Toward a Conceptual Foundation for Interoperation

Jack C. Wileden

Convergent Computing Systems Laboratory
Computer Science Department
University of Massachusetts at Amherst
Box 34610
Amherst MA 01003-4610 USA
Tel: (413) 545-0289
Fax: (413) 545-1249
Email: wileden@cs.umass.edu

Abstract

There is steadily growing interest in constructing software applications that incorporate components from a variety of sources. Individuals and organizations involved in developing, operating or maintaining such applications increasingly face interoperability problems – situations in which components that were implemented using different underlying models or languages must be combined into a single unified application. To aid in overcoming such problems, a range of interoperability approaches have begun to be employed. Unfortunately, existing approaches suffer from various shortcomings. These shortcomings can often be traced to a lack of suitable foundations for understanding important aspects of interoperability.

This position paper advocates the development of some necessary foundations for interoperability. Such foundations would contribute to the conceptual base of several areas of computer science, notably including programmming languages and software engineering. These foundations would also have potential practical value, since they could provide a basis for developing improved tools and techniques applicable to interoperability problems facing practicing software professionals.

Our primary focus, in our own research and in this position paper, is on two crucial aspects of interoperability: name management and typing. Name management – how computing systems allow names to be established for entities, permit entities to be accessed using names, and control the availability and meaning of names at any point in time – and typing are both important issues that have been addressed in many computer science domains. From the perspective of interoperability problems, however, previous work on name management and typing is not sufficiently general or robust to contribute to the desired foundations. We therefore see a need for a conceptual foundation for interoperation in the form of rigorous, formal models for relevant aspects of name management, typing and possibly additional aspects of interoperability.

Keywords: Components, interoperability, conceptual foundation, name management, typing

Workshop Goals: Learning; networking; developing a research agenda for foundations of component-based systems.

1 Background

The focus of our long-term research program is summarized by the name of our research group – the Convergent Computing Systems Laboratory. The term *convergent computing system* is intended to denote any system that comprises features or components drawn from different computing domains (as, for example, a persistent programming language combines features from both the database and the programming language domains) or implemented using different technologies (as, for example, in the case of a software system consisting of modules written in two or more different languages). The group's long-range goal is to understand phenomena associated with convergence and to produce tools and techniques applicable to the development and maintenance of convergent computing systems. Since interoperability is perhaps the most important manifestation of convergence in contemporary computing practice, developing foundations for interoperability falls squarely within our purview.

Our research on interoperability began several years ago with an initial attempt to describe and categorize interoperability problems and approaches, coupled with an early effort at defining an architecture for automated interoperability support [WWRT91]. Since that time, our work on interoperability has been motivated by a primary concern for the impact of an interoperability approach on applications developers. In our view, among the most important objectives for any approach to interoperability are the following:

- Developers should have maximum freedom to define properties of objects that their programs manipulate. In particular, they should always be able to use the name management and type systems provided by the language(s) in which they are designing and developing components of their applications.
- Whether an entity is to be shared among an application's components should have minimal impact on the components' developers. In particular, making (or changing) a decision about whether, or with what other components, an entity may be shared should not affect the definition of, or interface to, the entity. As a corollary, interoperation should not result in an unnecessary reduction in the ways in which the (now shared) entities can be manipulated.

Given these objectives, we have been drawn to consideration of three major sets of issues regarding interoperability approaches. Briefly, these are:

Naming How are entities accessed by components that wish to interoperate through sharing those entities? Currently, individual components typically rely on distinct and often incompatible name management mechanisms, with the result that their interactions are hampered by inconsistencies in the semantics of their respective name management capabilities [KW95]. At best, existing interoperability approaches generally provide only limited, non-native name management capabilities.

Timing When is the decision to share entities among an application's components made? This question has a dramatic impact on the suitability of different approaches to interoperability. Three distinct timing scenarios for interoperability decisions can be characterized by the relationship among the relative times at which the sharing or shared components are developed and the decision to share them is made, as illustrated in Figure 1. The salient features of each scenario are:

Easiest case: The decision to share is made before any components are developed. In this case, all points of potential interaction (e.g., names or types of shared objects) can, in principle, be identified and given special treatment (e.g., type definition via OMG's IDL), which may substantially facilitate interoperation.

Common case: The decision to share is made after one of the sharing components is developed but before any others are. In this case, points of potential interaction in the existing component can, in principle, be identified and retroactively given special treatment (e.g., wrapped with interfaces defined via OMG's IDL) while points of potential interaction in the new components are given corresponding special treatment (e.g., matching IDL type definitions), which may somewhat facilitate interoperation.

Megaprogramming: The decision to share is made after the sharing components are developed. In this case, retroactive special treatments (e.g., wrapping with interfaces defined via OMG's IDL) are unlikely to result in compatible interfaces, so achieving interoperation is facilitated little, if at all, by most existing techniques.

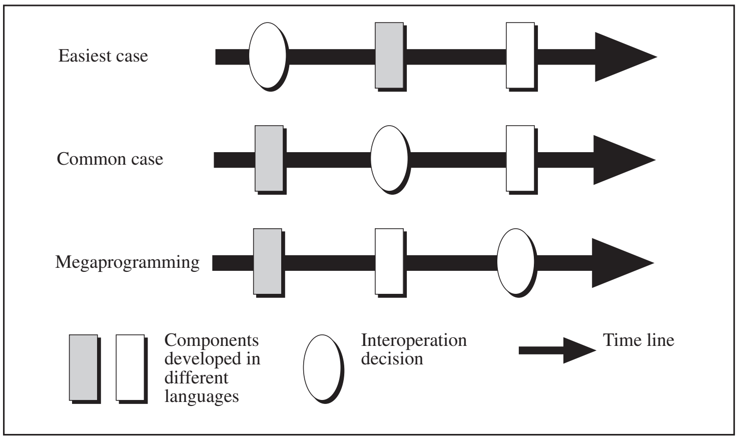

Figure 1: Interoperability Scenarios

Typing How do developers determine whether entities that they wish to share are of compatible types? Most existing approaches to addressing this question have been based on use of a unifying type model [WWRT91], such as the OMG IDL or the ODMG ODL. While such approaches may suffice for the easiest and common interoperability scenarios, however, they are inadequate for the megaprogramming case.

Our recent interoperability-related research has been focused on the issues of naming and typing. Oue work has been oriented toward addressing the third of the interoperability scenarios listed above and toward adhering to our principles of maximum freedom for, and minimum impact on, software developers. In the area of name management, we have been particularly interested in the formation, manipulation and use of *contexts*, which determine the meanings associated with individual names at a given point in time. We have also investigated extended notions of *closures*, which associate contexts with particular objects in order to refine each object's control over the meanings of names that it uses. Our research in this area has produced models of names and their usage that are more general and broadly applicable than previous models [KW93, KW95, Kap96]. It has also led to the development of name management mechanisms providing more powerful and accurate control over, and analysis of, context definition and use than any existing mechanisms [KW94, KW95, Kap96]. Prototype implementations of these mechanisms have been developed as extensions to the TI/Arpa Open Object-Oriented Database (Open OODB) [WBT92], and some of our name management capabilities are included in the currently released version of Open OODB.

Our research on typing issues in interoperability has centered on problems impeding cooperation among software modules of different languages, or *multi-language* interoperability problems. These naturally arise for any number of reasons, such as a desire to reuse legacy components, or because particular languages facilitate development of particular kinds of applications. In addressing multi-language interoperability problems, we make several distinctions. First, we differentiate situations in which a component written in one language needs to access one or more components (subprograms, data objects, etc.) written in a single second language from those in which a component written in one language needs to uniformly interact with a set of components written in two or more different languages (which may or may not include the language of the first component). We refer to the former, more homogeneous situation as *multilingual* interoperability, and the latter, more heterogeneous situation, as *polylingual* interoperability.[1] Second, we distinguish approaches by their level of *transparency* to software developers. Most current approaches force developers to be aware that multiple languages are involved and to build their modules to explicitly surmount the language boundaries. Multilingual interoperability may be achieved, for example, by using heterogeneous remote procedure calls, while polylingual interoperability can be realized by enforcing the use of a common, non-native type model among the modules (e.g., as in CORBA [BDE+95]). At the other end of the transparency spectrum, if developers of a multi-language software system need not be aware of language differences between the software modules then we call the interoperability approach *seamless*. Such seamlessness has long been a highly desirable property of interoperability, but it has rarely been achieved. As noted earlier, our work has always been motivated by a primary concern for the impact of an interoperability approach on applications developers, so we view seamlessness as an extremely important goal.

Existing approaches to interoperability, such as CORBA [BDE+95], ILU [JSS95], Concert [AR94], Microsoft's OLE [Bro94] or Apple's OpenDoc [App94], are generally not seamless because they typically involve use of a different invocation mechanism (e.g., low-level approaches and messaging systems) or a different type model (e.g., CORBA-style and database approaches) from those of the programming languages in which accessors are implemented. As a result, it is difficult to imagine how these approaches could be made transparent, even through automation. On the other hand, we have recently begun working on an approach that imposes neither different invocation mechanisms nor different type models and hence supports transparent, seamless interoperability. PolySPIN is an approach to Support for Persistence, Interoperability and Naming in POLYlingual systems [KW96]. The key component of PolySPIN is a language-neutral name management mechanism that allows for uniform name-based access to objects [Kap96]. This mechanism supports, but does not require, transparent polylingual interoperability. In the PolySPIN approach, a set of objects, implemented in diverse languages, can be transparently and automatically modified in such a way that they can be accessed seamlessly by software modules of different programming languages. That is, PolySPIN automates determination of *type compatibility* across language boundaries and implementation of cross-language access, thereby allowing a programmer to proceed as if the languages of the object and the accessor were the same. All software modules use only their native type systems to access these objects, not a common (but non-native) type system like CORBA's IDL, and existing modules need not be modified to do it. Hence PolySPIN meets our objective of minimizing the impact of an interoperability approach on applications developers. Our initial prototype implementations of PolySPIN and the PolySPINner toolset automating its use [BKW96] are based on the Open OODB. The way in which type compatibility is determined in these initial prototypes, however, lacks generalizable, formal foundations, does not address the polymorphism found in object-oriented programming languages, and does not assure type safety. Hence, we are

[1] This terminology is inspired by the analogy between "uniform processing independent of data type" (polymorphic) and "uniform processing independent of language" (polylingual).

currently developing a suitable formal foundation for polylingual type compatibility, based on concepts from signature matching [ZW95a] and object-oriented type theory [Bar97]. Our goal is to use this foundation to aid in reasoning about, and implementing support for, type safety in polymorphic, polylingual software systems [BRW97].

The various components of an approach to interoperability that we have developed to date have value in their own right, but we view them primarily as indicative that much more can and should be done. We believe that pursuing directions we have initiated, and exploring more rigorous and formal avenues, can lead to some suitable foundations for both theory and practice in the area of interoperability. In particular, we see an urgent need for a conceptual foundation for interoperation. It is our hope that participating in the workshop on Foundations of Component-Based Systems will contribute to developing an agenda for research that will lead us and others toward providing such a foundation. This in turn would contribute to our long-term goal of understanding convergence phenomena in computing systems.

2 Position

There is steadily growing interest in constructing software applications that incorporate components from a variety of sources. Component-based software development has been a long-standing goal of the software engineering community, as evidenced by the frequent appearance of terms such as *reuse* or *megaprogramming* [BS92] in the software engineering literature. Due to this trend toward utilizing software components from disparate sources, individuals and organizations involved in developing, operating or maintaining software applications increasingly face *interoperability problems.* All too often, when attempting to combine components that were implemented using different underlying models or languages into a single unified application, software practitioners find the effort excessive, or the results disappointing, or both. Given the pace of the developments that are encouraging component-based software development, we expect that interoperability will be among the most important software engineering challenges for the coming generation of software development and maintenance projects.

As interoperability problems have become increasingly apparent, a range of *interoperability approaches* have begun to be employed. Unfortunately, the approaches developed to date suffer from various shortcomings. Briefly stated, most current interoperability approaches lack general applicability, lack seamlessness and impose undue burdens on software developers and maintainers. It is our contention that these shortcomings can often be traced to a lack of suitable foundations for understanding important aspects of interoperability. Because such foundations are currently lacking, existing interoperability approaches are generally *ad hoc*, or based on limited models that are drawn from a single computing domain or programming language or system.

Our position is that a conceptual foundation for interoperation is urgently needed. More specifically, we advocate conceptual foundations for two crucial aspects of interoperability: name management and typing. Name management – how computing systems allow names to be established for entities, permit entities to be accessed using names, and control the availability and meaning of names at any point in time – and typing are both important issues that have been addressed in many computer science domains. From the perspective of interoperability problems, however, previous work on name management and typing is not sufficiently general or robust to contribute to the desired foundations. We therefore see a need for a conceptual foundation for interoperation in the form of rigorous, formal models for relevant aspects of name management, typing and possibly additional aspects of interoperability.

3 Comparison

A variety of approaches to interoperability have been developed over the years [WWRT91]. As noted earlier, however, most current interoperability approaches lack general applicability, lack seamlessness and impose undue burdens on software developers and maintainers. We believe that these shortcomings can often be traced to a lack of suitable foundations for understanding important aspects of interoperability. Indeed, there are presently no well-established conceptual foundations specifically addressing interoperability, and as a result the few technological foundations for interoperability that exist tend to be *ad hoc*, or based on limited models that are drawn from a single computing domain or programming language or system.

Because of the ubiquitous nature of names in computing, there have been a number of formalisms involving or related to the use of names (e.g., [CP89, Fra71, Joh71, Mil93, MABD90, Sal78, Sto77, WCW88]). Such formalisms, however, have generally been specific to individual, or limited sets of, computation domains. As a result, they are not directly appropriate for inclusion among the conceptual foundations of interoperability. In response to their limitations, we developed the PICCOLO[2] model [KW95]. PICCOLO generalizes various concepts found in many of these models and extends them with additional features to produce a unified perspective on a more complete set of name management issues across a much broader range of computational systems. Although only an initial model, with clear shortcomings of its own, we believe that PICCOLO is representative of the kind of unifying model of name management that could contribute to foundations for interoperability.

There has, of course, been a great deal of conceptual work dealing with typing in programming languages. Without exception, however, that work has been focused on typing models, type checking or type inference in a single language. Since the essential typing issue for interoperability is type compatibility across language boundaries, none of this existing work is immediately appropriate for a role in foundations for interoperability. The research of Zaremski and Wing on *signature matching* [ZW95a] and *specification matching* [ZW95b] offers one important starting point, however, since it focuses on various notions of matching among various representations of types, which is quite relevant to type compatibility determination. To date, though, this work also has been restricted to single-language settings and hence is not immediately applicable to interoperability. Another potentially relevant line of research includes several attempts to establish formal models for typing in object-oriented systems. Two prominent examples of this work are the recent books by Palsberg and Schwartzbach [PS94] and by Abadi and Cardelli [AC96]. Each develops a general framework within which important aspects of object-oriented type systems, such as inheritance, may be understood. Although not immediately applicable to interoperability, each is a potentially valuable contributor to a suitable formal model of type compatibility in object-oriented interoperability. As noted earlier, we are currently developing a formal foundation for polylingual type compatibility to support reasoning about, and implementing support for, type safety in polymorphic, polylingual software systems [BRW97]. This foundation is presently based on concepts from Zaremski and Wing's signature matching and the Abadi and Cardelli approach to object-oriented type theory [Bar97].

The Evolving Algebras (EA) formalism [Gur88] has been developed as a general purpose model of computational systems and applied to studying such things as the semantics of

[2] Precise Interface and Context Control in Object Libraries and Objects

Modula-2 [GM88]. We ourselves have used it as a basis for formalizing the semantics of the PICCOLO model [KW95]. Because it is an operational semantics formalism and hence EA models can be simulated using an interpreter [HM95], it has the advantage of providing some automated support as well as formal rigor. To make it useful in a foundation for interoperability, however, extensive development of appropriately high level EA models would be required.

Very recently, a few researchers have begun to explore some other formal foundations specifically aimed at interoperability concerns. Notable among these are the *interconvertibility* notion developed in the Mockingbird project [ACC97], the bottom-up type modeling approach to interoperability proposed by Sánchez-Ruíz [SR96] and the Larch-based formalization of IDL semantics by Sivaprasad [Siv95]. While none is likely to produce an appropriate foundation for polylingual interoperability, these various research efforts complement our own work toward developing such a foundation. The existence of these other efforts also suggests that we are not alone in believing that a conceptual foundation for interoperation is urgently needed.

4 Acknowledgments

We gratefully acknowledge the contributions that member of the Convergent Computing Systems Laboratory, notably Alan Kaplan, Dan Barrett and John Ridgway, have made to the research results and perspectives described here. We also appreciate the funding provided by the Defense Advanced Research Projects Agency via contracts administered by Intermetrics Inc. and Texas Instruments, in support of CCSL research.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, 1996.

[ACC97] Joshua Auerbach and Mark C. Chu-Carroll. The mockingbird system: A compiler-based approach to maximally interoperable distributed programming. Research Report RC 20718, IBM T. J. Watson Research Center, February 1997.

[App94] Apple Computer, Inc. OpenDoc for Macintosh: An overview for developers. White paper, Apple Computer, 1994.

[AR94] Joshua S. Auerbach and James R. Russell. The Concert signature representation: IDL as intermediate language. *ACM SigPLAN Notices*, 29(8):1–12, August 1994. From the Proceedings of the ACM Workshop on Interface Definition Languages, 1994. Originally IBM Research Report RC19229.

[Bar97] Daniel J. Barrett. *Polylingual Systems: An Approach to Seamless Interoperability*. PhD thesis, University of Massachusetts, Amherst, MA, September 1997.

[BDE+95] BNR Europe Limited, Digital Equipment Corporation, Expersoft Corporation, Hewlett Packard Corporation, IBM Corporation, ICL, plc, IONA Technologies, and SunSoft, Inc. CORBA 2.0/Interoperability. OMG TC Document 95.3.xx, Object Management Group, March 1995. Revision 1.8.

[BKW96] Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden. Automated support for seamless interoperability in polylingual software systems. In David Garlan, editor, *SIGSOFT '96: Proceedings of the Fourth ACM Conference on the Foundations of Software Engineering*, pages 147–155, San Francisco, October 1996. Published as *SIGSOFT Notes* 21(6), November 1996.

[Bro94] Kraig Brockschmidt. Ole integration technologies technical overview. World Wide Web URL http://www.microsoft.com/TechNet/technol/ole/ddjole.htm, October 1994. Adapted from an article appearing in Dr. Dobbs Journal, December 1994.

[BRW97] Daniel J. Barrett, John V.E. Ridgway, and Jack C. Wileden. Assuring type safety in polylingual software systems. (submitted).

[BS92] B. W. Boehm and W. L. Scherlis. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference*, pages 63–82, Los Angeles, CA, April 1992.

[CP89] Douglas E. Comer and Larry L. Peterson. Understanding naming in distributed systems. *Distributed Computing*, 3(2):51–60, May 1989.

[Fra71] A.G. Fraser. On the meaning of names in programming systems. *Communications of the ACM*, 14(6):409–416, June 1971.

[GM88] Y. Gurevich and J. Morris. Algebraic operational semantics and Modula-2. In E. Börger, H. Kleine Büning, and M.M. Richter, editors, *First Workshop on Computer Science Logic*, number 329 in Lecture Notes In Computer Science, pages 81–101, 1988.

[Gur88] Yuri Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[HM95] Jim Huggins and Raghu Mani. The evolving algebra interpreter, version 2.0. University of Michigan, 1995.

[Joh71] J.B. Johnston. The contour model of block structured processes. *SIGPLAN Notices*, 6(2):55–82, June 1971.

[JSS95] Bill Janssen, Denis Severson, and Mike Spreitzer. *ILU Reference Manual*. Xerox Corporation, March 1995. Documents ILU version 1.8.

[Kap96] Alan Kaplan. *Name Management: Models, Mechanisms and Applications*. PhD thesis, University of Massachusetts, Amherst, MA, May 1996.

[KW93] Alan Kaplan and Jack C. Wileden. Name management and object technology for advanced software. In *International Symposium on Object Technologies for Advanced Software*, number 742 in Lecture Notes in Computer Science, pages 371–392, Kanazawa, Japan, November 1993.

[KW94] Alan Kaplan and Jack Wileden. Conch: Experimenting with enhanced name management for persistent object systems. In *Sixth International Workshop on Persistent Object Systems*, Tarascon, Provence, France, September 1994.

[KW95] Alan Kaplan and Jack C. Wileden. Formalization and application of a unifying model for name management. In *The Third Symposium on the Foundations of Software Engineering*, Washington, D.C., Sep 1995.

[KW96] Alan Kaplan and Jack Wileden. Toward painless polylingual persistence. In *Seventh International Workshop on Persistent Object Systems*, Cape May, NJ, May 1996.

[MABD90] R. Morrison, M.P. Atkinson, A.L. Brown, and A. Dearle. On the classification of binding mechanisms. *Information Processing Letters*, 34(1):51–55, February 1990.

[Mil93] Robin Milner. Elements of interaction. *Communications of the ACM*, 36(1):78–90, January 1993.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[Sal78] J.H. Saltzer. Naming and binding of objects. In *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, chapter 3A, pages 99–208. Spring-Verlag, 1978.

[Siv95] Gowri Sankar Sivaprasad. *Larch/CORBA: Specifying the Behavior of CORBA-IDL Interfaces.* MS thesis, Iowa State University, Ames, IA, December 1995.

[SR96] Arturo J. Sánchez-Ruíz. Type models: Toward unifying concepts for language interoperability. In *Proceedings of the 22nd Latin-AMerican Conference on Informatics*, pages 541–552, Bogotá, Columbia, June 1996.

[Sto77] J.E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[WBT92] David L. Wells, Jose A. Blakely, and Craig W. Thompson. Architecture of an open object-oriented management system. *IEEE Computer*, 25(10):74–82, October 1992.

[WCW88] A.L. Wolf, L.A. Clarke, and J.C. Wileden. A model of visibility control. *IEEE Transactions on Software Engineering*, 14(4):512–520, April 1988.

[WWRT91] Jack C. Wileden, Alexader L. Wolf, William R. Rosenblatt, and Peri L. Tarr. Specification level interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.

[ZW95a] Amy M. Zaremski and Jeannette M. Wing. Signature matching, a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2), April 1995.

[ZW95b] Amy M. Zaremski and Jeannette M. Wing. Specification matching of software components. In *The Third Symposium on the Foundations of Software Engineering*, pages 6–17, Washington, D.C., Oct 1995.

5 Biography

Jack C. Wileden received the A.B. degree in mathematics and the M.S. and Ph.D. degrees in computer and communications sciences from the University of Michigan, Ann Arbor. He is a Professor in the Department of Computer Science at the University of Massachusetts at Amherst and Director of the Convergent Computing Systems Laboratory there. His current research interests center on tools and techniques supporting seamless integration of advanced capabilities into computing systems. Recent projects in his laboratory have focused on object management topics, including persistent object systems and name management, and on interoperability support for multilingual programming. Professor Wileden's previous research efforts were in the areas of software development environments and tools, notably analysis and debugging tools applicable to concurrent software systems. His research has been supported by DARPA, NSF and ONR and he is the author of more than sixty papers in journals and refereed conferences. With Dr. Alan Kaplan, he presented a half-day tutorial on "Software Interoperability: Principles and Practice" at the 1997 International Conference on Software Engineering.