A Framework for Event-Based Software Integration

Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise
Computer Science Department
University of Massachusetts
Amherst, MA 01003

Although event-based software integration is one of the most prevalent approaches to loose integration, no consistent model for describing it exists. As a result, there is no uniform way to discuss event-based integration, compare approaches and implementations, specify new event-based approaches, or match user requirements with the capabilities of event-based integration systems. We attempt to address these shortcomings by specifying a *generic framework for event-based integration*, the EBI framework, that provides a flexible, object-oriented model for discussing and comparing event-based integration approaches. The EBI framework can model dynamic and static specification, composition and decomposition, and can be instantiated to describe the features of most common event-based integration approaches. We demonstrate how to use the framework as a reference model by comparing and contrasting three well-known integration systems: **FIELD**, **Polylith**, and **CORBA**.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering

General Terms: Event-Based Systems, Control Integration, Interoperability

Additional Key Words and Phrases: Reference model, FIELD, Polylith, CORBA

1. INTRODUCTION

A serious concern in the construction of large software systems is *integration*: the process by which multiple software modules (programs, subprograms, collections of subprograms, etc.) are made to cooperate. Approaches to integration range

This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137.
Name: Daniel J. Barrett
Address: barrett@cs.umass.edu
Name: Lori A. Clarke
Address: clarke@cs.umass.edu
Name: Peri L. Tarr
Address: tarr@cs.umass.edu
Name: Alexander E. Wise
Address: wise@cs.umass.edu

from loose, in which modules have little or no knowledge of one another, to tight, in which modules require much knowledge about one another. Loose integration helps reduce the impact on a system when modules are added or changed. Event-based integration, in which modules interact by announcing and responding to occurrences called events, is perhaps the most prevalent loose integration approach — more than fifty event-based integration systems are available today (e.g., [Digital Equipment Corporation et al. 1993; Gerety 1990; SunSoft, Inc. 1993; Patrick, Sr. 1993; Purtilo 1994; Reiss 1990]).

Unfortunately, no precise, consistent model exists for describing event-based integration; in fact, there is not even a consistent vocabulary for discussing it. For example, a set of interacting software modules might be called programs, modules, tools, applications, processes, process groups, clients, objects, information objects, components, tasks, senders/recipients, agents, actors, or function hosts. These software modules may interact via events, messages, announcements, notices, performatives, data, solution information, or requests; and these may be routed by a message server, broadcast server, broadcast message server, server process, manager process, event manager, object request broker, distributor, or software bus.[1] Often there are subtle semantic differences between the concepts that these terms represent, but the differences and similarities are hard to discern.

Since there is no consistent model for describing event-based integration, it is hard to capture one's integration requirements and match them with those offered by various event-based approaches. Thus, it is difficult to choose a suitable approach; and if no existing approach is suitable, it is difficult to specify a new one. It is also difficult to identify the similarities and differences of various event-based approaches in order to support interoperability among them.

In this paper, we attempt to address these shortcomings by specifying a *generic framework for event-based integration*, the EBI framework. This framework is not "yet another" loose integration system; rather, it is a high-level, general, and flexible reference model with several purposes:

—To identify common components found at the heart of event-based software integration and to define a precise and consistent vocabulary for discussing them.

—To serve as a basis for comparison of specific event-based integration mechanisms and, as a consequence, to facilitate interoperability among them.

—To provide insight into what is required for high-level communication between software modules.

1.1 Scope of the Framework

The EBI framework models event-based integration, as found in **FIELD** [Reiss 1990], **SoftBench** [Gerety 1990], **Polylith** [Purtilo 1994], **ISIS** [Birman 1993], **AppleEvents** [Apple Computer, Inc. 1991], **Yeast** [Krishnamurthy and Barghouti 1993], and many other integration systems. We have chosen to concentrate on event-based integration in order to direct attention to this emerging technology and to provide a more focused model. Other related techniques, such as procedure call, shared repository (e.g., ECMA **PCTE** [Boudier et al. 1989], **EAST** [Gautier et al.

[1]Software Bus is a registered trademark of Eureka Software Factory.

1995]), streams (e.g., pipes), and compound documents (e.g., **OLE** [Brockschmidt 1995], **OpenDoc** [MacBride and Susser 1996]) were omitted because they did not fit the focus of our event-based model. Procedure calls (and likewise, object-oriented method invocations) require agreement between caller and callee that is unnecessary for the announcement of events. Shared repository or central database approaches allow data to remain in storage and to be accessed multiple times, whereas events are transitory. Stream-based approaches require a continuous flow of data rather than units of data. Finally, compound document approaches emphasize integration between passive data objects rather than active software modules.

Some other issues of distributed systems, such as reliability and concurrency, are not modeled by the EBI framework. While these issues must be addressed when implementing an event-based integration mechanism, they are not strictly part of the event-based model. Existing research on the comparison of concurrency control models [Chrysanthis and Ramamritham 1990] and of reliability models [Siewiorek and Swarz 1992] can be used to supplement the results provided by the EBI framework.

1.2 Roadmap

We begin in Section 2 by describing related work in event-based software integration. Section 3 gives a brief overview of the EBI framework. Section 4 defines a descriptive type model utilized in the detailed description of the framework, which follows in Section 5. Section 6 demonstrates the EBI framework's effectiveness by describing and comparing three existing integration systems: **FIELD** [Reiss 1990], **Polylith** [Purtilo 1994], and **CORBA** [Digital Equipment Corporation et al. 1993]. Finally, two appendices describe the type model and the **FIELD** case study in detail.

2. RELATED WORK

The concepts underlying event-based integration are found in diverse areas of computer science. Event-based programming is at the heart of countless software applications that wait for user-generated events (e.g., keypresses, mouse clicks) and respond to them. Dozens of software integration systems, such as **FIELD** [Reiss 1990], use event-based models in which multiple software modules react to events announced by other modules. Similar concepts have a long history in the artificial intelligence community, such as actors [Hewitt and Inman 1991] and blackboard systems [Jagannathan et al. 1989]. Some rule-based systems, such as those found in certain process-centered environments (e.g., **Darwin** [Minsky and Rozenshtein 1990], **Marvel/Oz** [Ben-Shaul and Kaiser 1995], **Oikos** [Montangero and Ambriola 1994], **Adele** [Belkhatir et al. 1994]), are based in part on event-based substrates, in that updates to data may trigger particular actions; such systems generally have a broader focus than software integration, encompassing configuration management, software process, and other domains. Several operating systems even have event announcement primitives built in to facilitate application integration (e.g., [Commodore-Amiga Incorporated 1992]).

There have been several attempts to formulate a general model of event-based integration. Wasserman [Wasserman 1990] defined control integration as the ability of software "tools" (modules) to "notify one another of events... as well as the ability to activate the tools under program control." Thomas and Nejmeh [Thomas and Nejmeh 1992] extended Wasserman's work and identified two basic control integration properties: provision of invokable operations ("services") by tools and use of those operations by other tools. Arnold and Mémmi [Arnold and Memmi 1992] defined an informal reference model to help compare and contrast control integration approaches. Garlan and Notkin [Garlan and Notkin 1991] used **Z** [Spivey 1989] to specify a formal model of implicit invocation, a subset of control integration typified by **FIELD** [Reiss 1990], and compared and contrasted several well-known systems.

The EBI framework provides a more precise model of loose control integration than Arnold and Mémmi's, capturing event-based functionality within a set of abstract components (abstract data types). Garlan and Notkin's model is more formal than the EBI framework but is restricted to implicit invocation and goes into less semantic detail than does the EBI framework.

The terms control integration, event-based integration, and implicit invocation are used frequently in the literature to mean similar things. We use the term "event-based integration" because we believe it best captures the scope of our model. Control integration is too broad a term, since it may refer to loose or tight integration, and the EBI framework supports only loose integration. Implicit invocation refers only to anonymous multicasting and is therefore too specific a term for the EBI framework, which encompasses other models of messaging.

The remainder of this section surveys existing approaches to event-based software integration. We examine these approaches across five dimensions: methods of communication, expressiveness of module interaction descriptions, intrusiveness of module interaction descriptions, static versus dynamic behavior, and module naming issues.

Methods of Communication. Two primary methods of communication among software modules are point-to-point and multicast. Point-to-point means that data is sent directly from one software module to another. Two common point-to-point approaches are procedure call[2] and application-to-application. A procedure call sends procedure parameters from a software module known as the caller to another known as the callee, optionally returning values to the caller. In the application-to-application approach, application programs have unique IDs, and other programs send messages to them using the IDs as addresses. This approach is common on personal computers (e.g., [Apple Computer, Inc. 1991; Zamara and Sullivan 1991]), since the simplifying assumptions of "one user per machine" or "one invocation of a given program at a time" can often be made. Application-to-application communication is often designed so that an end-user can specify the message-passing that occurs among programs, whereas procedure calls are generally specified only by software developers.

Multicast means that data is sent from one software module to a set of other software modules. Two popular multicasting approaches are implicit invocation and the software bus. In the implicit invocation approach, software modules express

[2]Procedure call is not itself an event-based mechanism, but it is often used to transmit messages between modules in event-based systems.

their interest in receiving certain types of data that are then routed, usually by a server, to the appropriate recipients. Implicit invocation, originally called selective broadcast, was pioneered by **FIELD** [Reiss 1990]. Today, implicit invocation is used in many commercial products such as Hewlett-Packard's **SoftBench** [Gerety 1990] and Sun's **ToolTalk** [SunSoft, Inc. 1993]. It has also been added to some programming languages (e.g., [Notkin et al. 1993]). In the software bus approach, software modules have their inputs and outputs bound to the channels of an abstract bus. Data sent to a bus channel is received by all tools with an input bound to that channel. A key feature is that bus connections can be rearranged without modifying the tools. An example of a software bus is **Polylith** [Purtilo 1994].

The EBI framework models the whole spectrum of point-to-point, multicast, and broadcast communication. The distinction between these three methods becomes moot in a dynamic model where the number of recipients may change. Point-to-point and broadcast are modeled as special cases of multicast to a single recipient and to all recipients, respectively.[3]

Expressiveness of Module Interaction Descriptions. Different approaches to integration provide varying amounts of expressiveness in specifying module interactions. The least expressive approaches have no explicit specification other than what can be inferred from the modules' source code. An example is **ARexx** [Zamara and Sullivan 1991]. More commonly, the input and output operations of each module are specified in terms of procedure signatures, as in **ToolTalk** [SunSoft, Inc. 1993]. These operations are typically known as a module's interface. **SoftBench** [Gerety 1990] not only specifies module interfaces, but also provides a programming language [Fromme 1990] for specifying the actions of modules on receipt of particular types of messages.

Some systems support specification of not only module interfaces but also the connections between modules. **Polylith** [Purtilo 1994] provides a module interconnection language (MIL) to specify direct connections between the inputs and outputs of module interfaces. For example, MIL can specify that a particular output of one module's interface is routed directly to a particular input of another module's interface. **Polylith** provides two types of connections: unidirectional and bidirectional data transmission. In contrast, software architecture models such as **Meld** [Kaiser and Garlan 1987], **ACME** [Allen and Garlan 1994], and **Rapide** [Luckham et al. 1995] add support for the specification of many types of connections, usually called connectors. Connectors are often first-class entities and may be user-specified. Software architecture models address a higher level of abstraction and a broader range of architectures than the EBI framework does. Thus, the two approaches are not directly comparable. In particular, software architecture presupposes important details that the EBI framework explores. For example, **ACME** has been used to model a simple "event-based" architecture [Garlan et al. 1994], but at a very high level that does not address many features found in existing event-based systems (e.g., rich support for dynamism, delivery constraints, and message transforming functions, as described in Section 5).

The EBI framework is independent of any particular module description language

[3]An implementation, however, may optimize point-to-point, multicast, and broadcast differently.

and provides an abstract container, called a configuration specification, where such descriptions are placed.

Intrusiveness of Module Interaction Descriptions. Different approaches to integration specify module interactions with varying degrees of intrusiveness. The most intrusive specifications are inserted into the modules themselves. Typically, the source code of each module is modified to allow the module to communicate with other modules, and there is no external specification of module interactions.

A less intrusive approach is encapsulation, or wrapping. Instead of modifying the source code of a module, an external layer of software, called a wrapper, is created to specify a module's behavior. **SoftBench** [Gerety 1990] uses wrappers. Some systems, such as **ToolTalk** [SunSoft, Inc. 1993], combine source code modification with wrapping, instrumenting a module's source code to enable it to communicate with the wrapper.

The EBI framework makes no assumptions about the intrusiveness of interaction descriptions and therefore supports both module modification and wrapping.

Static vs. Dynamic Behavior. Different integration approaches support varying degrees of static and dynamic specification of their behavior. There are advantages and disadvantages to each kind of specification. Static specification is more easily checked for correctness than dynamic specification, but it is severely limited in flexibility. In contrast, dynamic specification allows module behavior and/or interactions to be changed while modules are executing; however, reasoning about the behavior of large, dynamic systems can be very difficult.

The original **Polylith** [Purtilo 1994] supported only static specification of interactions, as modules had to be terminated and restarted to change their interactions.[4] The **SDL BMS** [Barrett 1993] supports only dynamic specification of interactions, allowing modules to register and unregister for service dynamically. **ToolTalk** [SunSoft, Inc. 1992] supports both static and dynamic specification of interactions. The EBI framework models both static and dynamic specification of behavior.

Naming Issues. In order for a module to receive messages, it must somehow be identifiable. There is a spectrum of approaches for identifying modules, ranging from abstract to primitive. At the "abstract" end of the spectrum, senders are completely unaware of the names and locations of recipients. Naming issues are typically handled by a message server that locates each module that should receive a message. For example, **FIELD** [Reiss 1990] modules do not have explicit names; instead, each software module registers message patterns with the message server, denoting the types of messages that the module wants to receive. Senders broadcast their messages blindly, without knowledge of any recipients, and the message server delivers to each module only those messages matching the module's message patterns.

At the next lower level of abstraction, senders still need not know the names and locations of recipients, but they must know references to those names, called aliases. (Aliases may themselves be names.) For example, **ISIS** [Birman 1993] allows multiple modules to be grouped under an alias, so any message sent to that

[4]The latest specification of **Polylith** models dynamism (e.g., [Purtilo and Hofmeister 1991]).

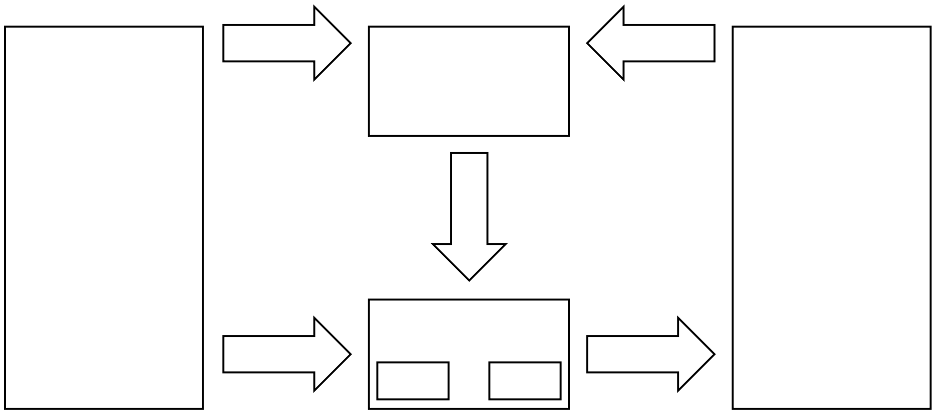

Fig. 1. Relationships among informer, listener, registrar, router, message transforming functions (MTFs) and delivery constraints (DCs).

alias is transparently forwarded to all members of the group.

At the next lower level of abstraction, senders must know the exact names of recipients. **CORBA** [Digital Equipment Corporation et al. 1993] requires each software module to have a globally unique name, and each message must contain its intended recipient's name. **CORBA** uses a name server to locate receivers that are distributed on multiple machines. **ARexx** [Zamara and Sullivan 1991] modules have globally unique names, but each message does not contain the recipient's name; instead, the name is used once to open a communication channel between sender and receiver.

At the "primitive" end of the spectrum, senders must know the exact names and locations of recipients. Such approaches are rare nowadays, since this degree of knowledge among modules increases coupling and therefore conflicts with the benefits of loose integration.

The EBI framework can model all of these methods of naming.

3. OVERVIEW OF THE FRAMEWORK

This section presents a brief, high-level overview of the EBI framework, in preparation for a detailed description in Section 5. We begin with a motivating discussion of participants and framework components. After that, we discuss how the EBI framework is used for modeling integration approaches.

Participants and Framework Components. In the EBI framework, depicted in Figure 1, one or more modules, called *participants*, transmit and/or receive pieces of information, called *messages*, in response to occurrences, called *events*. Participants that transmit messages are called *informers*, and those that receive messages are called *listeners*. A participant may be both an informer and a listener.

Participant interaction is supported through four types of *framework components*: registrars, routers, message transforming functions, and delivery constraints. Before a participant can transmit or receive any messages, it must register its intent to do so via a *registrar*. (Unregistration and re-registration are also modeled.) Using information from the registrar, a *router* delivers messages from informers to their intended listeners.

The EBI framework allows the content and/or routing of messages to be modified after the messages are sent but before they are received. *Message transforming functions* (MTFs) defined within a router can dynamically alter messages sent to listeners. One simple type of MTF commonly found in integration systems is a filter, which selectively accepts or rejects messages from a listener's incoming message stream if they satisfy certain criteria. Another is an aggregator, which transmits a new message whenever a particular sequence of incoming messages is detected.

Rules of message delivery, called *delivery constraints* (DCs), may also be defined within routers. For example, a constraint can specify that all messages must be received in exactly the order they are sent, or that messages must arrive within a certain period of time.

The EBI framework supports the description of *groups*. A group is a set of participants and/or framework components that may be logically treated as a unit. Groups permit the EBI framework to model composition and decomposition. For example, a group containing a registrar, router, and set of participants could itself register as a participant. Groups can also be used to support aliases, allowing multiple modules to be accessed together using a single name. Modules can be added to, removed from, or replaced in a group without changing the group name, further facilitating loose integration.

In summary: registrars, routers, message transforming functions, and delivery constraints are the components of the EBI framework in which participants communicate. Each represents a basic aspect of integration. Participants are the interacting software modules. Registration distinguishes modules that can interact from those that cannot. Routing transmits data among participants. Message transforming functions modify data en route to their destinations. Finally, delivery constraints control the delivery of data.

Using the Framework. As a reference model, the EBI framework itself does not specify implementation details such as particular registrars and routers, a method for turning software modules into participants, or particular patterns of interaction between participants. Instead, using a three-phase description process, these details are imposed on the framework to produce a descriptive model of a particular software integration system. The descriptive phases are called instantiation, adaptation, and configuration.

The *instantiation* phase is the process of describing the semantics of framework components, including framework component types, particular instances of components, groups of components, and policies for system behavior (e.g., how to handle delivery constraint violations). Instantiation is important because it allows us to discover and model high-level similarities and differences between event-based integration approaches, without getting bogged down with implementation details. We call an instantiation of the framework an *integration mechanism*. The description of a mechanism is called an *integration mechanism specification*. When it is unambiguous to do so, we use the simpler term *mechanism* in place of "integration mechanism" in the previous definitions.

The *adaptation* phase is the process of describing methods for turning software modules into participants that can interact within a mechanism. Such methods include modifying module source code, linking with custom libraries, or wrapping modules in another layer of software (discussed in Section 2). Many factors influence the choice of adaptation method, such as properties of the message types, flexibility of message delivery, and degree of concurrency desired [Notkin et al. 1993]. The design decisions involved in choosing an adaptation method are well studied [Notkin et al. 1993; Sullivan and Notkin 1992] and thus not explored further in this paper.

The *configuration* phase is the process of describing participants and their permissible interactions. For example, one can define participant types, participant instances, or that a given participant instance can use a given MTF. Such a description is called a *configuration specification*.

It is important to note the relationship between the configuration phase and registration. A configuration specification describes the allowable interactions between participants. Registration defines the current subset of allowable interactions that individual participant instances support. A mechanism must provide a policy for handling the case where a participant attempts to register information that violates the configuration specification.

4. A DESCRIPTIVE TYPE MODEL

To facilitate the discussion of the EBI framework in Section 5, we use a simple, descriptive type model. This type model allows us to reason about participants and framework components as abstract data types (ADTs). It also provides a convenient way to represent certain relationships within the EBI framework, as modeled by subtyping and inheritance.

Our type model is similar to those found in common, object-oriented programming languages like C++[Lippman 1990], Ada95 [Barnes 1995], and Smalltalk [Goldberg 1984]. A *type* is simply a named set of attributes and operations. An *attribute* is a named, typed value. An *operation* is a named, invokable entity. Notable features include:

—Support for multiple inheritance. A type inherits all attributes and operations from all of its supertypes and is behaviorally their subtype.

—Support for first-class types; that is, instances of type `Type`. Thus, to avoid a cycle in the type hierarchy (i.e., having types `Type` and `Attribute` inherit from each other), the type `MetaType` is introduced as the root, as in Smalltalk [Goldberg 1984].

—Support for queries over types. Queries can be used for many purposes, including determining the extent of a type (i.e., the set of all instances of the type), determining which types have an attribute with a particular name or type, or determining which instances of a type have a particular attribute value.

Figure 2 illustrates how the EBI framework maps to our descriptive type model. Using this type model, a router r, for example, is modeled as an instance of type `Router` or one of its subtypes. A complete specification of the type model is found in Appendix A.

This descriptive type model is primarily a notation for discourse, not an implementation guide. Use of this type model in an implementation would require

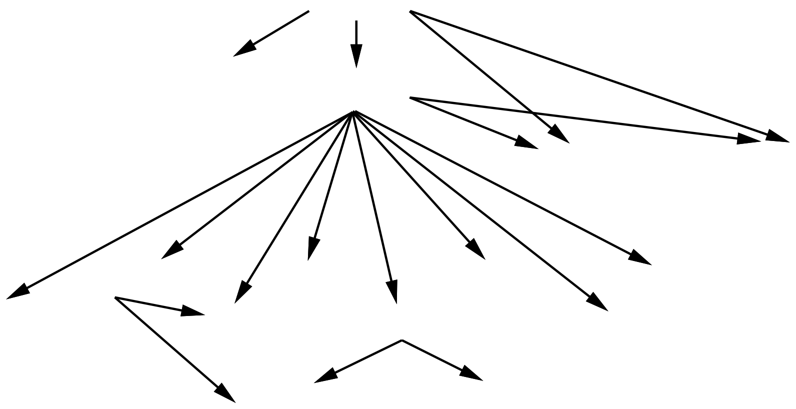
Fig. 2. ADT's of the EBI framework. Arrows point from supertypes to their subtypes.

specification of additional properties, such as an inheritance conflict resolution policy, a definition of type equivalence, constraints on subtype/supertype relationships, and a query language.

5. The EBI Framework

This section provides an in-depth description of the EBI framework. For each definition, we include abbreviated information about the associated type in the type hierarchy: its name, supertypes, and operation names (not including inherited operations).

5.1 Participants: Informers and Listeners

Type:	`Participant` (subtypes `Informer`, `Listener`)
Supertypes:	`Object`
Operations:	`Set_Router`, `Get_Router`, `Set_Registrar`, `Get_Registrar`

A *participant* is a software module that has been adapted to be compatible with a particular integration mechanism. An *informer* (subtype `Informer`) is a participant that detects events and sends messages, and a *listener* (subtype `Listener`) is a participant that receives messages. A participant may be both an informer and a listener (by defining a subtype that inherits from both `Informer` and `Listener`). Typical candidates to be modeled as participants are the clients of **FIELD** [Reiss 1990] and the clients and objects of **CORBA** [Digital Equipment Corporation et al. 1993].

5.2 Events and Messages

Type:	`Message`
Supertypes:	`Object`, `MetaType`
Operations:	`Set_Parameter_Value`, `Get_Parameter_Value`, `Message_From_Name`

An *event* is an occurrence such as the invocation of a program, the modification of a file, the change of a participant's state, or the sending of a message. Informers detect events by some means that is outside of the EBI framework. Messages are the manifestations of events in the framework.

A *message* (type `Message`) is information emitted by an informer in response to an event or events. Examples are the broadcast messages of **FIELD** [Reiss 1990] and the requests of **CORBA** [Digital Equipment Corporation et al. 1993]. The common notion of message parameters (analogous to the formal parameters of a subprogram) is modeled using attributes. For example, a message with parameters x and y is modeled as an instance of a subtype of `Message` with corresponding attributes x and y. The operations `Set_Parameter_Value` and `Get_Parameter_Value` provide a convenient shorthand for setting and getting parameter values without using attribute-related operations directly. Note that all parameters are attributes, but the converse is not necessarily true.

Type `Message` may have attributes to allow fine-grained control over the sending and receiving of individual messages. Three useful attributes are:

—`Delivery_Constraint`, specifying delivery constraints on this message. By default, the value is "no constraints specified."

—`Synchronization`, representing whether the message can be sent synchronously, asynchronously, or either. By default, the value is "either."

—`Access_Control`, specifying the listeners that are permitted to receive this message after it is sent. The value implicitly determines whether this message is sent point-to-point or multicast. By default, the value is "all listeners" (i.e., broadcast).

The values of these three attributes specify properties of message instances, rather than message types, so an informer can choose different delivery constraints, synchronization, and/or access control to apply to each message that it sends. We chose "per instance" specification over "per type" specification of these attributes for increased flexibility when messages are sent, since the former specification can model the latter, but not vice-versa. If desired, type-level specifications of delivery constraints, synchronization, and access control can be specified upon registration or in a configuration specification. Consistency checking may be necessary to insure that instance-level attribute values do not conflict with type-level and system-level specifications.

Another useful attribute for type `Message` is `Event`, which specifies the event that caused a message to be sent. This attribute can be used to bridge the gap between events and messages, giving listeners knowledge of events that have occurred.

5.3 Registrars

Type:	`Registrar`
Supertypes:	`Object`
Operations:	`Register_Informer`, `Register_Listener_Polling`, `Register_Listener_Active`, `Register_MTF`, `Register_Delivery_Constraint`, `UnRegister`

Before a participant can send or receive any messages, its intent to do so must be registered with a *registrar*. Information about a participant can be registered by that participant or by another entity, such as another participant or a software developer.[5] Both static registration (prior to participant invocation) and dynamic registration (during participant execution) can be modeled.

Informers must register their intent to send messages, and listeners must register their intent to receive messages. All participants can register message transforming functions and delivery constraints that operate on individual messages they send or receive. For example, a listener interested only in messages from a particular informer or pertaining to a particular event can register an MTF that filters out all incoming messages not matching those criteria. In addition, informers can register specialized delivery constraint, synchronization, and access control information for each type of message they intend to send, providing the "per type" control mentioned in Section 5.2.

When a participant registers, it is returned a handle to an instance of type `Router`. The participant calls that router's `Send` and/or `Receive`[6] operations to communicate with other participants. The router can be a physical router or a logical router (e.g., a group of routers), and a registrar can cause a participant's router to be replaced by another router (e.g., for optimization purposes) via the participant's `Set_Router` operation.

When a participant registers, consistency checking may be necessary to insure that the participant's request does not conflict with any other registrations. For example, if two different listeners both register a delivery constraint to "receive messages of type t before any other listener receives them," the registrations cannot both be satisfied simultaneously.

5.4 Routers

Type:	`Router`
Supertypes:	`Object`
Operations:	`Send, Receive, Message_Waiting`

A *router's* purpose is to receive messages from informers and deliver messages to listeners. As software modules register as informers and/or listeners, the registrar instructs the router to create communication channels between those informers and listeners. A communication channel may be as simple as a direct connection between an informer and a listener, or it may involve the dynamic evaluation of message transforming functions, delivery constraints, or other routing algorithms to determine the recipient of a message.

As a router receives messages to deliver, it maintains a partially ordered set (poset) of undelivered messages and their intended listeners and delivers them in an order consistent with the partial order. The complete path of a message in an

[5]For simplicity, in the remainder of the paper, we say "The participant registers..." to mean that either the participant performs the registration itself, or another entity performs the registration for that participant.

[6]Except in the case of active message delivery, described in Section 5.4, in which no `Receive` operation is needed.

integration mechanism is as follows. Let L be the set of all listeners, M be the set of all messages, and $POSET(X)$ be the set of posets of elements of set X.

(1) An informer sends a message $m \in M$.

(2) The message m is received by a router.

(3) The router applies all applicable message transforming functions and delivery constraints to message m, resulting in a partially ordered set $\Sigma = \{(m_1, l_1), (m_2, l_2), \ldots, (m_k, l_k)\} \subseteq POSET(M \times L)$, with $m_1, m_2, \ldots, m_k \in M$ and $l_1, l_2, \ldots, l_k \in L$. Note that m is not necessarily in $\{m_1, m_2, \ldots, m_k\}$.

    Σ is ordered because the evaluation of MTFs and delivery constraints may impose an order on the delivery of messages. Σ is a partial order because there may be many acceptable orderings that satisfy the MTFs and delivery constraints. By leaving the order as partial, rather than having the router select one of the possible total orderings, we support parallelism in the router, since incomparable elements of Σ can be delivered in parallel.

(4) The router combines partial order Σ with its existing partial order of undelivered messages. Each message m_i is delivered to listener l_i in an order compatible with the resulting poset.

Routers support two models of message delivery: polling and active. In the polling model, a listener invokes a router's `Receive` operation to receive a message. In the active model, a listener registers a custom "receive" operation, and whenever a message is ready to be delivered to the listener, the listener's router automatically invokes that operation. Each listener chooses one or both of these delivery models at registration time. We chose to support both models of message delivery at the framework level, rather than leave it as an implementation issue, because of the impact at the type model level: only the polling model needs a router `Receive` operation, and the signatures of the two corresponding `Registrar` operations are different.

In many integration systems, the functionality of the registrar and router is incorporated into a single component: the ORB of **CORBA** [Digital Equipment Corporation et al. 1993] and the broadcast message server of **FIELD** [Reiss 1990] are two examples. In the EBI framework, however, registrars and routers have sufficiently different semantics to justify their separation. Registration is the act of *obtaining permission* to communicate. A registrar can check whether the requested kind of communication violates some constraint. Routing is the act of *carrying out* communication. A router can check whether a particular message transmission violates some constraint. Routers handle messages, but registrars do not. In addition, routers are time-critical components: message transmission should proceed as quickly as possible, often subject to real-time constraints. The granting of permission to communicate, however, is arguably not as time-critical. In many integration systems, registration is performed only once per participant, whereas routing is performed many times per participant. Finally, a mechanism may allow its number of registrars to be different from its number of routers. For example, a mechanism with a single registrar may have multiple routers to optimize message transmission.

5.5 Message Transforming Functions

Type:	`MTF`
Supertypes:	`Object`
Operations:	`Transform`

A *message transforming function* (MTF) transforms a message in transit into a (possibly empty) poset of messages that may be delivered to other listeners. An MTF also has a state (the values of its attributes) that can affect its output. Formally, let M be the set of all messages, L be the set of all listeners, and S be the set of all MTF states, and recall that $POSET(X)$ is the set of all partially-ordered sets of X. An MTF is a function

$$f : M \times L \times S \rightarrow POSET(M \times L) \times S$$

For the purposes of converting messages into other messages and/or re-routing them to other listeners, messages and listeners are in the domain and range of an MTF so they can be changed by the MTF. The range contains a poset to allow the output of the MTF to be combined with a router's poset of undelivered messages, as described in Section 5.4. Having a state gives an MTF the ability to base its output on previously received messages (e.g., on sequences of messages), not just the single message given as input. For example, if we want an MTF that outputs a message m_0 whenever it receives the message sequence (m_1, m_2, m_3), then the MTF could change state on receipt of m_1, change state again on receipt of m_2, and then output m_0 on receipt of m_3. Of course, a simple MTF function could ignore its state information. MTFs are equivalent in power to on-line Turing Machines [Hennie 1966].

Most integration systems support some message transforming functions: most commonly, filters. **FIELD** [Reiss 1990] listeners register message patterns (filters) to specify which kinds of messages they receive. **ToolTalk** [SunSoft, Inc. 1993] allows listeners to register filters that accept or reject messages based on their "class." **Bart** [Beach 1992] uses "declarative glue" and relational algebra to design filters that select relevant data. Policies in **Forest** [Garlan and Ilias 1990] provide aggregation. A more subtle example of aggregation is that of **Odin** [Clemm and Osterweil 1990], in which objects have multiple dependents, all of which must (effectively) send an "up-to-date" message before the target object can send its own.

5.6 Delivery Constraints

Type:	`Delivery_Constraint`
Supertypes:	`Object`

A *delivery constraint* is a property of message delivery that is enforced by a mechanism. Examples are order of delivery, timing of delivery, and atomicity properties (e.g., a message is delivered either to all or none of its intended listeners). Delivery constraints can be represented by any of a number of formalisms, including temporal logic [Koymans 1992] and partially ordered sets [Luckham et al. 1995; Mishra et al. 1991].

In the EBI framework, a distinction is made between two classes of delivery constraints. A *system delivery constraint* is one that applies to all messages sent in the mechanism. These constraints often arise due to underlying physical constraints, such as "messages can be transmitted no faster than 10 megabits per second." A *user delivery constraint* may apply only to some messages sent in the mechanism; for example, "Messages of type t have higher priority than all other messages and should be delivered first."

It is possible for delivery constraints to be inconsistent with one another. A mechanism can have policies for resolving such inconsistencies. In addition, violations of delivery constraints may or may not be permitted by a mechanism. When permitted, a policy for responding to such violations can be specified.

Delivery constraints are found in some integration systems. **ISIS** [Birman 1993] has "message delivery orderings" that enforce causal relationships between messages. **Consul** [Mishra et al. 1993] has an "order protocol" that enforces orderings on received messages. The Amiga **Exec** [Commodore-Amiga Incorporated 1992] allows listeners to be prioritized for receiving messages. The router in Garlan and Scott's extended Ada [Garlan and Scott 1993] delivers messages to listeners in an order defined statically in the router's source code. The hard and soft deadlines of real-time systems also are examples of delivery constraints.

5.7 Groups

Type:	`Group`
Supertypes:	`Object`
Operations:	`Add_Member, Remove_Member, Is_Member, Members_Of`

Objects in the EBI framework, such as participants and framework components, can be collected into sets called *groups* and treated as a unit. The objects in a group are called its *members*. Groups may themselves be members of other groups, but a group cannot contain itself directly or indirectly. Groups may be empty. A simple example of a group is a mail alias, in which a single mail address is used to represent a set of mail addresses, each of which may itself be an alias.

Groups can be used to support composition. For example, a set of registrars, routers, and participants can form a group that itself registers to be a participant, as shown in Figure 3. Groups have other uses as well; for example, one could define a group of messages to mean that the messages should be considered logically equivalent.

It is possible to model groups indirectly using attributes; for example, a set of objects whose types share an attribute could be considered a group. Attributes, however, do not have special semantics associated with them, and issues such as naming and multiple group membership can make an attribute-based grouping model awkward to use. Therefore, the EBI framework models groups explicitly.

Groups appear in various integration systems, though generally only participant groups are supported. **ISIS** [Birman 1993] provides support for several kinds of participant groups, in which the members have varying degrees of awareness of each other. **FIELD** [Reiss 1990] supports participant grouping both explicitly (participants can limit the scope of their broadcasts to a select group) and implicitly (all participants that register a given message pattern form a group). The **Pilgrim**

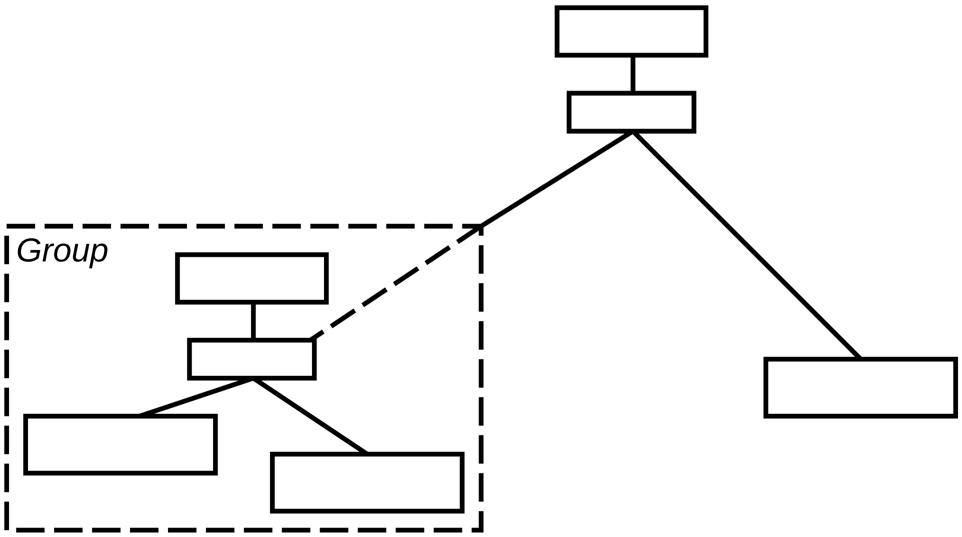

Fig. 3. Composition using groups. Registrar 1, Router 1, and Participants 1 and 2 form a group. The group itself is registered as a participant with Registrar 2. Communication to and from the group is accomplished by sending messages between the two routers.

Event Notifier [DiBella 1994] provides "subscription lists" as a grouping mechanism for listeners, and a message sent to a list is routed to all of the list's members. **Schooner** [Homer and Schlichting 1994] and **Cronus** [Schantz et al. 1986] have multiple servers, each serving a group of participants. **SoftBench** [Gerety 1990] tools can encapsulate multiple participants, implicitly forming a group.

5.8 Specification of Mechanisms

Having defined the framework components, we now turn our attention to framework instantiation. A mechanism specification, which defines all the instantiated properties of a mechanism, is created by defining the types, instances, policies, and languages that comprise an integration mechanism and distinguish it from other integration mechanisms. Each of these definitions is elaborated below, using general descriptions and motivating examples.

—Define subtypes and restrictions (i.e., removal of unneeded attributes or operations) of framework components, messages, and groups:
  —*Subtypes/restrictions of* `Message`. For example, **FIELD** [Reiss 1990] messages are restricted to strings, but **ACA** [Patrick, Sr. 1993] messages are abstract data objects. **ToolTalk** [SunSoft, Inc. 1993] messages are restricted to be sent asynchronously only,[7] whereas **Polylith** [Purtilo 1994] supports both synchronous and asynchronous communication.
  —*Subtypes/restrictions of* `Registrar`. For example, systems that combine the registrar and router into a single component (e.g., [Purtilo 1994; Reiss 1990])

[7]Synchronous messaging can be built from asynchronous messaging, but this requires additional implementation work by any developer using **ToolTalk**.

can create a subtype that inherits from both `Registrar` and `Router`.

- —*Subtypes/restrictions of* `Router`. For example, **Polylith** uses the polling model of message delivery, and **SoftBench** [Gerety 1990] uses the active model. The **ARexx** [Zamara and Sullivan 1991] router is restricted to point-to-point communication, whereas **FIELD** permits multicasting. A message history mechanism can be modeled by creating a subtype of `Router` with a `History` attribute, containing all messages the router has handled.
- —*Subtypes/restrictions of* `MTF` *and* `Delivery_Constraint`. For example, **Consul** [Mishra et al. 1993] has four subtypes of message ordering constraints available.
- —*Subtypes/restrictions of* `Group`. For example, **ISIS** [Birman 1993] groups can contain only participants, and their members may or may not be aware of one another.

—Define instances of framework components and groups:

- —*Particular registrars and routers*. For example, **SoftBench** has only one router, but **Schooner** [Homer and Schlichting 1994] supports multiple routers.
- —*Particular message transforming functions*. For example, particular **Forest** [Garlan and Ilias 1990] policies are specified in "Policy Definition Files."
- —*Particular delivery constraints*. For example, **Consul** supports the definition of acceptable partial orderings of messages.
- —*Particular groups* (not involving participants). For example, in **ToolTalk**, the message server could conceptually be viewed as a grouped registrar and router.

—Define policies for handling violations or resolving mismatches:

- —*Synchronization mismatches*. For example, if an informer and listener with incompatible `Send` and `Receive` synchronization attempt to communicate, some intermediate buffering may be necessary.
- —*Access control violations*. For example, **Zephyr** [DellaFera et al. 1988] uses the **Kerberos** [Steiner et al. 1988] authentication system to enforce access control.
- —*Delivery constraint violations*. For example, real-time systems specify that the violation of a time constraint indicates either complete or partial failure of the system.
- —*Delivery constraint inconsistencies*. For example, if two participants register conflicting delivery constraints, at least one registration must be rejected.
- —*Mechanism specification violations*. For example, if a component acts in a way contrary to its specification, an exception could be raised.

—Provide languages for describing MTFs, delivery constraints, and queries. For example, the MTF language of **FIELD** can describe only filters, **Forest** can describe only aggregators (since a policy action can be either a single message or the empty message), and **CORBA** [Digital Equipment Corporation et al. 1993] specifies no MTF language at all.

An initial mechanism specification statically describes the mechanism upon instantiation. If the components of an integration approach can change over time, this is modeled simply by changing the mechanism specification. For example, one might add new message transforming functions or change the number of routers. Of course, arbitrary changes might introduce inconsistencies into the mechanism, so some restrictions or consistency-checking may be necessary.

Mechanism specifications do not define participant-related information, such as participant types and instances. These are given in configuration specifications, the subject of the next section.

5.9 Specification of Configurations

A configuration specification, which describes participants and their allowable interactions, is created by defining types, instances, groups, and policies that pertain to participants. Each of these definitions is elaborated below. A participant instance can register to support all, or only a subset, of the defined interactions, but it cannot legally violate the configuration specification. A configuration specification defines:

—*Participant types*. This includes each type's attributes, operations, and allowable messages. For example, **CORBA** [Digital Equipment Corporation et al. 1993] has an Interface Definition Language for creating participant types.

—*Instances of participant types and subtypes*. For example, **Polylith** [Purtilo 1994] participants (called "tools") can be defined as instances of types (called "modules").

—*MTFs and delivery constraints* (defined in a mechanism specification) that may be used by participant types or instances. For example, **Forest** [Garlan and Ilias 1990] policies are associated with listeners.

—*Group instances containing participants*. For example, **DEEDS** [Liang et al. 1994] allows the specification of particular groups of participants.

—A policy for handling *configuration specification violations*. For example, listeners in **CORBA** have exceptions defined as part of their interfaces.

An initial configuration specification statically describes the configuration of the initial mechanism specification. If an integration approach supports dynamic reconfiguration of its participants, this is modeled simply by providing a new configuration specification. For example, a new participant type or instance could be added. Again, changes to the configuration specification may have to be checked for consistency.

6. CASE STUDIES

Many existing integration systems can be viewed as instantiations of the EBI framework. We present reasonable mechanism specifications for three well-known integration approaches: the implicit invocation system **FIELD** [Reiss 1990], the software bus system **Polylith** [Purtilo 1994], and the object-oriented **CORBA** 2.0 specification [Digital Equipment Corporation et al. 1993; BNR Europe Limited et al. 1995]. As the framework is intended for high-level comparison of systems, these case studies address conceptual, not operational (implementation), aspects of these systems. Once the mechanisms are mapped to the framework, the task of comparing and contrasting the systems becomes much easier, as we demonstrate.

Comparison of integration mechanisms is an important, practical task for several reasons. It is necessary at some level when deciding which of several mechanisms is most suitable for one's purposes. The EBI framework provides a useful level of abstraction for this comparison. In addition, such comparisons can facilitate

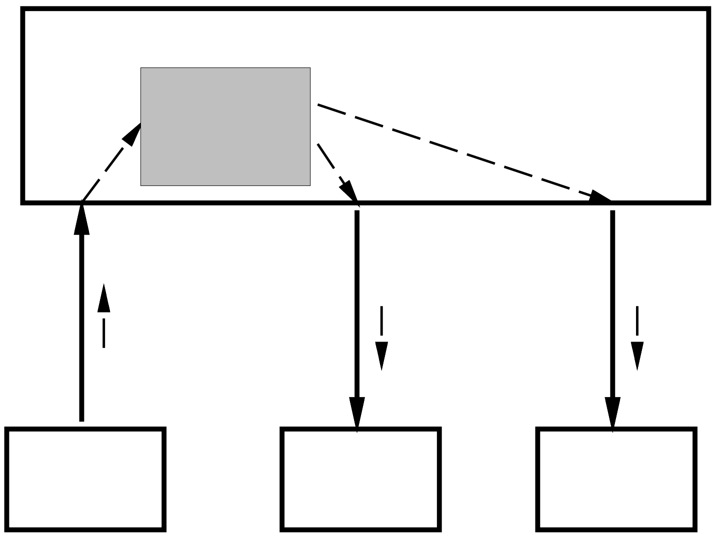

Fig. 4. Components of **FIELD**, plus participants. Tool 1 sends message m to **Msg**. **Msg** routes the message to all tools that have registered a message pattern matching m. The Policy tool is optional and not pictured.

interoperability between integration mechanisms. By identifying the components and functionality that are common among the mechanisms, attention is focused on the core features of the mechanisms, allowing a software developer to exploit their similarities and address their differences. One similarity that we will see in the case studies is that all of the mechanisms have a readily identifiable router component. Thus, one approach to interoperability would be to group the different router types into a single router that can handle messages from all three mechanisms. An important difference between the mechanisms, however, is that one provides point-to-point communication, one provides multicast, and one provides both. The EBI framework calls attention to this important distinction that must be addressed by any plan for interoperability among the mechanisms.

This section utilizes the framework types and operations defined in Appendix A. The case study for **FIELD** uses these type definitions in detail to illustrate how the EBI framework is used, and the remaining case studies provide more abbreviated information.

6.1 **FIELD**

FIELD [Reiss 1990] has a client-server architecture, as shown in Figure 4. Client programs, called tools, broadcast messages anonymously by sending them to a central message server, called **Msg**. Tools specify the classes of messages that they want to receive by registering message patterns with **Msg**. **Msg** delivers to each tool only the messages that match the tool's message patterns.

FIELD provides an optional tool, called the Policy tool, that can intercept messages and perform user-specified actions on receipt of those messages (reroute messages, replace them with other messages, etc.). These user-specified actions, called policies, are external to tools, so no tool modification is necessary to enforce policies. An example policy is, "exiting the text editor should cause an automatic recompilation of the program that was edited." We defer discussion of the Policy Tool until Section 6.1.3.

6.1.1 *An Informal Mapping for FIELD.* We begin by informally identifying aspects of **FIELD** that are likely candidates to be mapped to the EBI framework. Experience has shown us that a usable mapping is often produced by considering participants, messages, registrars, routers, groups, MTFs, and delivery constraints, in order.

Participants and Messages. **FIELD** tools are programs that communicate with one another, so we define them to be participants. Since tools may send and/or receive messages, they may be informers and/or listeners. Tools communicate via **FIELD** messages, which we map to framework messages.

Registrars and Routers. **FIELD** tools register to receive messages (that is, to be listeners) by providing message patterns to **Msg**, so we define **Msg** to be a registrar. Tools need not register explicitly to send messages (to be informers); this registration is done automatically as part of adaptation, as will be described in Section 6.1.4. Messages are routed to their destinations by **Msg**, so we define **Msg** also to be a router.

Groups. **FIELD** has a facility that allows a set of tools to multicast messages only among themselves [Reiss 1996]. This is clearly a form of participant grouping.

MTFs and Delivery Constraints. The purpose of a **FIELD** message pattern is much like that of a function that accepts or rejects a message m, depending on whether or not it matches the pattern. If listener l registers a message pattern p, and message m is broadcast by some tool, then listener l receives message m only if m matches pattern p. This behavior can be modeled by an MTF that takes m as input and outputs the pair (m, l) if and only if m matches pattern p. For delivery constraints, **Msg** must deliver messages in the order it receives them.

Once we have informally mapped the parts of **FIELD** onto the EBI framework, we perform the phases of instantiation, adaptation, and configuration, as given in Section 3.

6.1.2 *Instantiation.* We perform the instantiation phase according to the mechanism specification definition in Section 5.8 and the type definitions in Appendix A. A detailed instantiation of **FIELD** appears in Appendix B.

Subtypes of Message. All **FIELD** messages are strings. We model this by having **FIELD**'s message type inherit from the EBI framework types `Message` and `String`.

Subtypes of Registrar and Router. Since **Msg** needs the capabilities of a registrar and a router, we define a type `MSG` that is a subtype of both Registrar and Router. **FIELD** specifies a set of **Msg** operations [Reiss 1996] that we define as type `MSG` operations and map to operations of the `Registrar` and `Router` types:

—`MSGconnect`
This function establishes a connection between a tool and **Msg** so it can send messages. We map it to `Registrar.Register_Informer`.

—`MSGregister`
This function is used to register message patterns. We map it to the operation `Registrar.Register_MTF` because message patterns represent MTFs.

—`MSGsend`
This function sends a message asynchronously. We map it to a `Router.Send` operation that sets the message's `synchronization` attribute to "asynchronous."

—`MSGcall`
This function sends a message synchronously. We map it to another `Router.Send` operation that sets the message's `synchronization` attribute to "synchronous."

—`MSGcallback`
This function sends a message synchronously and temporarily registers the sender to receive a particular pattern in response. We map it the sequence of operations

```
Registrar.Register_Listener_Active
Registrar.Register_MTF
Registrar.UnRegister_Listener_Active
```

—`MSGreply`
This function sends a reply to a synchronous message. We map it to the operation `Router.Send`.

Subtypes of MTF. We define a subtype of MTF, `Message_Pattern`, corresponding to **FIELD** message patterns. Its `Transform` operation behaves as follows:

—`Message_Pattern.Transform`(m, l) return (m_{new}, l_{new})
If m matches the message pattern, then m is copied to m_{new}, and l_{new} is the listener who registered the message pattern. If m does not match the message pattern, the empty poset is output.[8] Although the `Transform` operation provides the parameter l, it is not needed for this computation.

Subtypes of Delivery_Constraint. Discussed as part of the Policy Tool in Section 6.1.3.

Restrictions of Group. The only groups that **FIELD** supports are groups of participants, called "Message Groups." These are formed using an alternative use of `MSGconnect`, which not only can register informers (as shown in **Subtypes of Registrar and Router**, above) but also can allow informers to restrict the scope of their multicasts. We map this **FIELD** operation to the framework operation `Group.Add_Member`.

Particular registrars and routers. **FIELD** allows multiple instances of **Msg**, but no more than one per user per host.

Particular MTFs. The **FIELD** specification does not predefine any particular instances of message patterns.

[8]To simplify the notation, we denote the return type by a single message/listener pair, when in fact it is a poset of size one (or zero).

Particular delivery constraints. **FIELD** enforces the following delivery constraint for **Msg**: "the router delivers messages in the order that it receives them."

Particular groups. **FIELD** does not predefine any groups.

Synchronization mismatches. Synchronization is determined by the sender of a message, and listeners have no control over it. Thus, there can be no mismatches.

Access control violations, delivery constraint violations, delivery constraint inconsistencies, mechanism specification violations. **FIELD** does not specify any means of handling these exceptional conditions.

Languages. The MTF language is `printf`-style strings.

6.1.3 *The FIELD Policy Tool.* The **FIELD** environment provides a number of software engineering tools — editor, debugger, cross-referencer, etc. — in addition to its event-based integration system. The Policy Tool is unique among **FIELD**'s supplied tools because it can have a significant, system-wide effect on the delivery of messages. Modeled after **Forest** [Garlan and Ilias 1990], the Policy Tool behaves according to user-defined rules, called policies, that can intercept and replace messages in transit. In this section, we model the key features of the Policy Tool: policies, the rerouting of messages to the Policy Tool, policy registration, policy evaluation order, and the policy language. These definitions supplement the **FIELD** mechanism specification given in Section 6.1.2.

A **FIELD** policy can be modeled as an MTF because it takes as input a message m, originally sent to the Policy Tool, and outputs messages to other listeners. We define a subtype of type `MTF`, called `Policy`, with the following `Transform` operation. Let X^* represent zero or more occurrences of X (Kleene closure).

—`Policy.Transform`(m, l) return $(M \times L)^*$
If m matches the specification of the policy, then the output is the sequence[9] of message/listener pairs to be delivered in accordance with the policy. l is always the Policy Tool, and is therefore ignored.

If the Policy Tool is used, then an MTF of this type is automatically registered for every listener. Another MTF is used to model the rerouting of messages to the Policy Tool for processing, instead of simply being sent to registered listeners. We define another subtype of type `MTF`, called `To_Policy`, with the following `Transform` operation:

—`To_Policy.Transform`(m, l) return (m_{new}, l_{new})
m is copied to m_{new}, and l_{new} is the Policy tool.[10] Although the `Transform` operation provides the parameter l, it is not needed for this computation.

Since the Policy Tool allows other tools to register MTFs (policies) with it, it needs the functionality of a registrar (for registration) and a router (for containing MTFs). We define a type, `Policy_Tool`, that inherits from types `Registrar` and

[9]The **FIELD** Policy Tool outputs sequences of messages; that is, a restricted form of posets. We simplify the notation correspondingly.
[10]Again, we simplify the notation, recognizing that the output is a poset of size 1.

`Router`. Since the Policy Tool is also a participant, `Policy_Tool` also inherits from type `Participant`.

The rules in a **FIELD** policy are evaluated in the order that they are specified. This provides user-definable control over the order in which messages are processed and are output by the Policy Tool. We model this as a type of delivery constraint over order of delivery.

Finally, we note that the policy language must be part of the mechanism specification. It consists of rules and triggered operations [Reiss 1996].

6.1.4 *Adaptation.* Software modules are adapted into **FIELD** participants by wrapping or modifying their source code to make calls to **Msg**'s `Router.Send` operation (**FIELD**'s `MSGsend/MSGcall` family of functions). **FIELD** specifies that all tools are automatically registered as informers on startup; thus, the wrapper or modification must ensure that `Registrar.Register_Informer` (`MSGConnect`) is invoked automatically for each participant instance at the start of its execution.

6.1.5 *Configuration.* Now we define the configuration specification according to the definition in Section 5.9.

Participant types. **FIELD** predefines types for the Policy Tool and many other tools provided in the **FIELD** environment: debugger, cross-reference server, profiler, etc. [Reiss 1996].

Participant instances. No instances of participants are defined in the **FIELD** specification.

MTFs and delivery constraints as used by participants. Message patterns (MTFs) are associated with the listeners that register them. No delivery constraints are associated with particular participants.

Group instances containing participants. No particular instances are defined by the **FIELD** specification.

Configuration specification violations. **FIELD** defines no rules for handling violations of the above specifications.

Appendix B provides a full specification of **FIELD** in terms of the EBI framework's type model.

6.2 Polylith

Polylith has a software bus architecture, as shown in Figure 5. Individual programs, called tools, connect their input and output ports to an abstract bus and send and receive messages on named bus channels. A module interconnection language (MIL) is used to encapsulate external programs as tools, and then bind the output ports of tools to the input ports of other tools.[11] Messages may be of simple, structured, or pointer types.

In terms of the EBI framework, **Polylith** tools are participants (subtypes of type `Participant`). The bus incorporates the functionality of both a registrar

[11]**Polylith** is not a bus in the hardware sense, since messages are not broadcast; they are either multicast or sent point-to-point between tools.

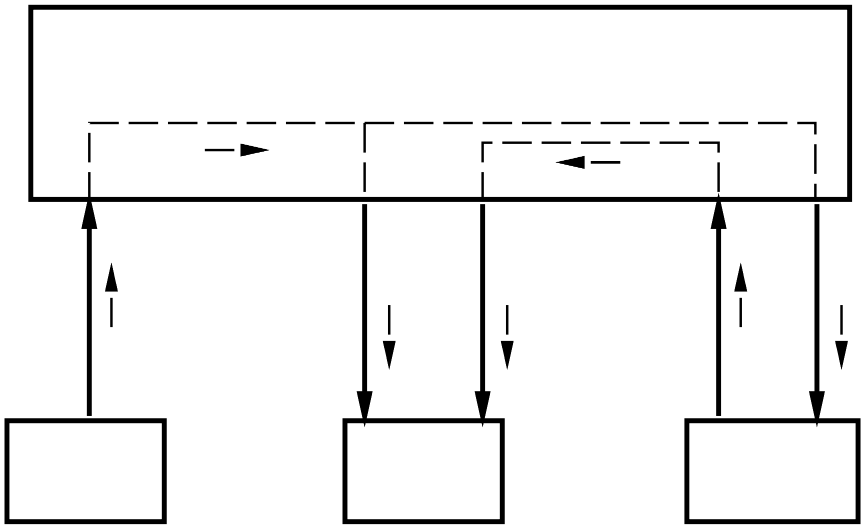

Fig. 5. Components of **Polylith**, plus participants. A message m sent by Tool 1 on a given bus channel is received by Tools 2 and 3 that are listening on that bus channel. Similarly, a message n is sent from Tool 3 to Tool 2 along a different bus channel.

and a router and is therefore a subtype of both Registrar and Router. Software modules are adapted into participants by modifying their source code to replace function calls with bus calls, and then creating a configuration specification in MIL. Registration is accomplished with the Registrar operations Register_Informer and Register_Listener_Polling. (These are both encompassed by **Polylith**'s mh_init function for the initial registration and mh_rebind for re-registrations.) Unregistration is handled by UnRegister (**Polylith**'s mh_rebind and mh_shutdown operations).[12] Messages are sent and received using the Router operations Send (**Polylith**'s mh_write function) and Receive (**Polylith**'s mh_read family of functions).

Polylith supports no MTFs except for the simple filtering provided by bus channels (i.e., a listener connected to a set of bus channels receives only the messages on those channels). There are no user-definable delivery constraints. **Polylith** has no explicit support for groups, so in this mapping, we do not use the EBI framework's Group type; but the participants connected to a given bus channel could be considered an implicit "group." Configuration specifications are written in MIL and define participant types (called "modules" in **Polylith**'s terminology), participant instances ("tools"), message types ("interface statements"), and the Access_Control attributes of message types ("bind statements") that specify which tools are connected to which bus channels. **Polylith** assumes that the configuration specification is never violated; i.e., all specified interconnections are created, and all messages are correctly formed.

[12]A single call of mh_shutdown by any participant unregisters all participants and shuts down the bus.

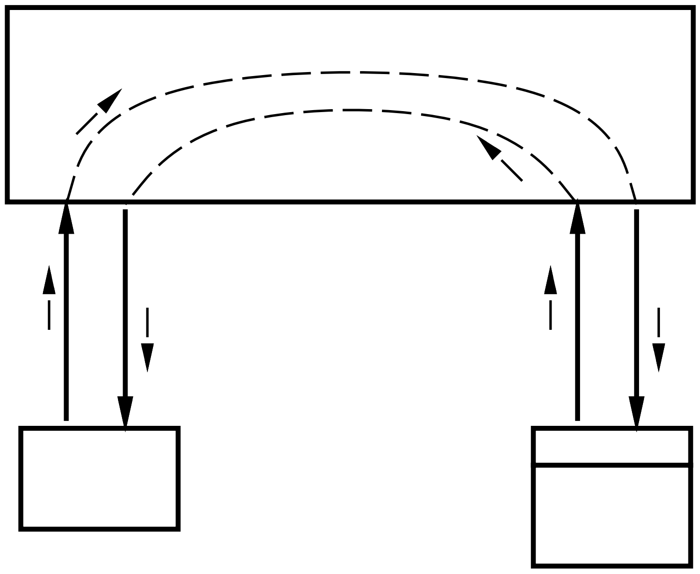

Fig. 6. Components of **CORBA**, plus participants. A client sends a request, m, which is routed to the intended object by an ORB. The object optionally returns a value, r, which is routed back to the client by the ORB.

6.3 CORBA

CORBA is a specification of a client-server architecture for software integration. **CORBA** takes a hybrid approach, incorporating aspects of both event-based integration and remote procedure call (RPC). Viewed as an event-based approach, **CORBA** provides registration, point-to-point message-passing with the aid of routers, grouping, and several (albeit weak) delivery constraints. Viewed as an RPC approach, **CORBA** specifies message-passing operations that return a value from a listener to an informer, just as an RPC can return a value from the callee to the caller. The EBI framework is not intended to model procedure call semantics; so in this case study, some aspects of **CORBA** cannot be mapped to the framework naturally. By modeling the event-based aspects, however, we have captured essential details of **CORBA**'s hundred-page specification, making it possible to perform a meaningful comparison between **CORBA** and other purely event-based integration mechanisms.

In the **CORBA** model, shown in Figure 6, programs, called clients, transmit messages, called requests, to uniquely identifiable, encapsulated entities, called objects. All messages are point-to-point; **CORBA** does not model multicasting. Objects respond to requests by executing operations. Each object has a type, called its interface, that defines the operations and attributes available to clients via that object. Operations are defined by listing their signatures. Attributes are named values that are accessible via "get" and "put" operations. Interfaces are written in **CORBA**'s Interface Definition Language (IDL). IDL is declarative only, containing no control statements. Operations are implemented using a traditional programming language, and a **CORBA** implementation must provide a mapping (stub generation) between IDL and declarations of the programming language. IDL supports various basic and structured types, multiple inheritance, and exception types.

Clients communicate with objects indirectly via servers called object request brokers (ORBs). ORBs provide transparent object location and delivery of requests. **CORBA** also specifies two kinds of repositories: interface repositories, containing object interfaces, and implementation repositories, containing information about the location of objects.

CORBA does not specify the names of most of its operations, leaving them to be determined by each implementation. Consequently, when mapping **CORBA** to the EBI framework, we present few mappings to functions.

In terms of the EBI framework, **CORBA** distinguishes two types of participants: clients and objects (both subtypes of `Participant`). All clients are informers (and may also be listeners) whose purpose is to send requests to objects. All objects are listeners (and may also be informers) whose purpose is to respond to requests from clients. Clients may be objects and vice versa.

The **CORBA** "request" type is a subtype of `Message` with four additional attributes:

(1) `target object`: a reference to the object intended to receive the request.
(2) `operation`: the action to be taken by the receiving object.
(3) A set of zero or more `parameters` of the operation. A parameter consists of a name, a value, and a parameter attribute of `IN`, `OUT`, or `INOUT`.
(4) `request context`: a set of name/value pairs, in which the values are strings, similar to UNIX environment variables. A request context is described as "additional, operation-specific information that may affect the performance of a request" [Digital Equipment Corporation et al. 1993].[13]

A request, like a remote procedure call, can optionally return a result. Thus, **CORBA**'s ORBs have the ability to associate a request (from client to object) with a return value (from the object back to the client). This feature is beyond the capabilities of a router, so it cannot be mapped to the EBI framework.

A software module is adapted to become a **CORBA** client by instrumenting its source code to contact an ORB. A software module is adapted to become a **CORBA** object in two steps:

(1) A software developer writes an IDL wrapper for the module, which is then compiled into stubs.
(2) The IDL wrapper may use different operation names than the ones found in the module. If so, a software developer must reconcile this difference: either by creating a second, intermediate wrapper that sits between the IDL wrapper and the module, or by modifying the module to use the IDL-generated operation names.

ORBs serve as both registrars and routers and are therefore subtypes of both `Registrar` and `Router`. As registrars, they permit objects to register their exis-

[13]It is unclear why request contexts are distinguished from operation parameters. This decision probably reflects UNIX biases: i.e., as environment variables are distinguished from command-line arguments.

tence and clients to register their interest in obtaining services from objects, using `Register_Informer` and either of the operations `Register_Listener_Polling` or `Register_Listener_Active` (depending on the **CORBA** implementation). As routers, ORBs route each client request to an object whose interface specifies that it can satisfy the request. All requests are point-to-point between client and object with assistance by an ORB.

The **CORBA** specification provides no MTFs. Some **CORBA** implementations, however, augment their ORBs with the ability to filter messages. Staying entirely within the **CORBA** specification, it may be possible for an object/participant to model an MTF, but there is no ORB support for allowing such an object to intercept requests intended for another object (e.g., to perform filtering).

CORBA defines two delivery constraints, called execution semantics, that can be associated with a request. "At Most Once" semantics specify that a request will be received by the intended listener exactly once, or else an exception will be raised. "Best Effort" semantics specify that delivery is not guaranteed, and no exception is raised if delivery fails.

CORBA 2.0 provides explicit support for grouping using domains, where a domain is a "set of objects sharing a common characteristic or abiding by common rules" [BNR Europe Limited et al. 1995]. A domain may itself be an object and a member of other domains, thereby supporting composition. There appear to be no operations supported by **CORBA**, however, that operate on the members of a group (e.g., multicasting a request to all objects in a group).

Configuration specifications for **CORBA** are written in IDL, and they specify only **CORBA** object (participant) types.[14] Policies for handling configuration specification violations are limited to several standard exceptions (transient failure, object does not support operation, etc.). IDL provides no way to specify the remaining parts of a configuration specification: MTFs and delivery constraints for participants, instances of participants, or groups.

6.3.1 *CORBA Event Service.* The **CORBA** Event Service [Object Management Group 1994] is an abstraction for event-based integration that has been designed on top of **CORBA**. It provides informers and listeners ("suppliers" and "consumers," respectively) that communicate via routers ("event channels") that support both polled ("pull") and non-polled ("push") message passing. Event channels also act as registrars, allowing consumers and suppliers to register to receive all messages that pass through a given event channel. (That is, event channels broadcast to all registered consumers.) Other semantics of event channels, such as the use of MTFs or delivery constraints, are intentionally unspecified and left up to the designer of the event channel. Suppliers, consumers, and event channels are **CORBA** objects, and communication occurs via **CORBA** requests. As a **CORBA** Service, the Event Service need not be included in a **CORBA** implementation.

6.4 Comparison

The features of **FIELD**, **Polylith**, and **CORBA** (minus the optional Event Service add-on) described in the case studies are summarized in Table 1. The three

[14]There is only a single Client type.

integration mechanisms have a number of notable similarities, some of which are apparent in Figures 4 through 6. All three mechanisms have readily identifiable participants that are adapted by source code modification, wrapping, or both. All of the mechanisms combine the functionality of registrar and router into a single component. Finally, none of the mechanisms provide queries over sets of objects, nor user-specifiable actions to handle configuration specification violations.

The three integration mechanisms differ significantly as well. **FIELD** has only one message type (string), limited user-definable delivery constraints, and no notion of explicit configuration specifications. **Polylith** has no user-definable delivery constraints, no MTFs except filters, no explicit groups, and no manner of handling configuration specification violations. **CORBA** has no multicasting, no MTFs, no user-definable delivery constraints, and no explicit specification of integration object instances.

This comparison points out similarities and differences that are central to the underlying behavior of these three systems, producing a clear, concise means of comparison between mechanisms, without getting bogged down with implementation details. Each row of Table 1 emphasizes an important interoperability issue. Similarities in a row indicate areas of high-level compatibility between mechanisms. Differences in a row call attention to potential interoperability trouble spots, such as the following:

—Of the three mechanisms, only **CORBA** does not support multicasting and filtering (though some of its implementations add these capabilities). Thus, in order for participants from all three mechanisms to receive messages in a consistent manner, an additional layer of abstraction around **CORBA**'s routers (ORBs) may be necessary to provide these features.

—All three mechanisms have different message types. Two points of contention are how **Polylith**'s pointer types should be interpreted by the other mechanisms that do not support pointers, and how to handle structured types so **FIELD** can interpret them as strings.

—If a **FIELD** or **CORBA** informer sends a message to a **Polylith** listener, and the message gets lost in transit, **Polylith** has no means of announcing this failure, since it cannot handle configuration specification violations. This is a problem because the informer expects to see a NULL return value for **FIELD** or raised exception for **CORBA** if a message fails to arrive.

7. CONCLUSIONS

More than fifty event-based integration systems are available today. Their models and terminology vary widely, and their documentation can be hundreds of pages long. Software engineers need to compare these systems, but the state of the art is to do an ad hoc comparison. The EBI framework provides a systematic method of comparison. It identifies common components, their basic features, and likely dimensions along which these components may differ. This provides a concise representation of important similarities and differences between integration mechanisms.

We have demonstrated the EBI framework's effectiveness by modeling three different, well-known integration approaches. By highlighting significant similarities

Table 1. Features of **FIELD**, **Polylith**, and **CORBA**.

(a) Instantiation

Feature	**FIELD**	**Polylith**	**CORBA**
Message Types	String	Simple, structured, and pointer types	Simple, structured, and interface types
Registrar	**Msg** message server	Bus	ORBs
Router	**Msg** pattern matching engine	Bus	ORBs
Message Sending	Multicast	Point-to-point, multicast	Point-to-point
Message Delivery	Non-polling (passive)	Polling (active)	Unspecified
MTFs	Filters, policies	Filtering by bus channel	None specified
Delivery Constraints	Policy priorities	Not user definable	At most once, best effort
Grouping	Participant groups	None	Participant and router groups, called domains
Specify Framework Component Instances?	No	No	No

(b) Adaptation

Feature	**FIELD**	**Polylith**	**CORBA**
Method	Wrap or modify source code	Wrap and modify source code	Wrap and modify source code

(c) Configuration

Feature	**FIELD**	**Polylith**	**CORBA**
Participants	**Msg** clients (executable programs)	Bus tools (executable programs)	Clients and objects (executable programs)
Specify Participant Instances?	No	Yes	No
Configuration Spec	Not user definable	MIL specification	IDL specification
Config Violation Handling	NULL return value	None	Predefined exceptions

and differences between mechanisms, the EBI framework provides guidance for choosing an appropriate mechanism for one's needs. In addition, it aids interoperability by calling attention to areas of compatibility and concern when planning an interoperability strategy. We plan to examine, via experimentation, the extent to which this information can be used directly to support interoperability.

The EBI framework is a reference model, not an implementation guide. An implementation would have to elaborate important issues beyond the instantiation of the mechanism, such as an instance model, runtime system semantics, and languages for MTFs, delivery constraints, and queries. In addition, while our object-oriented model provides a well-understood description of objects and their behavior (i.e., ADTs), it implies that many kinds of objects are first-class. This view may not always be desirable and may have to be addressed directly in an implementation. We also intend additional exploration of scalability. The EBI framework supports composition via groups, allowing the model to scale. We plan to investigate to what extent this composition translates to actual scalability of implementations.

ACKNOWLEDGMENTS

We thank Barbara Lerner, James Purtilo, Steven Reiss, and the reviewers for their insightful comments on earlier versions of this manuscript.

APPENDIX

A. THE TYPE MODEL

This appendix lists our abstract data types and defines the signatures and exceptions of their operations. The type model serves two purposes. The first purpose is to model the parts of the EBI framework, represented by the types `Message`, `Registrar`, `Participant`, etc. The second purpose is to allow the type hierarchy itself to be extended with new types of messages, registrars, participants, etc., during instantiation. This is accomplished using the types `Type`, `Object`, `Attribute`, and `Operation` to model the process of modifying the type model. For instance, a new router type would be created with `Type.Create`, attributes and operations would be added with `Type.Add_Attribute` and `Type.Add_Operation`, and the new type would be linked into the type hierarchy with `Type.Link_To_Supertype` and possibly `Type.Link_To_Subtype`.

All operation parameters are "in" parameters unless otherwise stated. Common exceptions such as "permission denied" are not listed.

A.1 Types for Modifying the Type Model

(1) `MetaType` (no supertypes)

Type `MetaType` is introduced as the root of the hierarchy to avoid a loop in the hierarchy, as discussed in Section 4. Type `MetaType` always exists and cannot be removed from the type hierarchy.

—**Do_Query**(`t: MetaType; q: Query`) return `<MetaType>`;

Retrieve information about a type or instance via associative access. No query language is supplied by the EBI framework; one must be defined in the mechanism specification.

The result of this operation is written as `<MetaType>` to indicate that a query can return a value of any type. An instantiation would provide a more specific type instead of `<MetaType>`.

(2) `Type` (supertypes: `MetaType`)

—**Create**(`supertypes: set of Type`) return `Type`;
Create a type that is a subtype of `supertypes`. Initially, it has no attributes.

—**Destroy**(`t: IN OUT Type`);
Destroy a type, automatically unlinking `t` from all supertypes and subtypes.
Exceptions: `t` is the sole parent of another type and therefore cannot be destroyed.

—**Link_To_Supertype**(`t: IN OUT Type; super: IN OUT Type`);
Add type `super` as a supertype of the type `t`.
Exceptions: `super` is already a supertype of `t`, inheritance policy violation.

—**Unlink_From_Supertype**(`t: IN OUT Type; super: IN OUT Type`);
Remove type `super` as a supertype of the type `t`. It is not possible to unlink a type from all of its supertypes except by invoking **Destroy**.
Exceptions: `super` is not a supertype of `t`, deleting `super` would detach `t` from the type hierarchy.

—**Supertypes_Of**(`t: Type`) return `set of Type`;
Return all of `t`'s supertypes. If `t` is MetaType, return the empty set. Otherwise, `t` is guaranteed to have at least one supertype, by the semantics of **Destroy** and **Unlink_From_Supertype**.

—**Subtypes_Of**(`t: Type`) return `set of Type`;
Return all of `t`'s subtypes. If `t` has no subtypes, then the empty set is returned.

—**Add_Attribute**(`t: IN OUT Type; a: Attribute`);
Add the given attribute to the type `t`. A type cannot have the same attribute twice, nor two attributes with the same name and the same type.
Exceptions: `t` already has the attribute `a`, `t` already has an attribute with the same name and type as `a`.

—**Delete_Attribute**(`t: IN OUT Type; a: Attribute`);
Delete the given attribute from the type `t`.
Exceptions: `a` is not an attribute of `t`.

—**Attributes_Of**(`t: Type`) return `set of Attribute`;
Return all attributes of the given type. If `t` has no attributes, then the empty set is returned.

—**Attribute_From_Name**(`t: Type; att_name: Attribute_Name; att_type: Type`) return `Attribute`;
Return the attribute of the type `t` that has the given name and type.
Note that a type can have two attributes with the same name but different types. Thus, the attribute type must be supplied as an argument in addition to the name. An instantiation could disallow a type from having two attributes of the same name and different types and thereby eliminate the `att_type` argument.
Exceptions: No such attribute found.

—**Add_Operation**(`t: IN OUT Type; op: Operation`);
Add the given operation to the type `t`. The same operation cannot be added twice, and a type cannot have two operations with the same name and the

same signature. A type may have multiple operations with the same name and different signatures; i.e., overloaded operations.
Exceptions: `op` is already an operation of `t`, `t` already has an operation with this name and signature.

—**Delete_Operation**(`t: IN OUT Type; op: Operation`);
Delete the given operation from the type `t`.
Exceptions: `op` is not an operation of `t`.

—**Operations_Of**(`t: Type`) return `set of Operation`;
Return all operations of the given type. If `t` has no operations, then the empty set is returned.

—**Operation_From_Name**(`op_name: Operation_Name; op_type: Type`) return `Operation`;
Return the operation of the type `t` that has the given name and signature (type).
Note that a type can have two operations with the same name but different types. Thus, the operation type must be supplied as an argument in addition to the name. An instantiation could disallow a type from having two operations of the same name and different types and thereby eliminate the `op_type` argument.
Exceptions: No such operation found.

—**Identical**(`t1, t2: Type`) return `Boolean`;
Determine whether two types are physically the same type.

(3) `Object` (supertypes: `MetaType`). Note that type `Object` is an instance of type `Type`.

—**Create**(`t: Type`) return `Object`;
Create a new object of type `t`.

—**Destroy**(`obj: IN OUT Object`);
Destroy an object.
Exceptions: Object is already destroyed.

—**Set_Type**(`obj: Object; t: Type`);
Set the type of object `obj` to be `t`. By default, the object loses the values of all of its old attributes.
Exceptions: Mechanism specification violation.

—**Get_Type**(`obj: Object`) return `Type`;
Get the type of object `obj`.

(4) `Attribute` (supertypes: `Object, MetaType`)

—**Create**(`n: Attribute_Name; t: Attribute_Type`) return `Attribute`;
Create a new attribute of the given name and type.

—**Destroy**(`a : Attribute`);
Destroy the given attribute.
Exceptions: Attribute is already destroyed, attribute sharing violation.

—**Set_Value**(`a: IN OUT Attribute; v: Attribute_Value`);
Set the value of the given attribute to the given value.
Exceptions: Value out of range, type mismatch.

—**Get_Value**(`a: Attribute`) return `Attribute_Value`;
Get the value of the given attribute.

Exceptions: Attribute has no value.
—**Equal**(`a1, a2: Attribute`) return `Boolean`;
Determine whether two attributes have the same type and value.

(5) `Operation` (supertypes: `Object, MetaType`)
—**Create**(`n: Operation_Name; t: Type`) return `Operation`;
Create a new operation of the given name and type. The names and types of the operands are the attributes of `Operation_Type`.
—**Destroy** is inherited from `Object`.
—**Set_Operand_Value**(`op: Operation; param: Attribute_Name; param_type: Type; value: param_type`);
A convenient renaming of

```
Attribute.Set_Value(
   Type.Attribute_From_Name(op, param, param_type),
   value);
```

—**Get_Operand_Value**(`op: Operation; param: Attribute_Name; param_type: Type`) return `param_type`;
A convenient renaming of

```
Attribute.Get_Value(
   Type.Attribute_From_Name(op, param, param_type));
```

A.2 Types for Modeling the Framework

(1) `Message` (supertypes: `Object, MetaType`)
—**Create**(`n: Message_Name; t: Message_Type`) return `Message`;
Create a new message of the given name and type. The names and types of the message parameters are the attributes of `Message_Type`.
—**Destroy** is inherited from `Object`.
—**Set_Parameter_Value**(`m: Message; param: Attribute_Name; param_type: Type; value: param_type`);
A convenient renaming of

```
Attribute.Set_Value(
   Type.Attribute_From_Name(m, param, param_type),
   value);
```

—**Get_Parameter_Value**(`m: Message; param: Attribute_Name; param_type: Type`) return `param_type`;
A convenient renaming of

```
Attribute.Get_Value(
   Type.Attribute_From_Name(m, param, param_type));
```

—**Message_From_Name**(`msg_name: Message_Name; msg_type: Type`) return `Message`;
Return the message of the type `msg_type` that has the given name and type. Note that a type can have two messages with the same name but different types. Thus, the message type must be supplied as an argument in addition to the name. An instantiation could disallow a type from having two messages of the same name and different types and thereby eliminate the `msg_type` argument.
Exceptions: No such message found.

If an instantiation disallows a type from having multiple attributes with the same name and different types, then the `param_type` argument can be eliminated from these operations.

Type `Message` has at least three attributes, as described in Section 5.2:

—`Synchronization`

—`Access_Control`

—`Delivery_Constraint`

(2) `Sender` (supertypes: `Object`)
This type exists only to provide a common supertype for `Informer` and `Router`, so they can both send messages using the `Router.Send` operation.

(3) `Participant` (supertypes: `Object`).
—**Get_Router**(`p: Participant`) return `Router`;
Return the router used by participant instance `p`.
—**Set_Router**(`p: IN OUT Participant; r: Router`);
Set the router used by participant instance `p`.
Provisions: Only a registrar can invoke this operation.
—**Get_Registrar**(`p: Participant`) return `Registrar`;
Return the registrar used by participant instance `p`.
—**Set_Registrar**(`p: IN OUT Participant; r: Registrar`);
Set the registrar used by participant instance `p`.
Provisions: Only a registrar can invoke this operation.

(4) `Informer` (supertypes: `Participant, Sender`).
—**Messages_Of**(`i: Informer`) return `set of Message`;
Return all messages that can be sent by the given informer. If `i` has no messages, then the empty set is returned.

(5) `Listener` (supertypes: `Participant`).
(No additional operations.)

(6) `Registrar` (supertypes: `Object`). Note that type `Registrar` is an instance of type `Framework_Type`.
—**Register_Informer**(`r: Registrar; i: Informer; msgs: set of Message`) return `Router`;
Register an instance of an informer. The returned router has the **Send** function that the informer needs for communication. (Note that this can be a logical router instead of a physical router.)
Exceptions: Configuration specification violation, type error.
—**Register_Listener_Polling**(`r: Registrar; l: Listener`) return `Router`;
Register a listener to receive messages via polling. Message transforming functions can be registered later to filter or otherwise modify the incoming messages.
The returned router has the **Receive** function that the informer needs for communication. (Note that this can be a logical router instead of a physical router.)
Exceptions: Configuration specification violation, type error, polling model not supported.

—**Register_Listener_Active**(`r: Registrar; l: Listener; op: Operation`);
Register a listener to receive messages actively. The given operation is invoked whenever the listener has a message delivered. Message transforming functions can be registered later to filter or otherwise modify the incoming messages.
Exceptions: Active model not supported.

—**Register_MTF**(`r: Registrar; p: Participant; f: MTF`);
Register this MTF to be evaluated whenever a message is sent by `p` (if `p` is an informer) or received by `p` (if `p` is a listener).

—**Register_MTF_Ordered**(`r: Registrar; p: Participant; f: MTF; i: Natural`);
Register this MTF to be the `i`th MTF evaluated whenever a message is sent by `p` (if `p` is an informer) or received by `p` (if `p` is a listener). The numeric positions of the original `i`th, `(i+1)`st, ... MTFs of that participant (if they exist) are automatically incremented by 1.
The order of evaluation of a participant's MTFs can be significant. If it is not, then **Register_MTF** should be used instead of this operation.
Exceptions: Config specification violation

—**MTFs_Of**(`r: Registrar; p: Participant`) return `sequence of MTF`;
List the MTFs associated with the given participant. This is necessary so a participant can know the order in which its MTFs are being evaluated so it can modify the sequence with **Register_MTF_Ordered**. Other information about a registrar can be obtained using **Type.Do_Query**.

—**Register_Delivery_Constraint**(`r:Registrar; d:Delivery_Constraint`);
Register a delivery constraint. A delivery constraint is passed as a single parameter because it can be arbitrarily complex, containing information about messages, participants, types, attributes, timing, and more.
Exceptions: Config specification violation, delivery constraint is inconsistent with a previously registered delivery constraint.

—**UnRegister**(...)
Each **Register_xxx** operation above has a corresponding **UnRegister_xxx** operation.
Exceptions: The information to be unregistered is not currently registered, config specification violation.

(7) `Router` (supertypes: `Object, Sender`). Note that type `Router` is an instance of type `Framework_Type`.

—**Message_Waiting**(`r: Router; l: Listener`) return `Boolean`;
Is a message waiting for the listener?
Exceptions: Polling model not supported.

—**Receive**(`r: Router; l: Listener`) return `Message`;
Receive the next message intended for the listener. The decision of which message is returned by **Receive** when called by listener `l` is dependent on `l`'s registration information, which caused particular router connections to be made, certain synchronization to be in effect, etc.
Use of **Receive** implies a polling model for message delivery. For an active model, see **Registrar.Register_Listener_Active**.

Exceptions: No message available, message type mismatch, polling model not supported.

—**Send**(`r: Router; s: Sender; m: Message`);
Send the message from a Sender (informer or router) to a router for delivery. Its delivery will proceed in whatever manner is specified in the message's attributes (synchronization, delivery constraints, access control).
Exceptions: Recipient not found, delivery constraint violated, message type mismatch.

(8) `MTF` (supertypes: `Object`)
—**Transform**(`f: MTF; m: Message; l: Listener`) return `poset of (Message, Listener)`;
Apply the MTF to `m` and `l`, as defined in Section 5.5. The MTF's state may change as a result.

(9) `Delivery_Constraint` (supertypes: `Object`)
Delivery constraints have no operations of their own. **Router.Send** raises an exception when a delivery constraint is violated.

(10) `Group` (supertypes: `Object`)
—**Add_Member**(`g: Group; obj: Object`);
Add an object to a group.
—**Remove_Member**(`g: Group; obj: Object`);
Remove an object from a group.
Exceptions: The object is not in this group.
—**Is_Member**(`g: Group; obj: Object`) return `Boolean`;
Is the object a member of the group?
—**Members_Of**(`g: Group`) return `set of Object`;
List the members of a group.

(11) Other types
Certain common types (e.g., `integer`, `set`, `sequence`, `poset`, and various types of names) are used in the specification above but are not formally defined here.

B. DETAILED MAPPING OF FIELD TO THE TYPE MODEL

This section presents a detailed mapping of **FIELD** to our descriptive type model. Details on **FIELD** types and operations can be found in the **FIELD** manual [Reiss 1996] and in Section 6.1.

The creation of the types used in this appendix are modeled by the operations `Create`, `Add_Attribute`, `Add_Operation`, and `Link_From_Supertype` of type `Type`, as explained in Appendix A. We do not present this modeling as it is straightforward. For brevity, we also do not present mappings for most `Create` and `Destroy` operations.

Some framework operations require formal parameters that the corresponding **FIELD** operations do not, because the framework assumes that many objects are first class (as mentioned in Section 7). For example, some operations on **Msg** do not require **Msg** as a parameter, but the corresponding framework operations do. Similarly, a **FIELD** tool can send a message without listing itself as a parameter to the send operation, whereas the framework's `Router.Send` requires the sender (informer) to be listed as a parameter. To get around these minor modeling difficulties, we have borrowed the `THIS` construct from C++[Lippman 1990], allowing the callee to be referenced within the body of the operation, and introduced an analogous construct, `CALLER`, to allow the caller to be referenced.

Some parameters of **FIELD** are representative of implementation details; for example, the fact that a file is used to implement a group. We have noted this where applicable and not mapped such parameters to our type model.

B.1 FIELD Messages

We use the type `FIELD_Message` to distinguish **FIELD** messages from instances of the EBI framework type `Message`.

```
type FIELD_Message (supertypes: String, Message);
```

In **FIELD**, messages are created and destroyed using C library functions such as `malloc` and `free` [Kernighan and Ritchie 1990]. Message parameter values also are set and retrieved using C library functions such as `sprintf` and `sscanf`. We model this by having `FIELD_Message` inherit all its operations `Create` and `Destroy` from type `String`, and `Set_Parameter_Value` and `Get_Parameter_Value` from type `Message`.

B.2 Tools

Type `Tool` can have any attributes and operations that the **FIELD** engineer desires.

```
type Tool (supertypes: Participant);
```

B.3 Message Patterns

Message patterns represent MTFs that match a message against a string (the pattern). For simplicity of notation,

```
type Message_Pattern (supertypes: MTF, String)
attributes
   pattern       : String;
   registered_by : Listener;
operations
   Transform(m : Message; lis : Listener)
      return poset of (Message, Listener)
   begin
      if (m matches pattern) then
         return((m, registered_by));  /* Poset of size 1 */
      else
         return(empty poset);
   end;
end Message_Pattern;
```

Note that listener `lis` is always the listener that registered the pattern.

B.4 The Message Server, Msg

```
type MSG (supertypes: Registrar, Router)
operations
```

```
   MSGregister(handle : MSG; pattern : Message_Pattern,
               function operation(...))
   begin
      Registrar.Register_Listener_Active(handle, CALLER, operation);
      Registrar.Register_MTF(handle, CALLER, pattern);
   end;

   MSGsend(handle: MSG; m : FIELD_Message)
   begin
      m.synchronization := asynchronous;
      Router.Send(handle, CALLER, m);
   end;

   MSGcall(handle: MSG; m : FIELD_Message) return FIELD_Message
   begin
      m.synchronization := synchronous;
      Router.Send(handle, CALLER, m);
      return(Router.Receive(handle, CALLER));
   end;

   MSGcallback(handle : MSG; function op(...); data : pointer;
               m : FIELD_Message)
   begin
      Registrar.Register_Listener_Active(handle, CALLER, op);
      Router.Send(handle, CALLER, m);
      Registrar.UnRegister_Listener_Active(handle, CALLER, op);
   end;

   MSGreply(reply : FIELD_Message)
   begin
      Router.Send(THIS, CALLER, reply);
   end;

   MSGconnect()
   begin
      Registrar.Register_Informer(...);   /* see below */
   end;
end MSG;
```

In `MSGreply`, we have omitted the first **FIELD** parameter because it represents an implementation detail. The first parameter, an integer, is a unique ID representing the original message requiring the reply. In our mapping, we assume that the router (**Msg**) keeps track of such details internally.

In `MSGregister`, we have omitted the fourth and fifth **FIELD** parameters since they are implementation details. The fourth parameter, an integer, represents the number of arguments in `pattern`, and the last parameter, an array of void pointers, represents default values for the pattern.

We do not include an explicit, parameter-by-parameter mapping of `MSGconnect` because its parameters in **FIELD** are highly implementation-specific. For example, **FIELD** uses a filename (a string) to represent the connection between the invoking tool and its message server (presumably more information is stored in the file), and a `NULL` value means to use **Msg** by default. In addition, neither **Msg** nor the invoking tool are directly represented by parameters. Nevertheless, the purpose of `MSGconnect` — to permit the invoking tool to send messages to the message server — is clearly analogous to that of `Registrar.Register_Informer`.

B.5 Groups

The only groups supported by **FIELD** are participant groups, called "message groups."

```
type Message_Group (supertypes: Group)
operations
   MSGconnect(lock : String; group : Group) return MSG
   begin
      Group.Add_Member(group, CALLER);
      return(Group.Get_Router(group));
   end;
end Message_Group;
```

As noted in Section B.4, **FIELD**'s `MSGconnect` function has very implementation-specific parameter types. The first parameter, `lock`, is not needed for this use of `MSGconnect`, so we do not model it. Also, note that `Get_Router` is inherited by type `Group` from type `Participant`.

B.6 The Policy Tool

Policies are MTFs that are written in a policy language (defined in [Reiss 1996]).

```
type Policy (supertypes: MTF)
   policy_statement : Policy_Language_Statement;
operations
   Transform(m : FIELD_Message; lis : Listener)
      return poset of (FIELD_Message, Listener)
   begin
      /* Apply the policy_statement to the message.
         The listener parameter is ignored. */
   end;
end Policy;
```

The Policy Tool maintains policies and applies them to messages that are sent to it by **Msg**. We represent the Policy Tool instance as `The_Policy_Tool`.

```
type Policy_Tool (supertypes: Registrar, Router, Participant)
operations
   Register_Policy(p : Policy)
   begin
      Registrar.Register_MTF(The_Policy_Tool, CALLER, p);
   end;
   Register_Policy_Priority(p : Policy; priority : Natural)
      d : Delivery_Constraint;
   begin
      d := Delivery_Constraint.Create(p, priority);
      Register_Policy(p);
      Registrar.Register_Delivery_Constraint(The_Policy_Tool, d);
   end;
end Policy_Tool;
```

An MTF is used to redirect each message to go to the Policy Tool.

```
type To_Policy (supertypes: MTF)
operations
   Transform(m : FIELD_Message; lis : Listener)
      return poset of (FIELD_Message, Listener)
   begin
      return((m, The_Policy_Tool));  /* poset of size 1 */
   end;
```

REFERENCES

Allen, R. and Garlan, D. 1994. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering* (Sorrento, Italy, May 1994), pp. 71–80.

Apple Computer, Inc. 1991. *Inside Macintosh*, Volume VI. Addison-Wesley Publishing Company, Inc, New York. Chapters 4-8.

Arnold, J. E. and Memmi, G. 1992. Control integration and its role in software integration. In EC2 Ed., *Proceedings of the Fifth International Conference Toulouse '92 "Software Engineering & its Applications"* (Toulouse, Dec. 1992), pp. 107–118. Also Bull System Products Research Report RAD/USARL/92021, December 1992.

Barnes, J. 1995. *Programming in Ada 95*. Addison-Wesley Publishing Company.

Barrett, D. J. 1993. SDL BMS: A simple broadcast message server. Arcadia Document UM-93-03 (Oct.), University of Massachusetts, Software Development Laboratory, Computer Science Department. Available from *laser@laser.cs.umass.edu*.

Beach, B. W. 1992. Connecting software components with declarative glue. In *Proceedings of the Fourteenth International Conference on Software Engineering* (Melbourne, Australia, May 1992), pp. 120–137.

Belkhatir, N., Estublier, J., and Melo, W. 1994. *ADELE-TEMPO: An Environment to Support Process Modelling and Enaction*, Chapter 8, pp. 187–222. Advanced Software Development Series. John Wiley & Sons Inc.

Ben-Shaul, I. and Kaiser, G. E. 1995. *A Paradigm for Decentralized Process Modeling*. Kluwer Academic Publishers, Boston MA.

Birman, K. P. 1993. The process group approach to reliable distributed computing. *Communications of The ACM 36*, 12 (Dec.), 37–53.

BNR Europe Limited, Digital Equipment Corporation, Expersoft Corporation, Hewlett Packard Corporation, IBM Corporation, ICL, plc, IONA Technologies, and SunSoft, Inc. 1995. CORBA 2.0/Interoperability. OMG TC Document 95.3.xx (March), Object Management Group, Framingham, MA. Revision 1.8.

Boudier, G., Minot, R., and Thomas, I. M. 1989. An overview of PCTE and PCTE+. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Feb. 1989), pp. 248–257. ACM Press. Published as SIGPLAN Notices, volume 24, number 2.

BROCKSCHMIDT, K. 1995. *Inside OLE.* Microsoft Press.

CHRYSANTHIS, P. AND RAMAMRITHAM, K. 1990. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *ACM SIGMOD International Conference on Management of Data* (May 1990), pp. 194–203.

CLEMM, G. AND OSTERWEIL, L. 1990. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems 12*, 1 (Jan.), 1–25.

COMMODORE-AMIGA INCORPORATED. 1992. *Amiga ROM Kernel Reference Manual: Libraries* (Third ed.). Addison Wesley Publishing Company.

DELLAFERA, C. A., EICHIN, M. W., FRENCH, R. S., JEDLINSKY, D. C., KOHL, J. T., AND SOMMERFELD, W. E. 1988. Zephyr notification service. In *USENIX Conference Proceedings* (Dallas, TX, Winter 1988). USENIX.

DIBELLA, K. S. 1994. PEN: Event notification in distributed environments. Technical Report UM-PILGRIM-94-01 (Aug.), Project Pilgrim, University of Massachusetts. http://www.pilgrim.umass.edu/pub/pilgrim/papers/pen_conf_paper.ps.

DIGITAL EQUIPMENT CORPORATION, HEWLETT-PACKARD COMPANY, HYPERDESK CORPORATION, NCR CORPORATION, OBJECT DESIGN, INC., AND SUNSOFT, INC. 1993. *The Common Object Request Broker: Architecture and Specification.* Object Management Group and X/Open. Revision 1.2, incorporated as part of CORBA 2.0.

FROMME, B. 1990. HP Encapsulator: Bridging the generation gap. *Hewlett-Packard Journal 41*, 3 (June), 59–68.

GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. 1994. Exploiting style in architectural design environments. In D. WILE Ed., *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering* (New Orleans, LA, Dec. 1994), pp. 175–188. ACM Press. Published as *SIGSOFT Notes* 19(5).

GARLAN, D. AND ILIAS, E. 1990. Low-cost adaptable tool integration policies for integrated environments. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments* (Dec. 1990), pp. 1–10.

GARLAN, D. AND NOTKIN, D. 1991. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of VDM'91: Formal Software Development Methods* (Oct. 1991). Also appeared as Carnegie-Mellon University technical report CMU-CS-91-114, March 1991.

GARLAN, D. AND SCOTT, C. 1993. Adding implicit invocation to traditional programming languages. In *Proceedings of the 15th International Conference on Software Engineering* (May 1993), pp. 447–455.

GAUTIER, B., LOFTUS, C., SHERRATT, E., AND THOMAS, L. 1995. Tool integration: Experiences and directions. In *Proceedings of the 17th International Conference on Software Engineering* (1995), pp. 315–324.

GERETY, C. 1990. HP SoftBench: A new generation of software development tools. *Hewlett-Packard Journal 41*, 3 (June), 48–59.

GOLDBERG, A. 1984. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA.

HENNIE, F. C. 1966. On-line Turing machine computations. *IEEE Transactions on Electronic Computers EC-15*, 1 (Feb.), 35–44.

HEWITT, C. AND INMAN, J. 1991. DAI betwixt and between: From "intelligent agents" to open systems science. *IEEE Transactions on Systems, Man, and Cybernetics 21*, 6 (November/December), 1409–1419.

HOMER, P. T. AND SCHLICHTING, R. D. 1994. A software platform for constructing scientific applications from heterogeneous resources. *Journal of Parallel and Distributed Computing 21*, 3 (June), 301–315.

JAGANNATHAN, V., DODHIAWALA, R., AND BAUM, L. S. Eds. 1989. *Blackboard Architectures and Applications*, Volume 3 of *Perspectives in Artificial Intelligence.* Academic Press, Inc.

KAISER, G. E. AND GARLAN, D. 1987. Melding software systems from reusable building blocks. *IEEE Software 4*, 4 (July), 17–24.

Kernighan, B. and Ritchie, D. 1990. *The C Programming Language* (Second ed.). Prentice-Hall.

Koymans, R. 1992. *Specifying Message Passing and Time-Critical Systems With Temporal Logic*. Springer-Verlag.

Krishnamurthy, B. and Barghouti, N. S. 1993. Provence: A process visualization and enactment environment. In I. Sommerville and M. Paul Eds., *Proceedings of the 4th European Software Engineering Conference* (Garmisch-Partenkirchen, Germany, Sept. 1993), pp. 451–465. Springer-Verlag. Published as *Lecture Notes in Computer Science* 717.

Liang, T.-P., Lai, H., Chen, N.-S., Wei, H., and Chen, M. C. 1994. When client/server isn't enough: Coordinating multiple distributed tasks. *Computer 27*, 5 (May), 73–79.

Lippman, S. B. 1990. *C++ Primer*. Addison-Wesley Publishing Company.

Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., and Mann, W. 1995. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering 21*, 4 (April), 336–355.

MacBride, A. and Susser, J. 1996. *Byte Guide to OpenDoc*. Osborne/McGraw-Hill, Berkeley, CA.

Minsky, N. H. and Rozenshtein, D. 1990. Configuration management by consensus: An application of law-governed systems. In R. N. Taylor Ed., *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments* (Irvine, CA, Dec. 1990), pp. 44–55. ACM Press. Published as *SIGSOFT Notes* 15(6).

Mishra, S., Peterson, L. L., and Schlichting, R. D. 1991. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report 91-32 (Nov.), University of Arizona, Department of Computer Science. Available from *tr_libr@cs.arizona.edu*.

Mishra, S., Peterson, L. L., and Schlichting, R. D. 1993. Experience with modularity in consul. *Software — Practice and Experience 23*, 10 (Oct.), 1059–1076.

Montangero, C. and Ambriola, V. 1994. *OIKOS: Constructing Process-Centred SDEs*, Chapter 6, pp. 131–151. Advanced Software Development Series. John Wiley & Sons Inc.

Notkin, D., Garlan, D., Griswold, W. G., and Sullivan, K. 1993. Adding implicit invocation to languages: Three approaches. In S. Nishio and A. Yonezawa Eds., *Object Technologies for Advanced Software: Proceedings of the First JSSST International Symposium* (Kanazawa, Japan, Nov. 1993), pp. 489–510. Springer-Verlag.

Object Management Group. 1994. *Common Object Services Specification, Volume 1* (First ed.). Object Management Group.

Patrick, Sr., P. B. 1993. CASE integration using ACA services. *Digital Technical Journal 5*, 2 (Spring), 84–99.

Purtilo, J. and Hofmeister, C. 1991. Dynamic reconfiguration of distributed systems. In *Proceedings of the International Conference on Distributed Computing Systems* (1991), pp. 560–571.

Purtilo, J. M. 1994. The Polylith software bus. *ACM Transactions on Programming Languages and Systems 16*, 1 (Jan.), 151–174.

Reiss, S. P. 1990. Connecting tools using message passing in the FIELD environment. *IEEE Software 7*, 4 (July), 57–67.

Reiss, S. P. 1996. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers.

Schantz, R. E., Thomas, R. H., and Bono, G. 1986. The architecture of the Cronus distributed operating systems. In *Proceedings of the Sixth International Conference on Distributed Computing Systems* (Cambridge, MA, May 1986), pp. 250–259.

Siewiorek, D. P. and Swarz, R. S. 1992. *Reliable Computer Systems: Design and Evaluation* (second ed.)., Chapter 3, pp. 79–84, 201–219. Digital Press.

Spivey, J. M. 1989. *The Z Notation: A Reference Manual*. Prentice Hall International.

Steiner, J. G., Neuman, B. C., and Schiller, J. I. 1988. Kerberos: An authentication service for open network systems. In *Usenix Conference Proceedings* (Dallas, Texas, Feb. 1988), pp. 191–202.

Sullivan, K. J. and Notkin, D. 1992. Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology 1*, 3 (July), 229–268.

SunSoft, Inc. 1992. *ToolTalk 1.0.2 Programmer's Guide.* Sun Microsystems, Inc.

SunSoft, Inc. 1993. *The ToolTalk Service: An Inter-Operability Solution.* Prentice-Hall.

Thomas, I. and Nejmeh, B. A. 1992. Definitions of tool integration for environments. *IEEE Software 9*, 2 (March), 29–35.

Wasserman, A. I. 1990. Tool integration in software engineering environments. In F. Long Ed., *Software Engineering Environments*, pp. 137–149. Springer-Verlag. Proceedings of the International Workshop on Environments, Chinon, France, September 1989. Published as Lecture Notes in Computer Science 467.

Zamara, C. and Sullivan, N. 1991. *Using ARexx On The Amiga.* Abacus.