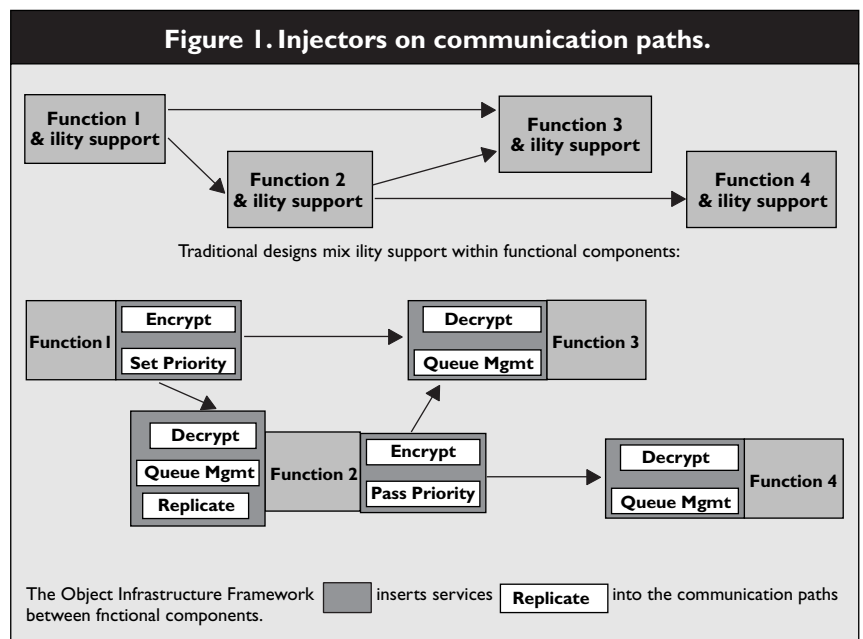


INSERTING ILITIES BY CONTROLLING COMMUNICATIONS

FOR MANY APPLICATIONS, MOST code is not devoted to implementing the desired input-output behavior but to providing system-wide properties like reliability, availability, responsiveness, performance, security, and manageability. We call such qualities *ilities*. This article describes a system that enables a more complete separation of *ility* implementations from functional components, allowing *ilities* to be developed, maintained, and modified with minimal impact on functional implementations.

Ilities can seldom be entirely implemented simply as discrete services. For example, many replication algorithms require logging and distributed update on every object modification. Similarly, performance, security, and manageability enhancements demand systematic and widespread code changes, complicating a clean design. While object-oriented design and programming has provided effective ways to modularize functional requirements into separately maintainable components, it has been less successful in enabling programmers to modularize code devoted to *ilities*. Object orientation does not provide programming structures that allow *ilities* and functionality to evolve independently over the software life cycle.

Separating *ility* support from functional components becomes significantly more important and complex in distributed applications. Distributed applications typically have more stringent *ility* requirements and need more complex *ility* algorithms. This article defines an approach that supple-



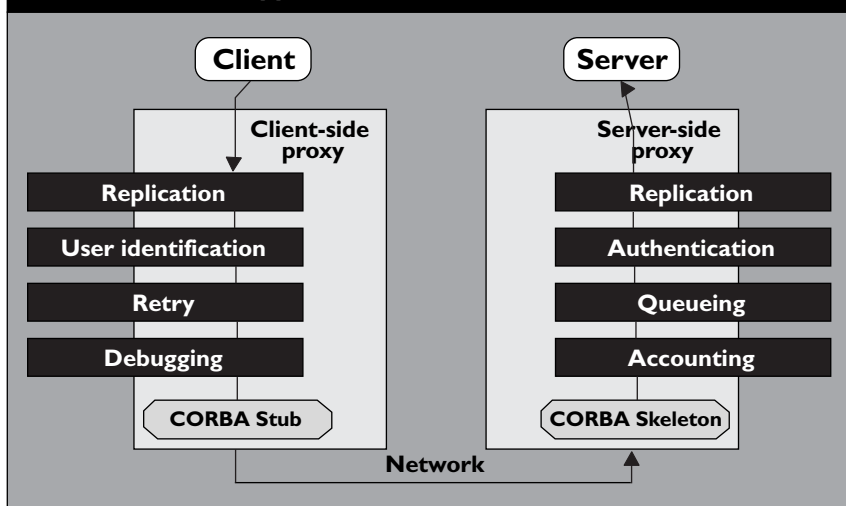
ments standard object-oriented methods with a general mechanism for injecting *ility* implementations into the communications between functional components. Algorithms that support *ilities* are separated from functional components but may be invoked whenever functional components communicate. This allows *ilities* and functionality to be modified and maintained with minimal impact on each other.

Achieving *Ilities* by Controlling Communication

Our research integrates the following key ideas:

- **Intercepting communications.** Our primary claim is that *ilities* can be achieved by intercepting and manipulating communications among functional components and by invoking appropriate “services” on all inter-component communications.

Figure 2. OIF inserts injectors between the application and the network.



- **Discrete injectors.** Our communication interceptors are first class objects, discrete components that have (object) identity and can be sequenced, combined, and treated uniformly by utilities. We call them injectors. In a distributed system, an ability may require injecting behavior on both the client and the server. For instance, security requires authenticating on the server credentials generated on the client. Figure 1 illustrates injectors on communication paths between components.
- **Injection by object/method.** Each instance and each method on that object can have a distinct sequence of injectors.
- **Dynamic injection.** Dynamic configuration allows us to place debugging and monitoring probes in running applications and to create software that detects its own obsolescence and updates itself. There is a tradeoff, however, since security and manageability require rigorous configuration control over injector changes.
- **Annotations.** Injectors need to communicate among themselves. For example, the authentication injector needs to know the identity and credentials of a service requestor. Our solution is to provide a general mechanism for annotating communications with meta-information. Injectors are capable of reading and modifying the annotations of requests (and reading and modifying the request arguments and target function name).
- **Thread contexts.** Our goal is to keep the injection mechanism invisible to the functional components. However, sometimes clients and servers need to communicate with injectors. For example, a quality-of-service injector may want to process

requests in order of their priority, but the only reasonable source of request priority is the client application. While some annotations must originate from the functional applications, separation of concerns would be destroyed if functional components have to be aware of all annotations. We make annotations largely transparent to functional components by providing an “alternative communication channel.” Each client and server thread has its own set of annotations, the thread context. The system arranges to copy annotations among the

client’s thread context, the request, and the server’s thread context.

- **High-level specification compiler.** There is a large conceptual distance between abstractilities and discrete sequences of injectors. To span this gap, we have created Pragma, a compiler that takes a high-level specification of desired properties and ways to achieve these properties, and maps that specification to an appropriate set of injector initializations.

Object Infrastructure Framework

We have illustrated these ideas by defining an architecture (the Object Infrastructure Framework or OIF), instantiating that architecture for a particular environment (CORBA®/Java™), and creating several validating applications within that framework [2,8].

Current technology for building distributed, component-based applications uses sockets, messages, remote procedure calls such as DCE™, or Object Request Brokers (ORBs, such as CORBA, JavaRMI™, and DCOM). Without too much loss of generality, we focus on ORB frameworks and use CORBA as our exemplar. CORBA implements distribution by building proxy objects on both the client (caller environment) and server (called environment) to represent a particular server object. The client-side proxy (or stub) is responsible for marshaling a client request into a form that can be transmitted over the network; the server-side proxy (or skeleton) demarshals the request into native data structures for the server to process. ORB technology provides object location transparency and hides the details of marshaling and communication protocols. What it doesn’t do is handle ility concerns like partial failures, security, and quality

of service. ORBs such as CORBA and Enterprise Java Beans™ provide different discrete mechanisms for particular ility issues, but such mechanisms typically provide only a finite number of choices for the application architect, and require a good understanding and diligent application of the mechanism by the application programmer.

Injectors. OIF's key implementation idea is to modify ORB proxies so that: (1) each stores a map from proxy methods to a sequence of injectors, and (2) in the proxy processing for a given method, that sequence is invoked between the application and marshaling. The action method of the injector gets an object rep-

in this list, providing it the rest of the list as its continuation.) This has the advantages of allowing the injector stack to naturally catch exceptions, and permitting an injector to forgo or transform the continuation sequence. Our authentication injector illustrates the former advantage. A server-side authentication injector dissatisfied with a request's credentials raises an exception that a client-side injector catches. The client side injector interrogates the user and reinvokes the request with the additional annotation.

Similarly, when methods return static values and do not have side effects, an injector can cache values returned from previous calls. When a request is already in the cache, the caching injector can omit the remote call and return the local value. This has the effect of doing "objects by value" for selected parts of a remote object.

Annotations. Annotations provide a language for applications and injectors to communicate regarding requests. That is, they are a meta-language for statements about requests and the processing state. Annotations can express notions like "This request is to be done at high priority," "Here are the user's credentials," and "Here is the cyberwallet to pay for this request." Annotations can be associated with both requests (request annotations) and processing threads (thread contexts).

OIF annotations are name-value pairs. The names are strings and the values are CORBA ANY types, allowing object references

representing the request. It can interrogate and modify that object for the request's target, method name, arguments, and annotations. Being code, it can perform arbitrary other operations, such as invoking methods on other (remote) objects and changing its local static state. Figure 2 illustrates CORBA proxies extended with injectors.

Injector processing is in "continuation style," meaning injectors invoke the rest of the injector sequence between their "before" and "after" behaviors. (With the continuation pattern, one of the parameters of a routine is a representation of "the rest of the work to be done" after this routine has finished. In OIF, the continuation is represented as a list of injectors, and invoking the continuation is simply calling the first injector

as annotation values. This requires annotation readers and writers to have an implicit agreement about annotation types. Object references in annotations are used for patterns such as continuations ("Send the results of this computation to X") and agencies ("Y can verify my identity"), but not strings; encoding object references as strings would burden the recipient with demarshaling. The framework defines certain common annotations, including session identification, request priority, sending and due dates, version and configuration, cyber wallet, public key, sender identity, and conversational thread. Programs can rely on the common meanings of these annotations. Applications and injectors can create other annotations. Annotations can be implemented as hash tables or property

Figure 3. Propagating annotations.

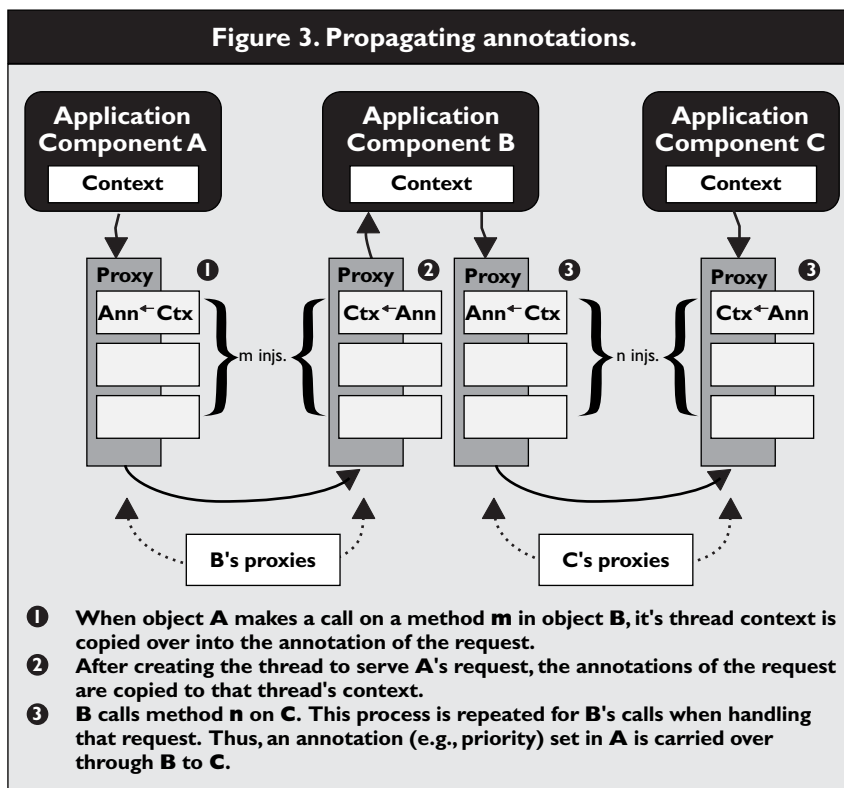
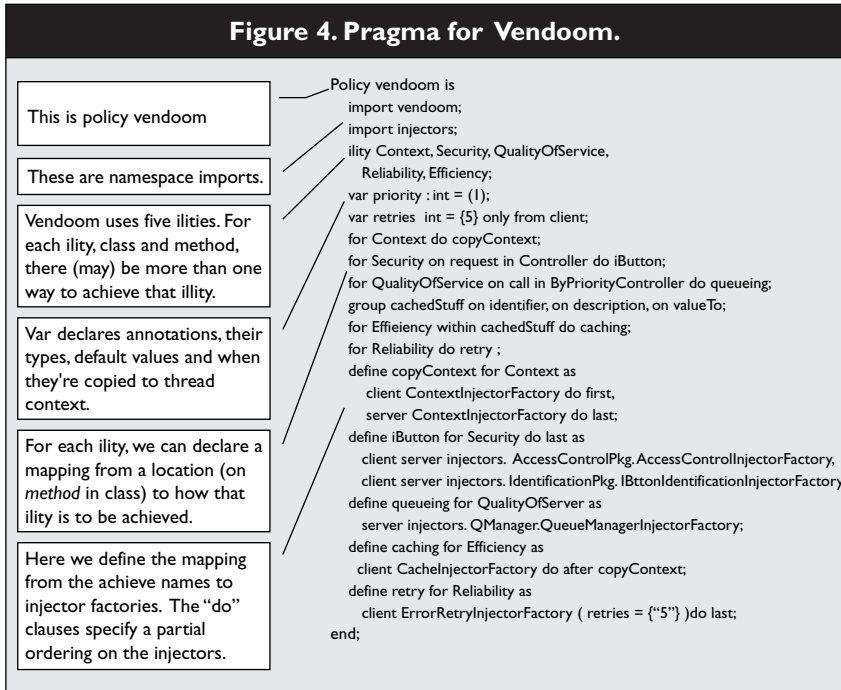


Figure 4. Pragma for Vendoom.



lists. OIF proxies marshal and demarshal request annotations like ordinary procedure arguments.

Requiring injectors to declare the annotations they read and write, and enforcing those declarations, can improve security. We may feel safer using an injector that is restricted to only read the due dates of messages rather than one that can alter user identification or method arguments.

Thread contexts, (annotations associated with processing threads), allow applications to communicate with injectors. On each call, the framework copies the client thread's annotations to the annotations of the nascent request. On the server side, the framework builds the context of the service thread from the request's annotations. On return, the framework copies the server's context to the annotations of the response and then back to update the context of the original client. This scheme has the feature of propagating context through a chain of calls: client A's call of B at priority x becomes B's context's priority of x. B's request of C (in furtherance of A's call) goes out with priority x. Figure 3 illustrates this pattern.

Thread contexts have the advantage of permitting client/injector communication without modifying the application interfaces. They have two disadvantages: newly spawned threads need to copy or share the context of their parents, and there is no primitive linguistic mechanism for neatly "block structuring" a change to a thread's context-allowing, for example, a thread to simply timeshare among tasks.

Declarations can control annotation copying. For example, the number of times the (client) retry-on-fail-

ure injector retries is not sent downstream. Similarly, we do not want a server to be able to update a client's user identification. The default behavior is to copy, enabling creating new annotations without modifying existing application code.

Pragma. Our high-level goal is to take functional code, ility specifications, and reusable ility service implementations and weave them together into the actual system code. Ideally, we would like to be able to press the "application: be secure," key, and, lo and behold, the application code is pervasively modified as necessary. That said, we have the sad task of reminding the reader of the dearth of magic in the world. Iilities must be implemented by invoking actual ser-

vices. Saying you want security does not create security. Rather, you have to define security, as, for example encrypting all communications using { 64 | 128 | 7 } bit { DES | RSA | ROT-13 }, checking the user's { password | fingerprints | DNA } for { every | occasional } access to { all | sensitive } methods, recognizing intrusions { from strange sites | trying a series of passwords | asking too many questions }, keeping track of privileges by { proximity | job function | dynamic agreements }, and so forth. We need the ility architect both to have implementations of the appropriate algorithms (injectors that actually do that work), and to specify where each set of injectors is applied.

Pragma posits a two-level structure to achieve these goals. The architect defines:

1. A number of ilitys (symbolic names like "reliability.")
2. Methods (actions), to achieve each ility. For example, the ility "security" might have an action "high security" that authenticates through fingerprints and includes extensive monitoring and intrusion detection, while the action "low security" might require only passwords and limited monitoring.
3. A map from the actions to locations in the program. A location can be on a particular method in the implementations of a particular interface, on all the methods of a particular interface, on all the methods of a given name, or everywhere. These definitions can also include assertions about injector ordering.

Pragma also includes constructs for declaring annotations (including their type, default values, and copying context) and for constraining the use of injectors. We support the latter in two ways: an assertion mechanism allows an injector to preclude or demand another, and a cascade mechanism allows the successive refinement of policies within an organization. More specifically, a policy (collection of Pragma statements) may import other policies. A policy may also specify a set of alternatives for an *ility*. Policies that import such restrictions can choose among (or further restrict) this set, but may not offer new choices. Thus, an enterprise architect may define three acceptable security alternatives, an application suite architect may restrict these to two, and the *ility* architect for a particular program may choose to use only one.

The Pragma compiler takes as input both a policy and the application IDL™, and generates injector initializations and annotation declarations. Pragma, for each method, interface, and *ility*, finds the “most specific” way of doing that *ility* on that method of that interface. (Subinterfaces are more specific than superinterfaces; method-mentions more specific than not, and, arbitrarily, method-mentions more specific than interfaces.) It then orders the actions on that *_interface*, *method_ pair* and outputs the results as data to the initialization mechanism. Pragma flags as errors combinations that violate constraints. Figure 4 shows the Pragma for Vendoom [8], a demonstration system developed using OIF.

Applied Ilties

Our work has been driven by demonstrating these ideas in a pair of prototypical applications. DisDev [2] implements a distributed repository and illustrates the use of injectors to achieve reliability through replication. Vendoom implements a simulation of a distributed, competitive network management application. It uses injectors to achieve quality of service (such as real-time performance), manageability, and security [8]. We review the lessons learned in pursuing these *ilties* in our framework.

Reliability. Our primary experiments in supporting reliability have centered on injecting replication algorithms into DisDev, a document management application. Replication algorithms typically send copies of messages to replicants. That is, if operation *f* is invoked on *x* and *y*, (and *f* mutates the application state), all replicants need to be aware of this action. Our work suggests this is more easily supported if *x* and *y* are symbolic, rather than pointers into a replicant’s memory space.

Other reliability injectors we have demonstrated include the retry injector, which repeats attempts that

time out or otherwise fail, and the rebind injector, which seeks alternative servers under the recognized failure of one.

Transactions are a reliability mechanism that illustrates the limits of this approach. Transactions require application objects that can start and end transactions and rollback on failure. If the application objects have these interfaces, injectors can be used to coordinate their invocation. For example, transaction identity is a straightforward application of request annotations. Sadly, however, transactions cannot be transparently achieved by injection to objects that lack them.

Quality of service. By quality of service we include a variety of requirements for getting things done within time constraints. The real-time community recognizes two varieties of real-time systems: hard real-time and soft real-time. A correct hard real-time system must complete all tasks by their deadlines. Soft real-time systems seek to allocate resources to more important tasks. Hard real-time requires cooperation throughout the processing chain (for example, in the underlying network), since the promise of particular service can be abrogated in too many places. (Doug Schmidt’s work on real-time CORBA ORBs [11] illustrates this point: commercial CORBA ORBs, built without constant real-time mindfulness, conceal FIFO queues and exhibit anti-real-time behavior.)

Soft real-time is amenable to several communication control tactics. These include using queue control to identify the most worthwhile thing to do next [8], calling the underlying system’s quality of service primitives, using side-door mechanisms to efficiently transport large quantities of data, and choosing among multiple problem solving approaches. We have demonstrated the first of these tactics in Vendoom. All except the last are easily done with injectors. If the application supplies the alternative problem solving methods (either by replicating the problem solving sites, allowing load balancing, or providing genuinely different algorithms), the communication control mechanism can apply the most effective problem solvers. Injectors, as stateful objects, can determine the best message target using tactics such as learning from historical experience and consulting traffic-reporting agencies.

Manageability. We take a network control perspective on manageability, dividing manageability into five elements: performance measurement, accounting, failure analysis, intrusion detection, and configuration management. The first four are amenable to generating events in relevant circumstances and directing those events to the appropriate recipients. For example, in Vendoom we have used injectors to publish events to update graphic displays, report payment

data, and debug the application. In general, to the extent that the semantics of interesting events are tied to communication acts, such as when a micro-payment is processed each time a service is called, or the trace of inter-component messages is sent to a system's debugger, the events can be realized through external communication controls. This technique is inadequate when the interesting actions happen completely within the application components. Examples of such internal activity include when payment is directly proportional to the number of records accessed by a database service, or debugging occurs wholly within a component..

We have also designed a configuration management injector that dynamically tests for incompatible versions and automatically updates stale configurations. Such management can be done only for clients and servers that provide the appropriate interfaces.

Security. Security, at least in a software sense, is primarily a combination of access control, intrusion detection, authentication, and encryption. Controlling the communication process allows us to encrypt communications, reliably send user authentication from client to server (and pass it along to dependent requests), and check the access rights of requests. All this is independent of the actual application code. (However, we may only be able to encrypt the message data, not its headers. Similarly, encrypting object references may confuse the marshaling code. In general, encryption is better done after marshaling.) Watching communications provides a locus for detecting intrusion events [5], although not, of course, specifying the actual algorithms for recognizing an intrusion. We have illustrated security in Vendoom with injectors that perform access control and, (by checking the user's Java ring), authentication.

Can such mechanisms yield security? Somewhat. Such mechanisms reflect common notions of security, but cannot prevent hazards such as subverting a system's personnel, tapping communication lines, brute-force cracking of encryption codes, or components that cheat. Magic has its limits.

Related work

We have described a mechanism for separately specifying system-wide concerns in a component-based programming system and then weaving the code handling those concerns into a working application. This is the theme of Aspect-Oriented Programming (AOP). OIF is an instance of AOP, and brings to AOP a particularly elegant division of responsibilities. Key work on AOP includes Harrison and Ossher's Subject-Oriented Programming [6] which extends OOP to handle different subjective perspec-

tives; Aksit and Tekinerdogan's message filters [1], which, like OIF, reify communication interceptors; Lieberherr's Adaptive programming [9], which proposed writing traversal strategies against partial specifications; and Kiczales and Lopes' [7] language for separate specifications of aspects, which effectively performs mixins at the source-code language level. Czarnecki and Eisenecker's book [3] includes a good survey of AOP technology.

The idea of intercepting communications has occurred several times in the history of computer science. Perhaps the earliest examples were in Lisp: the Interlisp advice mechanism and mix-ins of MacLisp.

It is common to tackle ility concerns by providing a framework with specific choices about those concerns. Examples include transaction monitors like Encina™ and Tuxedo® and distributed frameworks like Enterprise Java Beans and CORBA. It is worth noting that the CORBA security specification and many commercial CORBA implementations are emerging with some form of user-defined filter mechanism on communications. While these mechanisms are not as general as OIF, our work can be understood as a methodology for using CORBA filters.

The use of a separate specification language for creating filters parallels the work at BBN on quality of service [10], where the IDL-like Quality Description Language is woven with IDL to affect system performance.

Conclusion

Elsewhere we argued that requirements come in four varieties: functional requirements that exhibit the primary semantic behavior of a system and are typically locally realized, systematic requirements that can be achieved by “doing the right thing” consistently throughout the program, combinatoric requirements that are computationally intractable expressions of overall system behavior, and aesthetic requirements that express non-computable qualities of the system [4]. Conventional development does a good job of supporting the first of these, and the last two are difficult to automate in any case.

We believe the mechanisms described in this article—injectors on communication, annotations, and high-level specification languages—are a comprehensive approach to satisfying systematic requirements. While not all systematic algorithms can be implemented without application cooperation, we have demonstrated a technology for taking a high-level expression of desired systematic requirements and automatically propagating this behavior to the components of a distributed system. We believe our results generalize to other contexts. **G**

REFERENCES

1. Aksit, M. and Tekinerdogan, B. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. *AOP '98 workshop position paper* (1998), [http://www.trese.cs.utwente.nl/ Docs/Tresepapers/FilterAspects.html](http://www.trese.cs.utwente.nl/Docs/Tresepapers/FilterAspects.html).
2. Barrett, S. and Foster, P. Turning Java components into CORBA components with replication. *OMG-DARPA-MCC Workshop on Compositional Software Architectures* (Monterey, California, Jan. 1998), <http://www.objs.com/workshops/ws9801/papers/paper067.doc>.
3. Czarnecki, K. and Eisenecker, U. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, 1999.
4. Filman, R. E. Achievingilities. *Workshop on Compositional Software Architectures* (Monterey, California, Jan. 1998), <http://www.objs.com/workshops/ws9801/papers/paper046.doc>.
5. Filman, R. E., and Linden, T. Communicating security agents. *The Fifth IEEE-Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises-International Workshop on Enterprise Security* (Stanford, California, June 1996), 86–91.
6. Harrison, W. and Ossher, H. Subject-Oriented Programming (A critique of pure objects), in *Proc. OOPSLA 93, ACM SIGPLAN Notices 28*, 10 (1993), 411–428.
7. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., and Irwin, J. Aspect-Oriented programming, *Xerox PARC Technical Report* (Feb. 97), SPL97-008 P9710042. <http://www.parc.xerox.com/spl/projects/aop/tr-aop.htm>.
8. Lee, D., and Filman, R. E. Verification of compositional software architectures. *OMG-DARPA-MCC Workshop on Compositional Software Architectures* (Monterey, California, Jan. 1998), <http://www.objs.com/workshops/ws9801/papers/paper096.html>.
9. Lieberherr, K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, 1996.
10. Schantz, R., Bakken, D., Karr, D., Loyall, J., and Zinky, J. Distributed objects with quality of service: an organizing architecture for integrated system properties. *OMG-DARPA-MCC Workshop on Compositional Software Architectures* (Monterey, California, Jan. 1998).
11. Schmidt, D.C., Bector, R., Levine, D.L., Mungee, S., and Parulkar, G. An ORB endsystem architecture for statically scheduled real-time applications. In *Proc. IEEE Workshop on Middleware for Distributed Real-time Systems and Services* (San Francisco, Dec. 1997), 52–60.

We thank Lockheed Martin Corporation; Motorola Incorporated; National Aeronautics and Space Administration, Ames Research Center; Raytheon Company; Office of the Assistant Secretary of Defense for Health Affairs (OASD(HA)), Composite Health Systems (CHCS); and Southwestern Bell Information Services for their financial support.

Encina is a trademark of Transarc Corporation. DCE is a trademark of The Open Software Foundation. Java, Enterprise Java Beans and Java RMI are trademarks of Sun Microsystems, Inc. CORBA is a registered trademark and IDL is a trademark of Object Management Group, Inc. TUXEDO is a registered trademark of Novell, Inc.

ROBERT E. FILMAN (rfilman@arc.nasa.gov) is a senior scientist at the Research Institute for Advanced Computer Science at NASA Ames Research Center, in Moffett Field, California.

STUART BARRETT (stuwork@bigfoot.com) is a computer scientist at Caltech in Austin, Texas.

DIANA D. LEE (ddlee@arc.nasa.gov) is a computer scientist for Science Applications International Corporation at NASA Ames Research Center.

TED LINDEN (linden@computer.org) is independent consultant in Palo Alto, CA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2002 ACM 0002-0782/02/0100 \$5.00