

1 Introduction

We are currently developing JavaSPIN, a seamless integration of persistence, interoperability and naming with Java. The foundations for JavaSPIN are the SPIN framework, the kernel of the TI/Arpa Open Object-Oriented Database (Open OODB) and Java itself.

Our work on JavaSPIN has three main objectives:

- To produce a seamlessly extended version of Java having valuable capabilities beyond those provided in the basic Java language but with minimal barriers to adoption;
- To complement similar extensions to C++ and CLOS, thereby providing a convenient basis for extending our work on polylingual interoperability [5]; and
- To demonstrate the usefulness and generality of our previously-developed approaches to providing persistence, interoperability and name management.

In the remainder of this paper, we first provide some background on our previously-developed approaches to providing persistence, interoperability and name management. We then discuss our goals for JavaSPIN, outline the JavaSPIN approach and sketch our plans for longer-term development related to JavaSPIN.

2 Background

The SPIN (Support for Persistence, Interoperability and Naming) framework [3] was developed as a unifying conceptual foundation for integrating extended features in software systems. SPIN has previously been used as a basis for seamlessly integrating persistence, interoperability and naming capabilities in extended versions of the C++ and CLOS APIs of the Open OODB. The SPIN framework itself evolved out of our earlier work on persistence, interoperability and name management.

Our work on persistence began with the development of the PGRAPHITE [8, 10] and R&R [7] persistence extensions to Ada 83. At the time, PGRAPHITE and R&R were novel in that they added orthogonal, reachability-based persistence to (a subset of) an existing language by exploiting its data abstraction features. As a result, the persistence extensions required no new syntax and no modifications to the language system. Instead, a preprocessor was used to make transparent, source-language-level modifications to user-defined abstract data types. The modifications were transparent in the sense that only code directly concerned with persistence needed to be aware of them; code not concerned with the persistence properties of objects could be completely oblivious to the changes.

At about the same time, we were also beginning to investigate approaches to interoperability [11]. Our work on this topic, like our work on persistence, emphasized minimizing the

impact of the extended capability. In particular, we sought to make any decisions regarding interoperation transparent to code that wasn't directly involved in or dependent upon interoperation.

The TI/Arpa Open OODB project [9] subsequently adopted a perspective very similar to that which we had taken in PGRAPHITE and R&R. The Open OODB project produced both a Persistent C++ and a Persistent CLOS without introducing any new syntax in either language or any modifications to either language system. Where our PGRAPHITE and R&R prototypes had provided persistence for only a subset of Ada 83, Open OODB offers APIs providing persistence for essentially all of C++ and CLOS (all instances of classes). Like PGRAPHITE and R&R, however, and unlike most approaches to persistence for C++ (e.g., the ODMG C++ binding [2]), the Open OODB persistence extensions are seamless and transparent. The Open OODB approach also resembles the PGRAPHITE/R&R approach in that it is implemented via source-language-level modifications. For both C++ and CLOS, these modifications effect multiple (“mix-in”) inheritance, at least conceptually, of a `PersistentObject` class with any class whose instances are to be extended with the potential for persistence. This inheritance is exactly analogous to what the PGRAPHITE and R&R preprocessors were doing; indeed, those preprocessors would also have effected mix-in inheritance if Ada 83 had been an object-oriented language. The decision to implement via source-level modifications was appropriate, both for our preprocessors and for Open OODB, since source is the most portable representation for Ada, C++ and CLOS.

Given the similar perspectives on persistence underlying our PGRAPHITE/R&R work and the Open OODB, it was natural for us to collaborate with the Open OODB project. As part of that collaboration, we extended Open OODB (both APIs) with enhanced, language-neutral, name management support. This in turn enabled some novel, name-based persistence mechanisms and polylingual interoperability of persistent C++ and CLOS objects [5]. The combination of support for persistence, interoperability and name management rests on the foundation that we call the SPIN framework. When extended with automated support for polylingual persistence, we refer to the framework as PolySPIN.

For various reasons that will become evident shortly, the features of Java seem to make it an ideal candidate for very transparent and seamless extension with persistence, interoperability and name management support based on the SPIN (and eventually PolySPIN) framework. The remainder of this paper describes our current efforts aimed at creating a JavaSPIN prototype that will instantiate such a transparent, seamless extension and move us in the direction of realizing the objectives enumerated in Section 1.

3 Goals for JavaSPIN

Our JavaSPIN approach is motivated by the following goals:

Seamless Extension of Java Capabilities: Our highest priority is to provide a set of extensions to Java in the most seamless manner possible. Seamlessness implies that our extensions will be compatible with Java and the programming style that it defines. In particular, our extensions should not compromise the type safety or security properties of Java. The specific extensions that JavaSPIN will provide are:

Persistence: JavaSPIN will provide orthogonal, reachability-based persistence for Java in the style that Open OODB provides for C++ and CLOS. It will also provide the same kind of name-based persistence capabilities for Java that our enhancements to the Open OODB (incorporated in Open OODB 1.0) make available for C++ and CLOS.

Enhanced name management: Independent of (that is, orthogonally to) persistence, JavaSPIN will provide a set of extended name management capabilities, based on the Piccolo model [6] and therefore suitable for use with Conch-style tools [4]. As a result, this enhanced approach to name management will be uniformly applicable to C++, CLOS, and Java objects.

A Basis for polylingual interoperability among C++, CLOS, and Java: The extensions provided by JavaSPIN will transparently incorporate the necessary information into Java objects to support polylingual interoperability among C++, CLOS, and Java [5].

Minimal Barriers to Adoption: Our next highest priority is to make it as easy as possible for Java users to adopt our extensions. As always, we wish to minimize the impact of the extensions on programmers, especially those who might not be (direct) users of the extended capabilities. Hence, this goal is closely related to seamlessness. Two specific ways in which we intend to minimize barriers to adoption are:

No language extensions: JavaSPIN will not introduce any modifications in the syntax (including keywords) of Java, nor will it require the use of an additional or separate specification language. (This is in contrast to PGRAPHITE and R&R.)

No virtual machine modifications: By making it possible to run JavaSPIN programs on the standard Java Virtual Machine, we hope to encourage the use of JavaSPIN from web browsers and in other settings where adoption of a modified virtual machine is unlikely.

Maximal Opportunities for Interoperability: A unique feature of the JavaSPIN approach is that it will directly facilitate interoperation among C++, CLOS, and Java programs. This is because JavaSPIN will:

- Share the SPIN (and eventually PolySPIN) conceptual base with the C++ and CLOS APIs of Open OODB 1.0.
- Share use of the Open OODB kernel with the other Open OODB APIs.

Suitable Basis for Future Research: We intend to use JavaSPIN as a foundation for various experiments and extensions. To that end, we plan to make the system well modularized, with an open architecture and clean, well-defined interfaces.

4 The JavaSPIN Approach

In this section we outline our JavaSPIN approach. We first describe the APIs that JavaSPIN users will see. We then sketch our implementation strategy for JavaSPIN.

4.1 APIs

There will be two related APIs for JavaSPIN. The first will provide basic, Open OODB-style persistence to Java users. The second will extend the first by providing enhanced name management and name-based persistence for Java. The APIs will include methods added to each class processed by JavaSPIN, together with one or more JavaSPIN-specific classes (in a package tentatively named `EDU.umass.cs.ccs1.JavaSPIN`).

The appearance will be that all methods are added to the `Object` class and inherited from it by every other class. In reality this will not be the case because of return-type restrictions. Specifically, the `fetch` and `resolve` methods of a class are required to return an object of that class and thus must be specific to the class.¹

The basic API adds the following methods to each class:

`public void persist([String name])` When invoked on any object, this method will result in that object, and all objects reachable from it, becoming persistent. The optional `name` parameter can be used to assign a name to the persistent object, by which name it can later be retrieved. If no name is assigned, the object can only be retrieved if it is referenced from some other object.

`public void unpersist()` When invoked on an object that has previously been made persistent, this operation makes that instance transient.

¹We could inherit the `fetch` and `resolve` methods, but then they would have to return `Object` and the programmer would be required to cast the returned `Object` to an object of the desired class. This remains type-safe, but is slightly unpleasant.

`public static class fetch(String name)` When invoked, this method will return the persistent instance of the class corresponding to the name given by the `name` parameter. If there is no such instance in the persistent store, the `UnknownPersistentName` exception will be thrown.

The basic API also defines the `Store` class (in the `JavaSPIN` package):

```
public class Store {
    public void openStore(String store);
    public void beginTransaction();
    public void commitTransaction();
    public void abortTransaction();
    public void closeStore();
}
```

This class provides the means for establishing connection to a specific Open OODB persistent store, in the same manner as is done for the C++ and CLOS APIs. It also provides rudimentary transaction capabilities, again mirroring those provided by the other Open OODB APIs. Richer transaction models for `JavaSPIN` will be a subject of future research.

The second `JavaSPIN` API provides the following three classes: `Name`, `BindingSpace` and `Context`. The `Name` class encapsulates information about user-level names for both transient and persistent Java objects. Various operations for creating and parsing names are provided. The `BindingSpace` class provides a collection or directory abstraction for named Java objects. A specially designated instance of `BindingSpace` serves as the persistent root binding space for `JavaSPIN` programs. Finally, the `Context` class is used to form an appropriate context from one or more binding spaces and contexts. Here various methods are defined for forming suitable contexts for `JavaSPIN` programs.

```
public class Name {
    public void setDelimiter(char delimiter);
    public void create(String name);
    public Name leftmost();
    public Name rest();
    ...
}
```

```
public class BindingSpace {
    public void initialize();
    public boolean empty();
}
```

```

    public void beginIterate();
    public Name next();
    public void endIterate();
    ...
}

public class Context {
    public void initialize();
    public void infuse(BindingSpace bspace);
    public void unionOverride(BindingSpace bspace);
    public void restrict(Name[] names_to_include);
    public void exclude(Name[] names_to_exclude);
    ...
}

```

As noted above, the `Context` class provides various methods for forming contexts. The `infuse` method forms a context directly from a single binding space. The `unionOverride` method forms a context consisting of all the bindings in the context instance plus all the bindings in the binding space whose name components do not exist in any of the bindings in the context instance. `restrict` forms a context consisting of those bindings in the context instance whose name components are contained in a specified set of names, while `exclude` forms a context consisting of those bindings in the given context whose name components are not contained in a specified set of names.

The second JavaSPIN API further extends each class by adding the following methods:

```
public void assign(Name name, BindingSpace bspace) This method gives the object
    on which it is invoked the specified Name in the specified BindingSpace.
```

```
public static class resolve(Name name, Context context) This method is similar
    to the fetch method. When invoked it will return the persistent instance of the class
    corresponding to the name in the given context (as given by the name and context
    parameters, respectively). If there is no such instance in the persistent store, the
    UnknownPersistentName exception will be thrown.
```

4.2 Implementation Strategy

Our implementation strategy for JavaSPIN involves exploiting the data abstraction and object-orientation features of Java to seamlessly add the SPIN extensions. As described in the preceding subsection, the extensions are presented to the Java programmer in the form of additional methods for each class and some additional JavaSPIN-specific classes. We plan

to implement the addition of the methods to each class by modifying the Java compiler. Implementation of the added methods themselves, those added to each class and those in the JavaSPIN-specific classes, will primarily be done via native (i.e., C) method calls to components of the Open OODB kernel and of our name management and interoperability enhancements to Open OODB 1.0.

Basing our implementation on (re)use of the Open OODB kernel and our existing (C++-implemented) name management and interoperability code has several advantages. It clearly simplifies the implementation task by leveraging existing, reasonably robust, code. It also lays the groundwork for interoperability by having not just common specifications but largely common implementations of the persistence, naming and interoperability features of JavaSPIN and the Persistent C++ and Persistent CLOS APIs of the Open OODB.

Our choice of modifying the compiler, rather than providing a preprocessor, as was done for PGRAPHITE/R&R and for Open OODB, is motivated by some features of Java. The existence of a single root (the Object class) for the Java class hierarchy makes mix-in inheritance unnecessary and suggests a single point of modification, which is most straightforwardly handled via compiler modification. Moreover, since Java compilers produce a standardized byte code, we needn't restrict ourselves to source-code-level modifications in order to maximize portability of our approach. Finally, the availability of run-time-accessible class descriptions and of a stabilization capability suggest that our compiler-oriented strategy is very much in the style of the Java language.

5 Plans

We have completed a prototype of the persistence mechanisms of JavaSPIN. This prototype includes most of the facilities described in Section 4 pertaining to persistence, but did not include the modifications to the Java compiler. The per-class methods that the compiler will eventually generate were inserted by hand in our test programs.

In the near term, our plans focus on completing a prototype implementation of JavaSPIN. During this continued implementation effort we intend to explore several variations and alternatives regarding some aspects of the APIs and implementation strategies outlined in Section 4. We will report on the relative advantages and disadvantages of these alternatives in future papers. We expect to use to advantage the new capabilities, especially reflection, available in version 1.1 of the Java Developer's Kit.

In the longer term, we will undertake a variety of enhancements, extensions, improvements and experimentation based on JavaSPIN. For example, we plan to investigate various transaction models by replacing the initial model implemented in our `JavaSPIN.Store` class with richer alternative models. We also intend to extend our PolySPINner toolset [1] to automate polylingual interoperability support among Java, C++ and CLOS, thereby com-

plementing our existing support for polylingual interoperability between C++ and CLOS [5]. Another topic that we expect to explore using JavaSPIN is evolution of type definitions in the presence of persistent instances of the evolving types. Performance improvements, possibly based on modifying or replacing the underlying object store utilized by the Open OODB kernel, will also be investigated. Due to Java's user-accessible information about internals of the language and of programs (e.g., the class structures), JavaSPIN also seems likely to provide a good basis for attempting to achieve a tighter integration of the names, binding spaces and contexts used in our extended name management mechanism with the native name management features of the programming language. These and other investigations will contribute to improvements in support for persistence (and other extensions) for Java as well as to increased understanding of programming languages and their properties in general.

6 Conclusion

In this paper, we have outlined our goals, our approach, and some future plans for an integrated set of extensions to add persistence, interoperability and enhanced name management capabilities to Java. We believe that having the appropriate foundations makes the seamless addition of these extensions relatively clean and straightforward. One of those foundations is the SPIN framework, which offers a smoothly integrated set of concepts supporting the synergistic interaction of persistence, interoperability and name management. Another foundation is the kernel of the Open OODB, which gives us a proven substrate on which to implement JavaSPIN as well as a basis for polylingual interoperability among Java, C++ and CLOS through a common persistent store. The final foundation is the Java language itself. Java features such as a singly-rooted class hierarchy and the run-time accessibility of class information greatly facilitate seamless extension.

Not only does JavaSPIN offer the prospect of a persistent Java with minimal barriers to adoption, but it also provides a path to interoperation with the Persistent C++ and Persistent CLOS APIs of the Open OODB. We expect this to be extremely valuable. It will, for example, facilitate interoperation between Java programs and existing software written in C++ or CLOS. It also represents a simple route to an object querying capability, since Open OODB provides an OQL for C++ which will be directly usable from, and on, Java programs and objects via our polylingual interoperability mechanism.

Finally, we believe that JavaSPIN will demonstrate the appropriateness of relying on the object orientation features, both data abstraction and inclusion polymorphism, of modern languages as the basis for seamlessly extending them with enhanced capabilities. Eschewing modifications to the language syntax or the virtual machine seems to us the essence of orthogonality. In the case of JavaSPIN, we think that this approach will minimize barriers

to, and hence maximize prospects for, adoption of a persistence capability for Java.

Acknowledgments

This paper is based on work supported in part by Texas Instruments, Inc. under Sponsored Research Agreement SRA-2837024 and by a subcontract from Intermetrics, Inc. funded by the Air Force Materiel Command, Phillips Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F29601-95-C-0003. The views and conclusions contained in this document are those of the authors. They should not be interpreted as representing official positions or policies of Texas Instruments, Intermetrics, or the U.S. Government and no official endorsement should be inferred.

We are grateful for the cooperation that we have received from Texas Instruments, Inc. Our special thanks go to Craig Thompson, David Wells and Steve Ford, all of whom have been particularly generous in assisting, supporting and encouraging our work. The quality of their own work on the Open OODB has made a very important contribution to the success of our efforts.

We also appreciate the contributions made by the members of the Java team and the authors of the “Persistence for Java” project in CMPSCI 530/630 during the Spring semester of 1996 – Stephen LaValley, Todd McCartney, Andrew Rynkiewicz, Qing Shi, Keith Visco, Joe Zhongjing Xu, – in helping us learn about Java and exploring some initial ideas about how persistence might be added to Java.

REFERENCES

- [1] D. J. Barrett, A. Kaplan, and J. C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *Proceedings of the Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996. (to appear).
- [2] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1993.
- [3] A. Kaplan. *Name Management: Models, Mechanisms and Applications*. PhD thesis, The University of Massachusetts, Amherst, MA, May 1996.
- [4] A. Kaplan and J. Wileden. Conch: Experimenting with enhanced name management for persistent object systems. In M. Atkinson, D. Maier, and V. Banzaken, editors, *Sixth International Workshop on Persistent Object Systems*, Workshops in Computing, pages 318–331, Tarascon, Provence, France, Sept. 1994. Springer.
- [5] A. Kaplan and J. Wileden. Toward painless polylingual persistence. In *Seventh International Workshop on Persistent Object Systems*, Cape May, NJ, May 1996. (To appear).
- [6] A. Kaplan and J. C. Wileden. Formalization and application of a unifying model for name management. In *The Third Symposium on the Foundations of Software Engineering*, pages 161–172, Washington, D.C., Oct 1995.
- [7] P. L. Tarr, J. C. Wileden, and L. A. Clarke. Extending and limiting PGRAPHITE-style persistence. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 74–86, August 1990.
- [8] P. L. Tarr, J. C. Wileden, and A. L. Wolf. A different tack to providing persistence in a language. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 41–60, June 1989.
- [9] D. L. Wells, J. A. Blakely, and C. W. Thompson. Architecture of an open object-oriented management system. *IEEE Computer*, 25(10):74–82, Oct. 1992.
- [10] J. C. Wileden, A. L. Wolf, C. D. Fisher, and P. L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, November 1988.
- [11] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, and P. L. Tarr. Specification level interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.