for *SPIN*.

Table 1 summarizes the class operations that we examine, and the compile-time and link-time mechanisms that can be used to control access to them. Run-time mechanisms to check each operation in ways analogous to link-time mechanisms can also be used to achieve greater flexibility. In *SPIN* we have generally chosen to use link-time mechanisms: they are more flexible than compile-time mechanisms, and less costly than run-time mechanisms.

```

(* System Code *)
INTERFACE Thread;
IMPORT CPU;
TYPE T <: ROOT;
TYPE Regs = CPU.RegisterFile;
END Thread.

MODULE Thread;
(* Make the type Thread.T unique *)
REVEAL T = BRANDED OBJECT
    machineState : Regs;
    (* ... *)
END;
END ThreadPrivate.

(* Extension Code *)
MODULE Untrusted;
IMPORT Thread, CPU;
TYPE ExposedThread = OBJECT
    machineState : Thread.Regs;
    (* ... *)
END;

VAR thread := NEW (Thread.T);
    eth : ExposedThread;
BEGIN
    (* the following statement is illegal,
       because Thread.T is a unique type
       *)
    eth := NARROW (thread, ExposedThread);
    eth.machineState[CPU.PC] := 16_dead;
END Untrusted.

```

Figure 1. Class uniqueness in Modula-3. The fact that `Thread.T` is unique prevents it from being spoofed. If `Thread.T` were not unique, the extension code could create an equivalent class, which would enable it to manipulate the thread’s program counter. For example, the extension could set the program counter to an invalid address.

3.1: Class definition

In order to protect a class, a programmer must be able to make its definition unique. Otherwise, a malicious client could define an equivalent class. With that equivalent class, the client could access the internals of the class, as well as create his own instances of the class.

Our experience in *SPIN* is that structural equivalence creates uniqueness problems, because it does not allow for a notion of uniqueness. Fortunately, Modula-3 supports both structural equivalence and a form of name equivalence, as illustrated in Figure 1. A class can be declared unique using the **BRANDED** keyword. In Figure 1, `Thread.T` is unique; if it were not, a client could guess the structure of `Thread.T`, and duplicate it in the declaration of the `ExposedThread` class. The `ExposedThread` class would be the same as `Thread.T`, and the client could create illegal threads, or modify the internal fields of legal threads. In order to enforce protection in *SPIN*, every class that can be used by an untrusted client must be **BRANDED**.

Java uses name equivalence, except for array types. As a result, all classes except arrays are unique, so a programmer cannot easily make the mistake of not making a class unique. In order to make an array class unique, it must be wrapped inside an object.

Modula-3 is more flexible than Java because it supports both structural and name equivalence. Therefore, classes that do not need to be protected need not be declared as abstract data types. The flip side of this flexibility is that programmers in *SPIN* must remember to brand their types when necessary.

A dynamic means for protecting classes would be to check all structurally equivalent classes at link-time, to see if the programmers who defined them trust each other. Such a link-time mechanism would have the advantage of more flexibility over compile-time

```

(* System Code *)

INTERFACE Thread;
IMPORT CPU;
TYPE T <: ROOT;
TYPE Regs = CPU.RegisterFile;
END Thread.

MODULE Thread;
(* Thread.T is unique, but... *)
REVEAL T = BRANDED OBJECT
  machineState : Regs;
  (* ... *)
END;
END ThreadPrivate.

(* Extension Code *)

MODULE Untrusted;
IMPORT Thread, CPU;
(* ExposedThread is a subclass of
   Thread.T, so it can access the
   internal state of Thread.T.
   *)
TYPE ExposedThread = Thread.T OBJECT
  (* extra fields and methods *)
END;

VAR thread : Thread.T;
BEGIN
  thread := NEW (ExposedThread);
  eth.machineState[CPU.PC] := 16_dead;
END Untrusted.

```

Figure 2. Subclassing in Modula-3. An extension can spoof the system’s `Thread.T` by subclassing it. Similar to the example in Figure 1, the extension could then manipulate the thread’s program counter.

mechanisms. For example, certain clients would be allowed to define equivalent classes, and have greater access to class internals. Such flexibility seems of limited use, because other mechanisms (which we describe in the following sections) can be used to expose class internals to clients.

3.2: Subclassing

The ability to create abstract data types does not fully protect classes against spoofing. A malicious extension could use subclassing to create its “own” instances of a class, as well as gain access to the class’s internals. Therefore, a mechanism to restrict subclassing is necessary. At compile-time, a mechanism could be used to forbid subclassing of a class. At link-time, an access control check can be used to only allow trusted clients to subclass.

In *SPIN*, domains can deny linkage based on subclassing. In Figure 2, a client attempts to subtype the system `Thread.T` class. At dynamic link time, the client must ask to link its domain D_{client} against the system domain D_{spin} . The dynamic linker will not link D_{client} against D_{spin} unless the client has permission to create a thread subclass. For example, a superuser should be able to create a thread subclass that has different-sized stacks; a normal user should not be able to do so.

Java allows programmers to control most forms of subclassing only at compile-time. A non-abstract class that is declared as **final** cannot be subclassed. Because **final** is a compile-time mechanism, it does not allow programmers to say *who* is allowed to create subclasses. As a result, **final** does not support different protection policies, which prevents the programmer from allowing different extensions to subclass different classes. For instance, in our thread example, *SPIN* can prevent the D_{client} from subclassing the `Thread.T` class, but allow a privileged extension to do so.

Java does not provide any mechanism for controlling the subclassing of interfaces.¹ Compile-time mechanisms are not possible, because it would not make sense to have **final**

¹Java does provide a compile-time mechanism for controlling the visibility of interfaces.

interfaces: the whole point of interfaces and abstract classes is to allow multiple implementations. Nevertheless, access control over subclassing is necessary, in order to allow clients to use but not instantiate an interface. For example, an example similar to that in Figure 2 can be constructed, where the system code uses an interface to capture several of its own thread implementations: the system might want to prevent an extension from providing its own thread implementation.

The subclassing of non-interface classes in Java can only be controlled at run time, by putting access control inside constructors. For example, subclassing of class loaders is protected at run time in Java: the class loader constructor calls the security manager to check if the creation of a class loader is allowed. Such a mechanism has the disadvantage of imposing higher overhead than link-time mechanisms: a security check must happen upon every constructor call. In addition, although run-time checks are more flexible, they do not raise any errors until run time.

The primary advantage of using **final** is that it allows optimization of method dispatch to happen at compile-time. Without **final**, method dispatch optimization can only occur at link-time [7] or run-time [8]. In addition, Java also allows class members to be declared as **final**, in which case they cannot be redefined by subclasses. As a result, Java provides a finer grain of control over methods than Modula-3 in *SPIN*.

3.3: Member access

Class members (static methods and fields) must have their own access controls, in order to support flexible protection policies. Flexible control over access to members is necessary in order to distinguish between different levels of trust: it requires the ability to expose certain members to some clients, and hide them from other clients. Class members are usually protected using the same mechanisms (usually access control declarations) that are used to protect instance members.

In order to support flexible protection policies, *SPIN* depends on Modula-3's support for multiple *revelations* (object class interfaces). Modula-3 allows classes to export a number of different revelations, all of which must be strictly linearizable in terms of the members that they reveal. In *SPIN*, a class can export different revelations in different interfaces. Each interface can be exported from a different domain; different domains can then be given different access control lists. As a result, different clients can be given different views of class members.

Java provides a compile-time mechanism for controlling the views of a class through the member modifiers: **public**, **private**, and **protected**. These modifiers can be applied to methods and fields, and control visibility with respect to packages. The ability to declare members **protected** can be used to control the effects of subclassing (which was discussed in Section 3.2) to a limited extent, since **protected** members are only visible to subclasses. Nonetheless, member modifiers in Java can only enforce static restrictions, which limits the flexibility of the protection model.

3.4: Class instantiation

Default constructors can cause access control problems in extensible systems. Several languages, such as Modula-3, Oberon [19], and C++ [6] provide such constructors. Because these languages provide default constructors for every object class, any client can create an instance of the class without the possibility to prevent such creation. Although default

constructors are a convenience for programmers, we show that they can leave access control holes if they are not properly designed.

Modula-3 is an example of a language that does not allow class implementors to control constructors at compile-time. In Modula-3, clients can call the default **NEW** operator on any object class — Nelson discusses the design rationale for this decision in a short article [22]. As a result, calls to **NEW** can only be enforced at link-time, or at run-time.

Ideally, we would fix Modula-3 for *SPIN* by removing **NEW** as a default constructor. Unfortunately, such a solution would require us to change a great deal of existing Modula-3 code. For example, all Modula-3 code that creates **MUTEX** objects uses **NEW** to do so. Another possible solution would be to add initializers, which would be called whenever **NEW** is called. Such a language change would be rather drastic, as it would change the basic semantics of objects in Modula-3. A third possible solution would be to allow programmers to override default constructors (as in C++). Such a change would also require us to change a great deal of existing Modula-3 code. Our solution, which was chosen primarily for simplicity, is to require the programmer to check in object methods whether the object was properly created. This solution is somewhat unsatisfactory, but we chose it because all of the solutions had drawbacks.

Java allows programmers to guarantee object initialization at compile-time, because a programmer can specify exactly what constructors a class exports. Initialization is guaranteed because clients can only create objects by calling these constructors. Java also requires that subclasses call superclass constructors, which helps to reduce the likelihood of errors in initialization.

Note that constructors, except for the presence of default constructors, are simply class members. Therefore, the link-time mechanisms described in Section 3.3 can be used to protect access to constructors.

3.5: Method extraction

Some languages allow a client can “un-override” methods in a subclass: a client of a subclass can call superclass methods on an instance of the subclass, even if the subclass overrides those methods. For example, in Modula-3, C++, and Oberon, the explicit use of the syntax `T.write` refers to the procedure that implements `write` in the class `T`. The resulting procedure can be called on any subclass of `T`. Allowing clients to invoke superclass methods on a subclass instance effectively leaks the superclass implementation into the subclass, which violates protection.

In *SPIN* this problem is addressed by restricting the ability to explicitly extract methods at compile-time. First, only a module that implements a subclass can explicitly name a method in the superclass. Concretely, `T.write` can only be called within a module that defines a subclass `Tlocal` of `T`. Second, the explicitly named method can only be called on instances of the subclass `Tlocal` defined in the given module. That is, the subclass implementor can only call the method on his subclass, not on a sibling class. Finally, an explicitly named method may only be called: that is, it cannot be used as a value. Otherwise, a malicious programmer could circumvent the first two restrictions.

Figure 3 illustrates the first two restrictions on method extraction. The `CompressedDisk` module may extract methods on the `CompressedDisk.T` class, since it is a subclass of `Disk.T` and it is in the same scope of the subclass definition. On the other hand, the `CompressedDiskClient` module is incorrect. It may not extract the `write` method from

```

(* System Code *)

INTERFACE Disk;
TYPE Data = REF ARRAY OF CHAR;
TYPE T = BRANDED OBJECT METHODS
    write(data: Data);
    read() : Data;
END;
END Disk.

(* Extension Code *)

INTERFACE CompressedDisk;
IMPORT Disk;
TYPE T <: Disk.T;
END CompressedDisk.

MODULE CompressedDisk;
IMPORT Disk;
REVEAL T = Disk.T BRANDED OBJECT
    (* write compressed data to disk *)
    OVERRIDES write : Write;
END;

MODULE CompressedDiskClient;
IMPORT CompressedDisk;
VAR
    cdisk := NEW(CompressedDisk.T);
    data : Disk.Data;
BEGIN
    (* improper usage of method extraction
       write uncompressed data to cdisk
    *)
    Disk.T.write(cdisk, data);
END CompressedDiskClient.

PROCEDURE Write(self:T; data:Disk.Data) =
    BEGIN
        (* compress the user provided data *)
        data := Compress(data);
        (* proper Type.method extraction
           call supertype's method *)
        Disk.T.write(self, data);
    END Write;

```

Figure 3. Method extraction in Modula-3. The extension code uses method extraction to invoke the `write` method of the `cdisk` object's supertype, which writes uncompressed data to the compressed disk object. As a result, subsequent `cdisk.read()` invocations will fail to read the data properly. The `cdisk.write` method (implemented by the `CompressedDisk.Write` procedure) makes legal use of method extraction to invoke its supertype's `write` method.

the `cdisk` object, because it does not define a subclass of `CompressedDisk.T`.

The two disadvantages of our language change are incompatibility and greater fragility. First, although this change to Modula-3 is incompatible, the practical effect was negligible. Of the three standard Modula-3 libraries that are used in *SPIN*, only one file out of nearly 600 had to be updated. Second, although directly invoking grandparent methods makes the class hierarchy more fragile, we do not consider the fragility unacceptable. Inheritance is a relation on implementations, and adding another implementation constraint is not overly restrictive. In a language where such fragility is unacceptable, the restrictions on method extraction could be strengthened so as to be equivalent to `super`.

Java does not allow methods to be extracted. `super` is used to allow subclasses to invoke supertype methods. `super` is strictly less expressive than our modified Modula-3. It is not clear that this lack of expressiveness is important. Note that `super` could not be used in Modula-3, because it can only be used inside a method: in Modula-3, methods are

Class Operations	System	
	<i>SPIN</i>	Java
Definition	BRANDED types	name equivalence
Subclass	link-time check	final classes constructor checks
Member access	multiple revelations link-time check	multiple interfaces public, private, protected
Instantiation	check at method invocation	no default constructors
Method extraction	restrict extraction to subclasses	no method extraction, super

Table 2. *SPIN* vs. Java mechanisms for access control over classes.

syntactically indistinguishable from procedures.

The use of method extraction could be controlled at link-time. Any piece of code that used method extraction would have to be checked at link-time; trusted principals would be allowed to link code that uses method extraction. We did not implement such a solution, because it seems unlikely that the extra flexibility gained by providing a link-time mechanism would be useful.

3.6: Comparison

Table 2 summarizes the differences between *SPIN* and Java with respect to access control over classes. In *SPIN*, we have generally chosen to use link-time mechanisms to provide access control over classes. These mechanisms are external to Modula-3, and are provided by *SPIN* domains. Java, on the other hand, generally uses compile-time restrictions based on language mechanisms to control access to classes.

4: Conclusions

As object-oriented programming languages become widely used in extensible systems, a better understanding of the interaction between object-oriented language design and protection issues is necessary. Many object-oriented languages are not immediately suitable for use in an extensible system, because of features that were not designed to be used in less trusted environments. We have presented a case study of access control in an object-oriented extension language for a system that uses name spaces for protection, and have used Modula-3 and Java as examples.

Telescript [26] is an object-oriented language designed by General Magic for writing mobile agents. The most interesting feature of Telescript is it directly supports the notion of a location at which a program executes. Otherwise, it is not significantly different in its features from other object-oriented languages.

Other object-oriented languages support different kinds of operations on classes. Obliq [4] is an prototype-based object-oriented language that preserves the notion of scope across a distributed system. In other words, the internal state of an object is protected, even if its state is distributed. Obliq allows object implementors to protect objects at compile-time from cloning by clients. Cloning in Obliq corresponds to subtyping in class-based languages, so Obliq's mechanism corresponds to **final** in Java.

The Theta language [15] is used to write safe extensions in the Thor object-oriented database [14]. Theta is an object-oriented language that supports separate subtyping and inheritance hierarchies: a subclass need not be a subtype of its superclass. Theta allows a class to forbid subclassing at compile-time; as we have discussed, dynamic access control mechanisms are more flexible. Theta also supports special methods called *makers* that allow a superclass to isolate itself from a subclass: a subclass uses a maker to initialize superclass fields, but cannot use the maker to create an instance of the superclass.

The notion of controlling subclassing points out a formal aspect of linkage that should be examined more carefully. Liskov and Wing defined the notion of behavioral subtyping [16]. Several researchers have described how behavioral subtyping could be enforced by forcing supertype specifications to be inherited by subtypes [5, 17]. A client can be allowed to subtype a type if its subtype can be verified to be a behavioral subtype. Ideally, clients at different trust levels could have different behaviors to satisfy. It is possible that a form of proof-carrying code could be used for automatically verifying that superclass specifications are inherited. It would be valuable to formalize this relationship between linkage, classes, and behavior specifications.

ML [18] is a functional language that allows programs to manipulate *structures* (code environments) as values through the use of *functors*. Functor application in ML corresponds to linkage in more traditional environments. In order to use ML in an extensible system, it would be necessary to provide access control over structures. For example, the ability to prevent functors from being applied to a structure would be necessary. In addition, it would be interesting to examine the relationship between access control over types and type inference.

Jones and Liskov proposed a mechanism for statically controlling fine-grained access rights by typing variables and expressions with $\langle type, access\ right \rangle$ pairs [10]. Their mechanism removes some of the run-time overhead for fine-grain access control. The use of such a mechanism has never gained popularity, probably because static, fine-grain access control is insufficiently flexible. In dynamically extensible systems, the ability to specify access rights should be dynamic.

In this paper we have presented a case study of using an object-oriented language in an extensible system: *SPIN*, an operating system that uses access control over name spaces for protection. In *SPIN*, we have chosen to use dynamic link-time mechanisms to provide access control in *SPIN*. These link-time mechanisms are generally more flexible than those in Java, and represent an interesting set of design choices for languages for extensible systems.

5: Acknowledgments

We thank Patrick Tullmann, Greg Morrisett, Kathleen Fisher, Bob Gruber, Stefan Savage, Michael Ernst, Craig Chambers, Gary Leavens, and the anonymous referees for their feedback on drafts of this paper.

References

- [1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

- [2] 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.
- [3] 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.
- [4] L. Cardelli. "A Language with Distributed Scope". *Computing Systems*, 8(1):27–59, January 1995.
- [5] K.K. Dhara and G.T. Leavens. "Forcing Behavioral Subtyping Through Specification Inheritance". In *Proceedings of the 18th International Conference on Software Engineering*, pages 258–267, Berlin, Germany, March 1996. IEEE Computer Society Press.
- [6] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, New York, NY, 1990.
- [7] M.F. Fernández. "Simple and Effective Link-Time Optimization of Modula-3 Programs". In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, La Jolla, CA, June 18–21, 1995.
- [8] U. Hölzle and D. Ungar. "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback". In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–335, Orlando, Florida, June 1994.
- [9] "Inferno: la Commedia Interattiva". <http://inferno.lucent.com/inferno/infernosum.html>, 1996. A brief overview of the Inferno OS.
- [10] A.K. Jones and B.H. Liskov. "A Language Extension for Expressing Constraints on Data Access". *Communications of the ACM*, 21(5):358–367, May 1978.
- [11] T. Kistler and M. Franz. "A Tree-Based Alternative to Java Byte-Codes". In *Proceedings of the International Workshop on Security and Efficiency Aspects of Java '97*, 1997. Also available as TR 96-58, Department of Information and Computer Science, UC Irvine, December 1996.
- [12] D. Kramer. "The Java Platform". Contributions by B. Joy and D. Spenhoff. Available on the Web. <http://www.javasoft.com:81/doc/whitePaper.Platform/CreditsPage.doc.html>, May 1996.
- [13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, January 1997.
- [14] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. "Safe and Efficient Sharing of Persistent Objects in Thor". In *Proceedings of the 1996 International Conference on Management of Data*, Montreal, Canada, June 1996.
- [15] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A.C. Myers. "Theta Reference Manual". Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, February 1995. <http://clef.lcs.mit.edu/papers/thetaref/index.html>.
- [16] B. Liskov and J. Wing. "A Behavioral Notion of Subtyping". *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
- [17] B. Meyer. *Eiffel: the language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
- [18] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [19] H. Mössenbock and N. Wirth. "The Programming Language Oberon-2". *Structured Programming*, 12(4):179–195, 1991.
- [20] G.C. Necula. "Proof-Carrying Code". In *Proceedings of the 24th Symposium on Principles of Programming Languages*, Paris, France, January 1997.
- [21] G. Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [22] G. Nelson. "Initialization of Object Types". *Threads: A Modula-3 Newsletter*, Fall 1995. Available on the Web, <http://www.cmass.com/threads/1/>.
- [23] P. Pardyak and B.N. Bershad. "Dynamic Binding for Extensible Systems". In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pages 201–212, Seattle, WA, October 1996.
- [24] E.G. Sirer, M. Fiuczynski, P. Pardyak, and B.N. Bershad. "Safe Dynamic Linking in an Extensible Operating System". In *First Workshop on Compiler Support for System Software*, pages 141–148, Tucson, AZ, February 23–24, 1996.

- [25] D.C. Swinehart, P.T. Zellweger, R.J. Beach, and R.B. Hagmann. "A Structural View of the Cedar Programming Environment". *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.
- [26] J. Tardo and L. Valente. Mobile agent security and Telescript. In *Proceedings of the 41st IEEE Computer Society International Conference*, pages 58–63, Santa Clara, CA, February 1996.
- [27] D.S. Wallach, D. Balfanz, D. Dean, and E.W. Felten. "Extensible Security Architectures for Java". In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 116–128, St. Malo, France, October 1997.