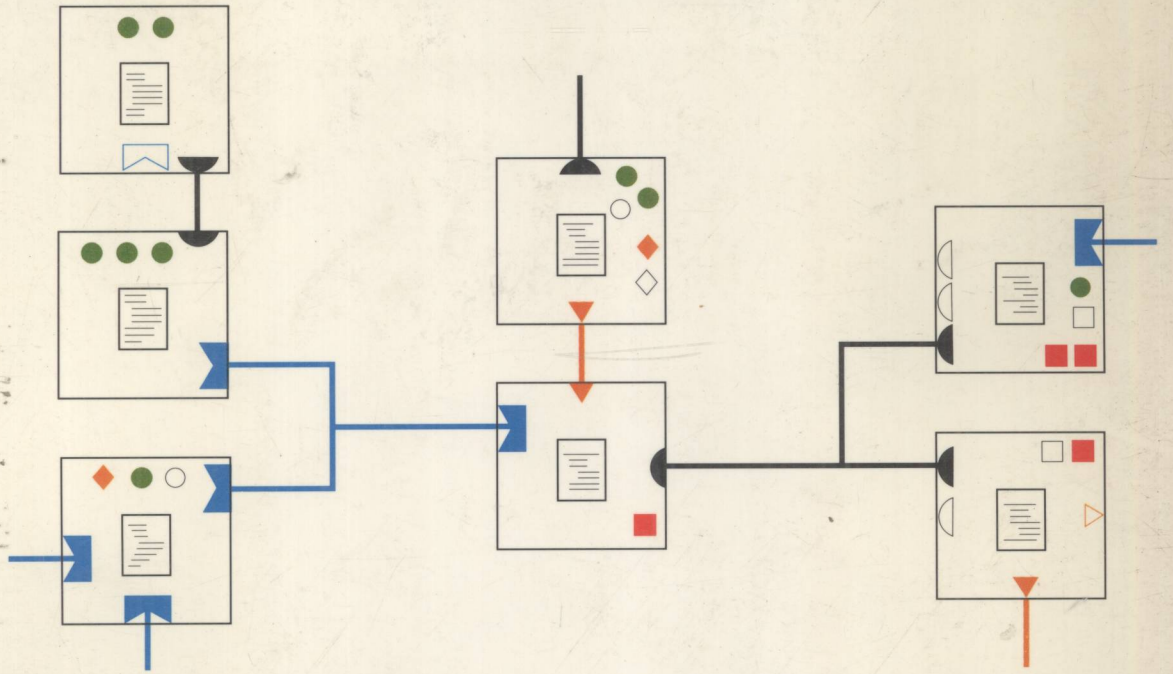
Foreword by BRUCE SHRIVER

HERMES

A Language for Distributed Computing



ROBERT E. STROM
DAVID F. BACON / ARTHUR P. GOLDBERG
ANDY LOWRY / DANIEL M. YELLIN
SHAULA ALEXANDER YEMINI

Prentice Hall Series in Innovative Technology

Hermes

A Language for Distributed Computing

Robert E. Strom

David F. Bacon Arthur P. Goldberg Andy Lowry
Daniel M. Yellin Shaula Alexander Yemini

IBM T. J. Watson Research Center

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

Editorial/production supervision: *Mary P. Rottino*
Cover design: *Wanda Lubelska, Designs*
Manufacturing buyers: *Kelly Behr and Susan Brunke*
Acquisitions editor: *Paul Becker*

A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

The publisher offers discounts on this book when ordered in bulk quantities. For more information, write:

Special Sales/College Marketing
College Technical and Reference Division
Prentice Hall
Englewood Cliffs, New Jersey 07632

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-389537-8

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Prentice Hall Series In Innovative Technology

Dennis R. Allison, David J. Farber, and Bruce D. Shriver *Series Advisors*

Johnson *Superscalar Microprocessor Design*
Kane *MIPS RISC Architecture*
Rose *The Open Book: A Practical Perspective on OSI*
Rose *The Simple Book: An Introduction to Management of TCP/IP-based internets*
Shapiro *A C++ Toolkit*
Slater *Microprocessor-Based Design*
Strom, et al. *Hermes: A Language for Distributed Computing*
Wirfs-Brock, Wilkerson, and Weiner *Designing Object-Oriented Software*

Foreword

Distributed computing is significantly changing the computer systems landscape. Users require increasingly complex and sophisticated systems that include multiple communicating applications distributed over corporate, national, and international networks. Many factors contribute to the accelerated growth of distributed computing and the resulting computing paradigm shift—performance, economics, the need to interconnect previously standalone systems, and the geographical dispersion of many organizations. Modern operating systems, such as Mach, have been developed to support the need for multi-application systems.

Most programming languages, however, are still designed for standalone applications. Therefore, multi-application systems designers and developers are forced to interact with the operating system. They must deal not only with programming language concepts such as typed variables and procedures, but also with communications and operating system concepts such as memory regions, address spaces, and connections. They must understand how variables are represented in the memory of each system, the multitasking, synchronization, and communications models of the different operating systems and their programming interface, how to lay out processes and data to optimize performance, and how to recover from systems and communications failures. The resulting programs not only require special expertise to produce but are also lengthy, difficult to maintain, and much too closely tied to a particular system environment to be portable.

The Hermes project, led by Rob Strom at IBM's T. J. Watson Research Center, has made substantial progress toward fulfilling a vision of a simple distributed computational model and a programming language, compiler, and integrated tool set supporting this model. Their work began in the early 1980's and led to the development of the Network Implementation Language (NIL) in 1982. They have made important contributions to the discipline of designing and implementing complex, distributed systems as well as to programming language theory and practice. Among their impressive results are: the development of the typestate mechanism; the efficient, transparent recovery technique called optimistic recovery; and the development of new approaches to program reuse and optimization. Such contribu-

tions, manifested in real implementations as well as in theory, have added to our understanding of how to create more reliable and portable programs.

The Hermes language offers a novel approach to controlling the complexity of distributed programming. It incorporates a computational model integrating computation, communication, and system configuration within a simple high level language. Implementations of Hermes map this high level model down to the specific features available on the target hardware and operating system and include optimizing transformations to make more efficient use of system resources. The burden of dealing with the details of particular systems and of optimizing performance is thus shifted from the programmer to the designer of the compiler and run-time environment.

The Hermes work has been guided throughout by some well-known and widely accepted principles of language design, including modularity, abstraction, and orthogonality. Part of what differentiates Hermes from other programming languages is the disciplined and uncompromising manner in which these principles have been applied. Whenever conflicting language and system design issues were encountered, the Hermes team opted for adhering to these fundamental principles.

For example, modularity is the notion that programs should be constructed by composing modules that interact in well-defined and easily understood ways. While many languages allow programs to be constructed in this manner, Hermes is unique in the extent to which these interfacing requirements are enforced. In fact, it is impossible for one Hermes module to corrupt another, even without address space boundaries or other firewalls commonly used to provide this protection.

As another example, the notions of abstraction and information hiding in Hermes are addressed not only by providing the programmer with the means to create abstractions, but also by adhering strictly to the ideal that details of machines, operating systems, and operating environment should be of no concern to the programmer. For instance, in Hermes, communication is performed by making calls and passing typed objects, not by building session or datagram connections, passing buffers and RPC handles, and handling flow control and timeouts.

The Hermes approach advances compiler technology to produce efficient

code for complex architectures and the run-time environments provided by different operating systems. Many of these problems have yet to be addressed fully; nevertheless, Hermes has much to offer from a practical perspective. Programming teams will find that the strict checking of module interfaces will greatly ease the task of integrating separately written program modules. Individual programmers will find that the high-level abstractions offered in Hermes can greatly simplify many of their programming tasks. Program development time will be reduced because of the many type and typestate errors detected by the Hermes compiler—errors that in other languages lead to run-time errors like memory violations and are often extremely difficult to track down.

This book is a combination tutorial and reference manual, full of rich insights about distributed systems software technology. It integrates many contributions of the Hermes team in the areas of language design, program analysis, compilation, and distributed computing and illustrates how the application of what they have learned can yield startling benefits for the programmers who design, implement, and maintain the growing body of distributed applications code. It is definitely worthwhile reading for those interested in complex systems, be they designers, developers, teachers, students, or researchers.

Bruce Shriver
Professor, University of Southwestern Louisiana

Preface

This book is a tutorial and a reference manual for the Hermes programming language.

What is Hermes

Hermes reflects our vision of how the software systems of the future should be developed. In our view, stand-alone applications will evolve towards systems composed of interacting applications. System-dependent, "low-level hacking" approaches to building complex software systems will be replaced by newer technologies in which details of operating system, communications protocols, and machine architecture are hidden from the programmer. Compiler techniques will improve to the point where programmers will not explicitly manage the distinctions between main memory and secondary storage, between local and remote procedures, between shared memory and message communications. Programmers will program to a simplified abstract machine; compilers will map this abstraction to different and changing physical configurations. Programmers will be concerned with functionality and modularity; compilers will be concerned with architecture and performance.

Hermes is an initial realization of this vision. It is a secure, representation-independent, programming language. It supports a software engineering methodology for developing complex systems by incorporating a simple, powerful dynamic distributed process model within a high-level language.

Hermes has a number of fundamental principles. Many of them appear to be widely accepted truisms. However, Hermes applies these principles more extensively than other languages in common use today.

- **Modularity:** *Systems should be built from small, autonomously executing modules whose interfaces can be written down and enforced.* Hermes modules are processes, encapsulating both data and an associated program. Passive objects are a special case of processes: their

program consists of a single loop waiting for calls. A running system will contain many (hundreds or thousands) of autonomously executing processes. Because Hermes must support a multi-application environment, we enforce protection (what Hoare calls *security*[Hoa81]) at the granularity of a module. In other languages, unpredictable side effects can occur as a result of unchecked "erroneous" programs (e.g. accessing an undefined value in Ada) or as a result of misuse of "unsafe" programming constructs (e.g. untraced references in Modula-3[CDG+89]). In these languages, such side effects are tolerated because it is assumed that each separate application has been put in a separate "address space" by the operating system. In Hermes, there is no unsafe subset. A new compile-time checking technique called *typestate* checking guarantees that each module is protected as if it were in its own address space, even though the implementation can put many applications' processes in the same address space. Thus Hermes provides a finer granularity of protection within an application and more efficient communication across applications.

- **Uniformity:** *Modules interact the same way whether they are local or remote, and whether or not they belong to the same application.* In Hermes, modules interact by sending messages or by making calls. There is no sharing, and no aliasing. We believe that the possibility for interference in shared memory systems inhibits modularity and analyzability. Another consequence of uniformity in Hermes is that there is one kind of module (process), as opposed to Ada's tasks, packages and procedures.

- **Abstraction:** *The high-level language should hide as many low-level details of the underlying implementation as possible. Programs should be easy for people to read and for compilers to check and optimize.* Although many people pay lip-service to this principle, it conflicts with another widely accepted principle—that of *performance transparency.* Performance transparency means that the cost of the underlying implementation can be estimated by looking at the source code. Performance transparency implies, for example, that the programmer should know (and be able to control) whether an n-byte data structure is being passed by reference (constant-time cost) or by value (order n cost). This principle underlies the design of C, Pascal, and the Modula family of languages. *In Hermes, we intentionally give up performance transparency in favor of abstraction.* We do this

for reasons of simplicity, portability, and efficiency. Hermes programmers see a simpler model—serial processes, persistent variables, and reliable communication. They do not see the machine architecture, the storage hierarchy, the physical representation of data, the operating system, or the communications systems. The Hermes programmer gives up the ability to write an exotic program with multithreading, aliased variables, and other properties which make programs hard to understand. In exchange, the compiler gains the ability to detect the possibility of applying an optimization which may introduce aliasing and multithreading under the covers. Our conjecture is that by providing a simpler high-level model in which programmers may not write "tricky" code, and a library of optimizations which reintroduce the "tricks" into the implementation, we will improve reliability, enable reuse of optimizations in any program using the optimized constructs, and do as well or better in efficiency.

Who should read this book

The Hermes language should be interesting to system designers, system developers, computer science researchers, and teachers of programming language design, program analysis, software engineering, operating systems and distributed systems.

- *System developers* will find that hiding the underlying system technology allows them to write much shorter and simpler programs, to get them right faster, and to port them without change across different environments. They will find the modularity provided by processes, and the separately compiled type and typestate-checked interfaces valuable for managing large programming efforts. They will appreciate the automatic error detection provided by typestate checking and the fact that they will no longer need to examine core dumps.

- *System designers* will find that they can directly implement and execute layered architectures in Hermes by using a process for each component and an (input, output) port pair for for each inter-process connection, without paying a performance penalty.

- *Teachers of distributed systems* will find in Hermes a clean system-independent model for teaching fundamental operating systems concepts such as processes, capabilities, inter-process communication, concurrency and synchronization, separation of policy from mechanism, and others. Complex system services, such as window managers, spoolers, and logon shells can be coded very compactly in Hermes.

- *Computer science teachers and researchers* interested in program analysis and optimization techniques will find that the high level of abstraction of Hermes provides an excellent framework for exploring of novel program optimization techniques that exploit distributed configurations: parallelization, replication of data and processes, remote evaluation, process migration, and others. Those interested in software reuse will find in the program datatype a fertile ground for studying and unifying techniques such as generics and inheritance, as well as exploring new approaches to reuse: reusing arbitrary program fragments, generating programs from program templates or from other programs by substituting syntactically and semantically well-formed program components, generating debugger drivers, and others.

How can I get a copy of Hermes?

Currently we will give Hermes for free to anyone who wants to use it for non-commercial purposes. Mail the tear-out postcard at the back of this book to us and we'll tell you how to obtain Hermes. Currently, the system is implemented on several UNIX platforms, including: Sun 3 and 4, NeXT machines, and IBM RT PC and Risc System/6000. Future plans include porting Hermes to run on Mach. Other related software systems include the Optimistic Recovery Layer we are developing on Mach to provide transparent recovery to general applications without the need for transactions, and the Concert/C extensions to the C programming language for supporting a Hermes-like process model in C.

Organization of this Book

This book contains a tutorial, a reference manual, and appendices.

The tutorial introduces you to Hermes by guiding you through a set of examples. We begin with a simple program which outputs "Hello world," and continue through more complex examples, ending with a window system. We then discuss additional useful features of Hermes, such as typestate analysis. We assume that you have some experience writing application programs in a procedural language, such as C, Pascal, or Ada. We will highlight the differences between programming in Hermes and programming in other languages, so you will get a feeling for "idiomatic Hermes." At the end of the tutorial, you will know the basic vocabulary of Hermes. You will be able to write some Hermes programs by imitating the examples. You will be able to compare Hermes to other languages. But you will not know the precise rules of Hermes—these are covered in the reference manual.

The reference manual is more formal than the tutorial. We also give examples in the reference manual, but with a different purpose. The examples in the reference manual illustrate the language rules. They highlight the difference between legal and illegal programs rather than illustrate "typical" programs.

The appendices are the most formal. They contain the rules of Hermes in tabular form. They are produced from the same machine-readable files that are used to produce the compiler itself.

This document does not describe how to use any of the existing Hermes implementations, nor does it describe their idiosyncrasies. Such information is to be found in the *Hermes Users Guide* distributed with the implementation.

Acknowledgments

Hermes derives from earlier work on the NIL language at IBM's T.J. Watson Research Center. The success of the NIL prototype made it possible to continue our work with Hermes. We acknowledge the contribution of Francis Parr to the early language design of NIL, and to the first experimental

prototype. Nagui Halim and John Pershing, Jr. were responsible for much of the implementation of the NIL compiler and run-time environment. They also wrote significant components of the communication applications used to prove the practicality of NIL. Mike Conner provided important managerial and moral support. Other contributors to the NIL prototype include Dan Milch and Jim McInerney.

Van Nguyen made important contributions to the Hermes implementation and to earlier drafts of the manual. Joel Auslander implemented much of the Hermes external interface and contributed useful programming tools.

We thank Tom Marlowe for his numerous, extensive, and timely comments on our manuscript. Other readers who contributed valuable comments include German Goldszmidt, Willard Korfhage, Jim Russell, and Peter Wegner.

Finally, we thank IBM and our colleagues for their support and for providing the intellectual environment from which we profit in untold ways each day.

Contents