An Event-Based Software Integration Framework

Daniel J. Barrett
Lori A. Clarke
Peri L. Tarr
Alexander E. Wise

CMPSCI Technical Report 95–48
May 19, 1995

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Advanced Research Projects Agency under grant F30602-94-C-0137.

An Event-Based Software Integration Framework*

Daniel J. Barrett Lori A. Clarke Peri L. Tarr
Alexander E. Wise
Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, MA 01003
E-mail: {barrett,clarke,tarr,wise}@cs.umass.edu

Version 3.0†
May 19, 1995

Abstract

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 products. We attempt to address these shortcomings by specifying a *generic, event-based integration framework* that provides a flexible, object-oriented model for discussing and comparing event-based integration approaches. The 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 popular integration mechanisms: FIELD, Polylith, and CORBA.

Keywords: integration, interoperability, event-based notification, reactive integration, reference model, FIELD, Polylith, CORBA

1 Introduction

Integration, the process by which multiple software modules (programs, subprograms, collections of subprograms, etc.) are made to cooperate, is a serious concern in the construction of large software systems. Approaches to integration range from loose, in which modules have little or no knowledge of one another, to tight, in which modules require much knowledge about one another. The looser the integration, the less impact there is 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 products are available today (e.g., [CCC+93, Dig, Ger89, JR93, Pat93, Pur90, Rei90]).

†Supersedes the version of September 19, 1994.

*This work was supported by the Advanced Research Projects Agency under Grant F30602-94-C-0137.

Unfortunately, no consistent model for describing event-based integration exists; 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, server process, manager process, event manager, object request broker, distributor, or software bus. 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, event-based integration framework*. This framework is <u>not</u> "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.

The framework models only event-based software integration. It is not intended to model other integration techniques, such as various kinds of procedure call (e.g., single program, remote procedure call), shared repository or central database (e.g., ECMA **PCTE** [LM93], **EAST** [GLST95]), and stream-based (e.g., pipes). It is unrealistic to expect a single model to subsume all possible integration models. The framework, however, does support many common and useful integration models.

We begin in Section 2 by describing related work in event-based software integration. Section 3 gives a brief overview of the framework. Section 4 defines a descriptive type model that is utilized in the detailed description of the framework, which follows in in Section 5. Section 6 demonstrates the framework's effectiveness by describing and comparing three existing integration products: **FIELD** [Rei90], **Polylith** [Pur90], and **CORBA** [CCC+93]. Finally, an appendix describes the type model in detail.

2 Related Work

Existing approaches to event-based software integration can be examined across at least four orthogonal dimensions: methods of communication, specification of module interactions, 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[1] 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 a value to the caller. In the application-to-application approach, application programs have unique ID's and transmit messages to one another using these ID's as addresses. This approach is the basis of the **CORBA** specification [CCC$^+$93]. It is also common on personal computers, since the simplifying assumptions of "one user per machine" or "one invocation of a given program at a time" can often be made (e.g., [App91, Com91]).

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, the sender is unaware of the recipients. Software modules express 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** [Rei90]. Today, implicit invocation is used in many commercial products such as Hewlett-Packard's **SoftBench** [Ger89], Sun's **ToolTalk** [JR93] and DEC's **FUSE** [Dig]. It has also been added to some programming languages (e.g., [NGGS93]). 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** [FHC$^+$93, Pur90].

Our 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.[2]

Specification Of Module Interactions: Different integration approaches specify module interactions at various levels of abstraction, ranging from low-level to high-level. At the lowest level, the source code of each module is modified to allow the module to communicate with other modules, and there is no explicit specification of interactions that exists outside the modules. A higher-level approach is encapsulation, or wrapping. Instead of (or in addition to) modifying the source code of

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

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

a module, an additional layer of software, called a wrapper, is created for the module. A wrapper provides a module with an abstract interface of input and output operations, and all communication to and from the associated module is accomplished via that interface. **ToolTalk** [JR93], for example, combines source code modification with simple encapsulation that specifies module input and output operations. **SoftBench** [Ger89] provides a more powerful, procedural encapsulation language [Fro89] that is rich enough to make source code modification unnecessary. **Inscape** [Per89] provides a language of input and output operations plus preconditions and postconditions on them, supporting automated reasoning about module correctness.

The next level of abstraction is the specification of not only the module interfaces but also the connections between modules. **Polylith** [Pur90] uses 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.

The highest level of abstraction, software architecture, adds support for the specification of many types of connections, usually called connectors. (**Polylith**, in contrast, provides only two types of connections: unidirectional and bidirectional data transmission, with no additional semantics.) Connectors are often first-class entities and may be user-specified. **Meld** [KG87], **ACME** [AG94], and **Rapide** [LKA$^+$95] are examples of software architecture models. Software architecture models address a higher level of abstraction and a broader range of architectures than our framework does. Thus, the two approaches are not directly comparable. In particular, software architecture presupposes important details that our framework explores. For example, **ACME** has been used to model a simple "event-based" architecture [GAO94], 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).

Our framework is independent of the method of specifying module interactions, be it source code modification, encapsulation, or external specification.

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 interconnections to be changed while the modules are executing; however, reasoning about the behavior of large, dynamic systems can be very difficult.

The original **Polylith** [Pur90] supported only static specification of interactions, as modules had to be terminated and restarted to change their interconnections.[3] The **SDL BMS** [Bar93] supports only dynamic specification of interactions, allowing modules to register and unregister for service dynamically. **ToolTalk** [Sun92] supports both static and dynamic specification of interactions. Our framework models both static and dynamic specification of behavior.

[3]The latest specification of **Polylith** models dynamism (e.g., [PH91]).

Naming Issues: In order for a module to receive messages, it must somehow be identifiable. There is a spectrum of approaches to 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 receiving a multicast message. For example, **FIELD** [Rei90] 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** [Bir93] allows multiple modules to be grouped under an alias, so any message sent to that 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** [CCC+93] requires each software module to have a globally unique name, and each message must contain its intended recipient's name. **ARexx** [Com91] 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. Both **CORBA** and **DCE** [BGH+93] use name servers to locate receivers that are distributed on multiple machines.

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.

Our framework can model all of these methods of naming.

3 Overview of the Framework

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

Participants and Framework Components: In the 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.

Participants interact by way of 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*. Participants may also unregister and re-register as needed. Using information from the registrar, a *router* delivers messages from

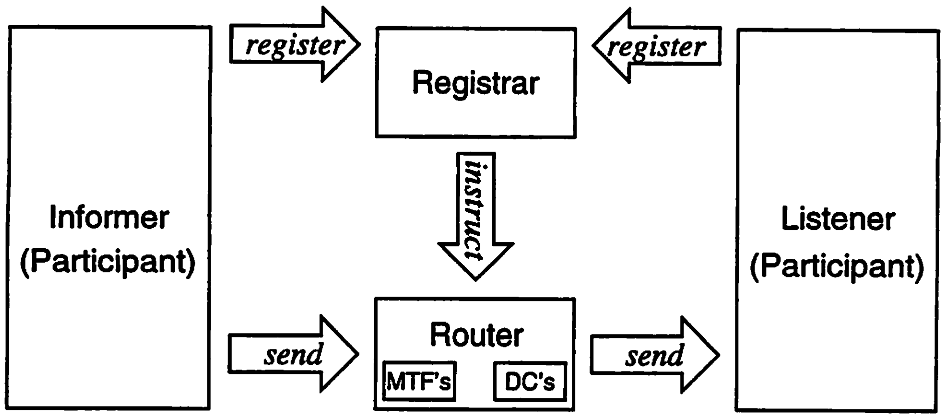

Figure 1: Relationships among informer, listener, registrar, router, message transforming functions (MTF's) and delivery constraints (DC's).

informers to their intended listeners.

The 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* (MTF's) defined within a router can dynamically alter messages sent to listeners. A simple type of MTF commonly found in integration products 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*, 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 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 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 the 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 generic, event-based integration framework in which participants communicate. Each represents a basic aspect of integration. Participants are the interacting software modules. Registration distinguishes modules that are interacting from those that are not. 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 generic 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 instantiated 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 [NGGS93]. The design decisions involved in choosing an adaptation method are well studied [NGGS93, SN92] 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 type of 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 detailed discussion of the generic, event-based integration framework in Section 5, we define a descriptive type model. This type model allows us to reason about the parts of the framework as black boxes (abstract data types). It also provides a convenient way of representing certain relationships between the various parts of the framework, as modeled by subtyping and inheritance.

A *type* is defined as a named set of attributes, operations, messages, and roles. An *attribute* is a named, typed value. An *operation* is a named, invokable entity. A *message* is a named, typed unit of data. Roles are explained below. The attributes, operations, messages and roles of a type are referred to collectively as its *fields*.

A type inherits all fields from all of its supertypes via multiple inheritance.[4] For example, if a type t has a supertype s, and s has an attribute a, then t also has an attribute a. To avoid a cycle in the type hierarchy (types have fields, and fields have types), the type `MetaType` is introduced as the root.

The multiple inheritance model is extended through the use of *roles*, which are optional supertypes. An instance of a type may choose, for each of its roles, whether or not to inherit *all* fields of that role.[5] By default, a type inherits the fields of all of its roles. Roles are similar to variants in a programming language (e.g., Ada [Bar84]). Roles are also similar to subjects in the subject-oriented programming model [HO93]. Subjects allow a type t to have multiple interfaces (sets of fields), and a given instance of t can appear to have any of those interfaces, depending on who is accessing the instance. Similarly, roles allow types to have different interfaces, but unlike subjects, the roles of a given instance are fixed. Thus, roles are weaker than subjects, but sufficient for our descriptive type model.

Fields inherited from roles are named explicitly. If the fields of a role r are $f_1, f_2, \ldots, f_k$, and instance i chooses to inherit from role r, then i's inherited fields are named $r.f_1, r.f_2, \ldots, r.f_k$. For example, suppose type `Application_Program` has the roles `Text_Editor`, `Compiler`, and `Debugger`. An instance of type `Application_Program`, p, chooses to inherit from roles `Text_Editor` and `Debugger`. If type `Text_Editor` has operations `Insert` and `Delete`, and type `Debugger` has attribute `Current_Breakpoint`, then p has fields `Text_Editor.Insert`, `Text_Editor.Delete`, and `Debugger.Current_Breakpoint`. An example of the use of roles is shown in Section 6.1.

Roles are a solution to the problem of allowing participants and framework components to use the fields of multiple types without combinatorial explosion in the number of types. Continuing the example above, a second instance of type `Application_Program`, p', can inherit a different subset of roles than p does and still have the same type as p, even though its fields are different. To accomplish this inheritance without roles, p and p' either would have to be of different types or would inherit fields that they do not need. For example, program p might have to inherit the fields of the `Compiler` type unnecessarily. Note that roles do not subvert normal subtyping rules, since roles can only add fields to a type, not remove them.

The type model supports queries over all instances of a type. Queries can be used for many purposes, including determining the extent of one or more types (i.e., the set of all instances of the type(s)), 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 parts of framework map to our descriptive type model.[6] Using this type model, a router r, for example, is modeled as an instance of type `Router` or one of its

[4]By "inherits," we mean the type is behaviorally a subtype; thus, the subtype has, at minimum, all fields of the parent.

[5]Ossher and Harrison [OH90] use the term "role" differently to mean a collection of operation interfaces. Any type supporting all operations of a role may be used in place of that role. Thus, their roles are higher-level abstractions than types. Our roles are types.

[6]Roles do not appear in the diagram because they are not first class; see type `Framework_Type` in Appendix A for role-related operations.

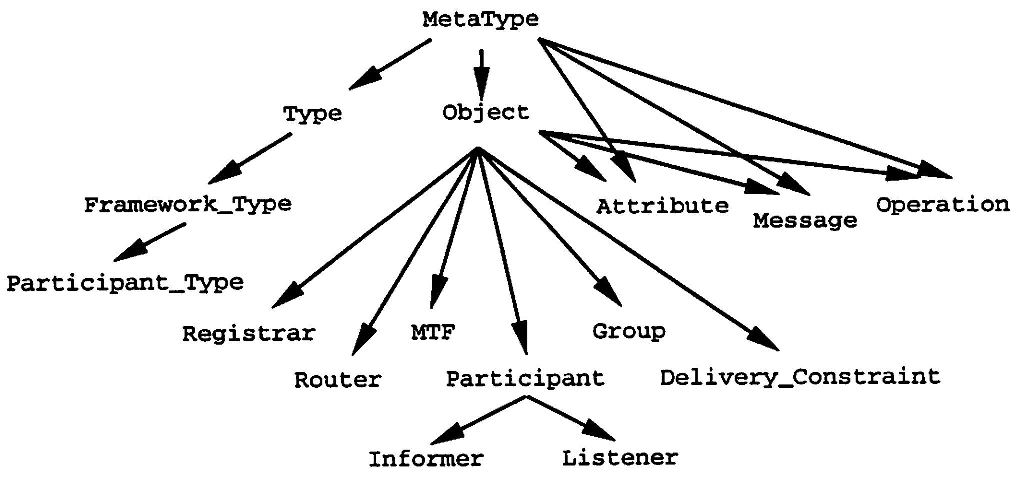

Figure 2: ADT's of the generic integration framework. Arrows point from supertypes to their subtypes.

subtypes. A complete specification of the type model is found in Appendix A.

This type model is primarily a notation for discourse, not an implementation guide. Use of this type model in an implementation would require specification of additional properties, such as an inheritance conflict resolution policy, a definition of type equivalence, and constraints on subtype/supertype relationships.

5 The Generic, Event-Based Integration Framework

This section provides an in-depth description of all parts of the 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`
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`).

5.2 Events and Messages

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

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 are notified of events by some means that is outside of the framework. Messages are the manifestations of events in the framework.

A criticism of event-based approaches in general is that they cannot handle pre-notification: that is, notification about an occurrence before it happens. After-the-fact notification is not sufficient for solving certain kinds of problems, such as consistency maintenance, in which consistency violations must be prevented before they actually occur. Our framework supports the modeling of pre-notification by specifying that "event x is about to occur" is itself an event.

A *message* (type `Message`) is information emitted by an informer in response to an event or events. 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`, which specifies delivery constraints on this message. By default, the value is "no constraints specified."
- `Synchronization`, which represents whether the message can be sent synchronously, asynchronously, or either. By default, the value is "either."
- `Access_Control`, which specifies 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 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 these instance-level attribute values do not conflict with other 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. Both static (prior to participant invocation) and dynamic (during participant execution) registration can be modeled.

Informers must register their intent to send messages, and listeners must register their intent to receive messages. Informers can also 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.

All participant instances can register a variety of additional information. Participants can register message transforming functions and delivery constraints that operate on 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. Registration is also the point at which participant instances select the subset of roles from which they inherit, as described in Section 4. This allows participants to utilize different sets of message types and operations as they are needed.

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[7] 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.

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

5.4 Routers

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

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 and delivery constraints to determine the recipient of a message.

The complete path of a message in an integration mechanism is as follows. Let L be the set of all listeners and M be the set of all messages.

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 multiset $\Sigma = \{(m_1, l_1), (m_2, l_2), \ldots, (m_k, l_k)\}$, 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 MTF's 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 MTF's 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. Σ is a multiset rather than a set because there is no reason for the router to limit the number of times that a particular message is sent to a particular listener.

4. The router delivers each message m_i to listener l_i in an order compatible with the poset Σ.

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 participant'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 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 jobs of the registrar and router are incorporated into a single component. In our 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 products, 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 intended for a particular listener into a sequence of messages intended for particular 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. An MTF is a function

$$f : M \times L \times S \to (M \times L)^* \times S$$

where $(M \times L)^*$ is a sequence (possibly empty) of message/listener pairs.[8]

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. 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. MTF's are equivalent in power to on-line Turing Machines [Hen66].

Most integration products support some message transforming functions: most commonly, filters. **FIELD** [Rei90] listeners register message patterns (filters) to specify which kinds of messages they receive. **ToolTalk** [JR93] allows listeners to register filters that accept or reject messages based on their "class." **Bart** [Bea92] uses "declarative glue" and relational algebra to design filters that select relevant data. Policies in **Forest** [GI90] provide aggregation. A more subtle example of aggregation is that of **Odin** [CO90], 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.

[8]We defined the output to be a sequence of messages, rather than a poset (which would match the router's output), because the responsibility of delivering messages in a particular order is ultimately the router's. The delivery order of the messages output by an MTF can later be modified by delivery constraints if desired.

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 [Koy92] and partially ordered sets [LKA+95, MPS91].

In the 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 products. **ISIS** [Bir93] has "message delivery orderings" that enforce causal relationships between messages. **Darwin** [MR90] has system "laws" that can specify message delivery rules. **Consul** [MPS92, MS92] has an "order protocol" that enforces orderings on received messages. The Amiga **Exec** [Com92] allows listeners to be prioritized for receiving messages. The router in Garlan and Scott's extended Ada [GS93] 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 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, the messages in a group could be considered logically equivalent.

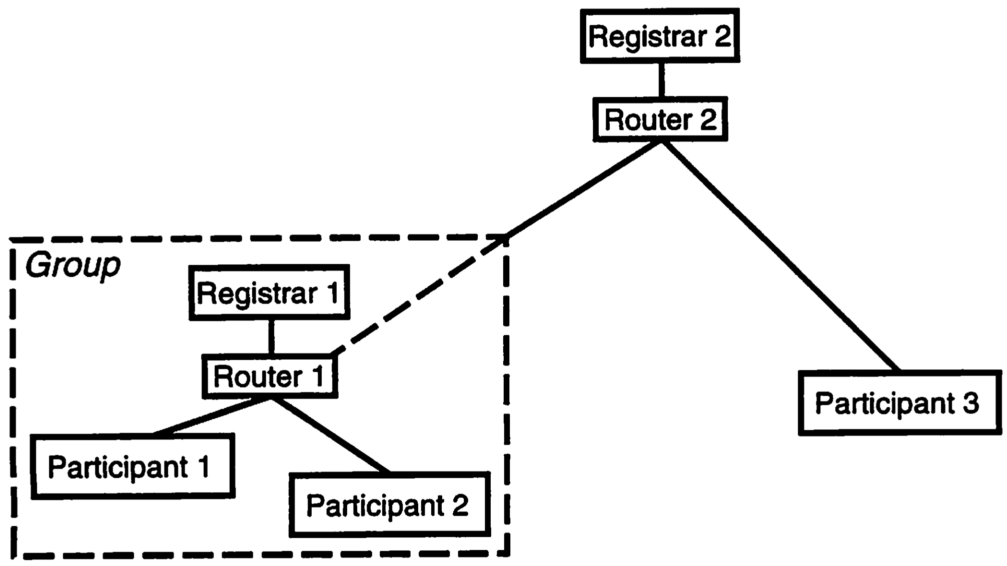

Figure 3: Composition using groups. Registrar 1, Router 1, and Participants 1 and 2 form a group. The group itself is registered as a participant with Registrar 2. Communication to and from the group is accomplished by sending messages between the two routers, using the `Send_To_Router` operation.

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 framework models groups explicitly.

Groups appear in various integration products, though generally only participant groups are supported. **ISIS** [Bir93] provides support for several kinds of participant groups, in which the members have varying degrees of awareness of each other. **FIELD** [Rei90] 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 Event Notifier** [Bha91] 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** [HS92] and **Cronus** [STB86] have multiple servers, each serving a group of participants. **SoftBench** [Ger89] 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* is an instantiation of the generic framework. A *mechanism specification* is a description specifying instantiated properties of a mechanism.

A mechanism specification 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 fields) of framework components, messages, and groups:
  - *Subtypes/restrictions of* `Registrar`. For example, systems that combine the registrar and router into a single component (e.g., [Pur90, Rei90]) can create a subtype that inherits from both `Registrar` and `Router`.
  - *Subtypes/restrictions of* `Router`. For example, **Polylith** [Pur90] uses the polling model of message delivery, and **Q** [MOS90] uses the active model. The **ARexx** [Com91] router is restricted to point-to-point communication, whereas **FIELD** [Rei90] 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** [MPS92] has four subtypes of message ordering constraints available.
  - *Subtypes/restrictions of* `Message`. For example, **FIELD** messages are restricted to strings, but **ACA** [WT90] messages are abstract data objects. **Q** has a `Message` subtype with a `High-Priority` attribute to represent important messages. **ToolTalk** [JR93] messages are restricted to be sent asynchronously only, whereas **Polylith** supports both synchronous and asynchronous communication.
  - *Subtypes/restrictions of* `Group`. For example, **ISIS** [Bir93] 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** [Ger89] has only one router, but **Schooner** [HS92] supports multiple routers.
  - *Particular message transforming functions*. For example, particular **Forest** 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 **FIELD**, the message server could be modeled 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** [DKE+89] uses the **Kerberos** [SNS88] 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.

- Define languages for describing MTF's, delivery constraints, and queries. For example, the MTF language of **FIELD** can describe only filters, **Forest** [GI90] can describe only aggregators (since a policy action can be either a single message or the empty message), and **CORBA** [CCC+93] has 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, defined in Section 5.9.

5.9 Specification of Configurations

A *configuration specification* describes participants and their allowable interactions. A configuration specification 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 and roles. For example, **CORBA** [CCC+93] has an Interface Definition Language for creating participant types.

- *Participant instances*. For example, **Polylith** [Pur90] participants (called "tools") can be defined as instances of types (called "modules").

- *MTF's and delivery constraints* (defined in a mechanism specification) that may be used by participant types or instances. For example, **Forest** [GI90] policies are associated with listeners.

- *Group instances containing participants*. For example, **DEEDS** [LLC+94] allows the specification of particular groups of participants.

- A policy for handling *configuration specification violations*. For example, **CORBA** raises an exception if a listener fails to respond to a message.

An initial configuration specification statically describes the configuration of the initial mechanism specification. Systems such as **Polylith** provide this feature. If an integration approach supports dynamic reconfiguration of its participants, this is modeled simply by changing the configuration specification. For example, a new participant type or instance could be added. Again, changes to the configuration specification may need to be checked for consistency.

6 Case Studies

Many existing integration products can be viewed as instantiations of the generic, event-based integration framework. We present reasonable mechanism specifications for three well-known integration approaches: the implicit invocation product **FIELD** [Rei90], the software bus product **Polylith** [Pur90], and the object-oriented specification of **CORBA** 1.2 [CCC+93].[9] Once the three mechanisms are mapped to the framework, the task of comparing and contrasting them 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. Our framework provides a useful level of abstraction for this comparison. In addition, such comparisons can facilitate 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. As we shall see in our example, all three mechanisms have a readily identifiable router component. One approach to interoperability would be to group the three router types into a single router that can handle messages from all three mechanisms. On the other hand, one of the mechanisms provides point-to-point communication, one provides multicast, and one provides both. The framework calls attention to this important distinction that must be addressed by any plan for interoperability among the three mechanisms.

This section utiliizes the framework types and operations defined in Appendix A.

6.1 FIELD

FIELD [Rei90] 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. Messages can be intercepted by a special tool, called the Policy tool, which takes 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. An example policy is, "exiting the text editor should cause an automatic recompilation of the program that was edited."

In terms of our framework, **FIELD** tools are participants. More specifically, they are subtypes

[9] Several additional features of **CORBA** 2.0 are noted as well.

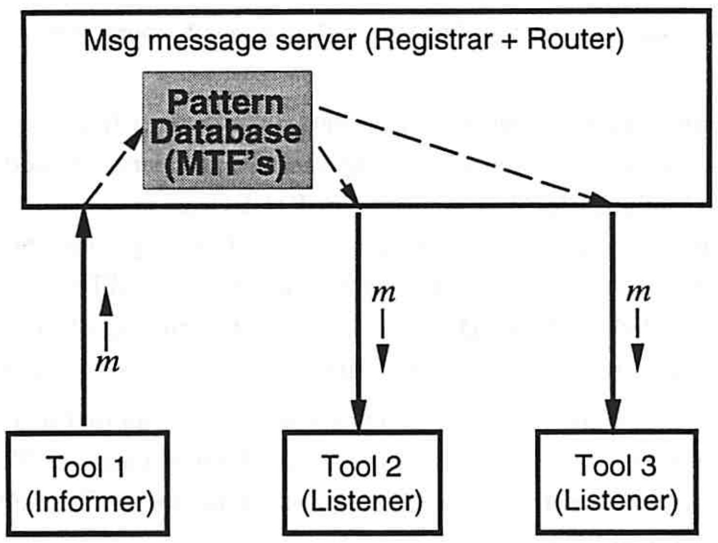

Figure 4: The participants and components of **FIELD**. Tool 1 sends message m to **Msg**. **Msg** routes the message to all tools that have registered a message pattern matching m. The Policy tool is optional and not pictured.

of type `Participant`. **Msg** incorporates the functionality of both a registrar and a router and is therefore a subtype of both `Registrar` and `Router`. Software modules are adapted into participants by wrapping or modifying their source code to make calls to **Msg**'s `Router.Send` operation (**FIELD**'s `MSGsend` family of functions). All tools are automatically registered as informers on startup; that is, the `Registrar` operation `Register_Informer` is invoked automatically. A tool registers as a listener by explicitly invoking the `Register_Listener_Active` operation (**FIELD**'s `MSGcallback` family of functions). Before a tool can receive any messages, however, it must register message patterns, since no patterns are registered by default.

Message patterns and policies are **FIELD**'s two types of MTF's. Message patterns are filters, represented by `printf`-style format strings [KR90]. When a listener registers message patterns, it receives all messages that match those patterns. Message patterns are registered with **Msg** using its `Registrar` operation `Register_MTF` (**FIELD**'s `MSGregister` function).

The Policy tool is a special participant that reads and processes policies. It registers message patterns with **Msg** to receive all messages that are relevant to its policies. A **FIELD**-supplied system delivery constraint causes these messages to be sent synchronously to the Policy tool before any other tool. While **Msg** waits for this synchronous message send to complete, the Policy tool applies any relevant policies and sends any resulting messages to **Msg**. The Policy tool then returns a result to **Msg** that indicates whether the original message should be sent to other tools or discarded.

Since policies are MTF's, and MTF's reside in a router, we model the Policy tool as a grouped registrar and router that itself registers as a participant with **Msg**. Policies are registered with the Policy tool via its registrar's `Register_MTF` operation. Additional, limited delivery constraints can be specified by giving priorities to policies, specifying their order of execution, and registering the priorities with the Policy tool's Register_Delivery_Constraint operation (the LEVEL keyword in **FIELD**'s policy language).

FIELD supports participant grouping so that certain messages from one group member are multicast only to the other group members, using the Group operation Add_Member (**FIELD**'s MSGconnect function). Configuration specifications in **FIELD** are not user definable; any participant can register any message pattern, join any group, etc. The only message type is String, and message strings are converted into sequences of typed values by an MTF. The only configuration specification violation handled by **FIELD** is the sending of a message that does not match any registered message pattern; a reply of NULL is returned to the informer that sent the message.

Recall from Section 4 that roles are optional supertypes; that is, an instance can choose whether or not to inherit from a role. As an example of the use of roles, consider **FIELD**'s Annotation Editor [Rei95], a type of participant used for editing and/or browsing files. **FIELD** supplies four different kinds of annotation editors:

1. **annotedit**, which lets the user edit text, edit annotations (placemarkers in the text for use by other participants), and execute user commands;
2. **aedit**, a simplified **annotedit** with no annotation editing capabilities;
3. **annotddt**, an extended **annotedit** with additional debugger-related operations;
4. **annotview**, a browser without editing capabilities.

FIELD has only one Annotation Editor type, but it has the above four instantiations. To model this structure with traditional multiple inheritance, one could use four types, one for each of the four kinds of editors, as in Figure 5(a), inheriting various capabilities from supertypes (e.g., annotation operations from type Annotator, editor operations from type Editor, etc.). Unfortunately, this approach fails to model the fact that **FIELD** has only one Annotation Editor type. Alternatively, one could use a single Annotation Editor type, Annotation_Editor, and model the four kinds of annotation editors as its subtypes, as in Figure 5(b). This approach, however has the unwanted feature that all four subtypes will inherit operations that they might not need. For example, **aedit** does not have any annotation features, but type aedit wastefully inherits from type Annotator. Using roles, as in Figure 5(c), one can do away with the subtypes of Annotation_Editor by allowing instances to choose the supertypes from which they will inherit. For example, **aedit** would be an instance of Annotation_Editor that inherits fields only from the roles Editor and CommandTool. Note that the use of roles is type-safe. No subtyping rules have been broken; rather, each instance selects a subset of these optional supertypes and inherits only from them.

6.2 Polylith

Polylith has a software bus architecture, as shown in Figure 6. 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

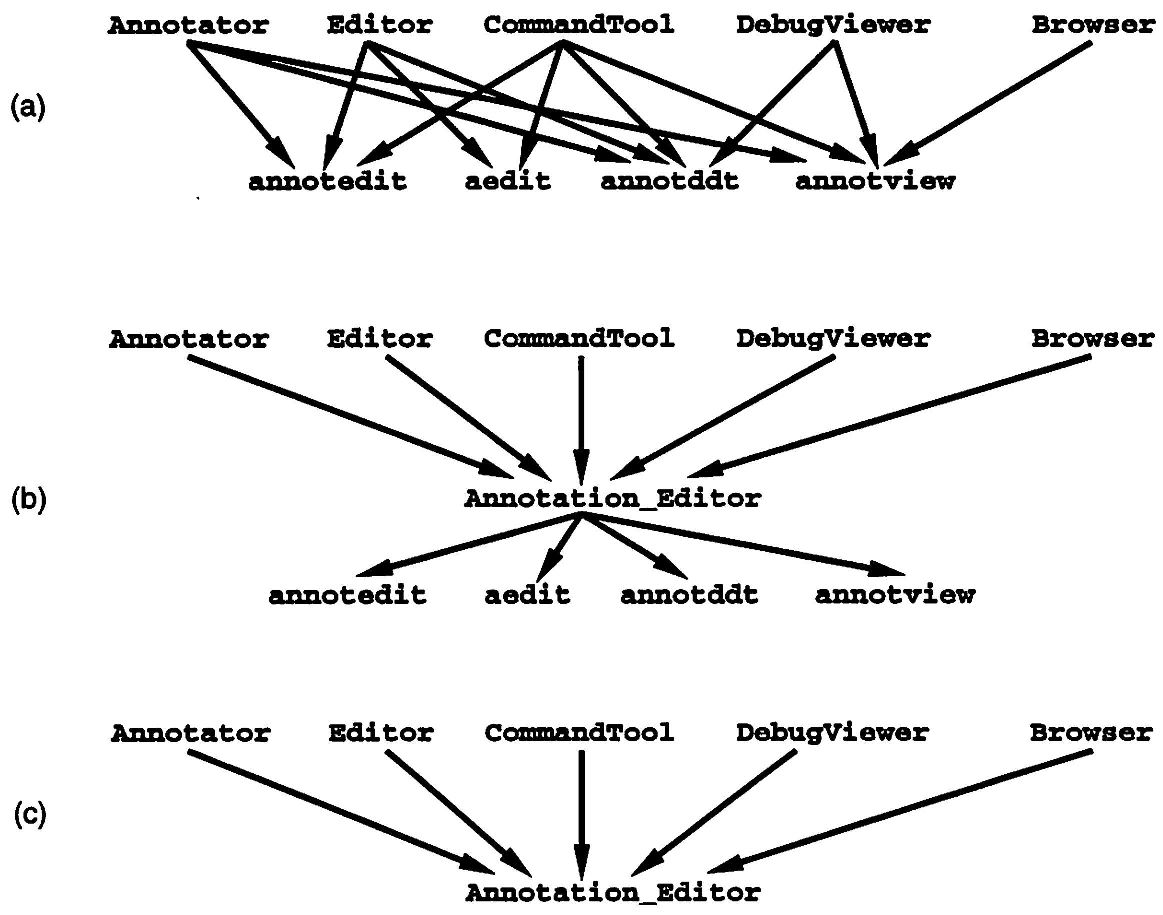

Figure 5: Use of roles (c) versus separate types (a) and subtyping (b). Arrows indicate inheritance from supertype to subtype.

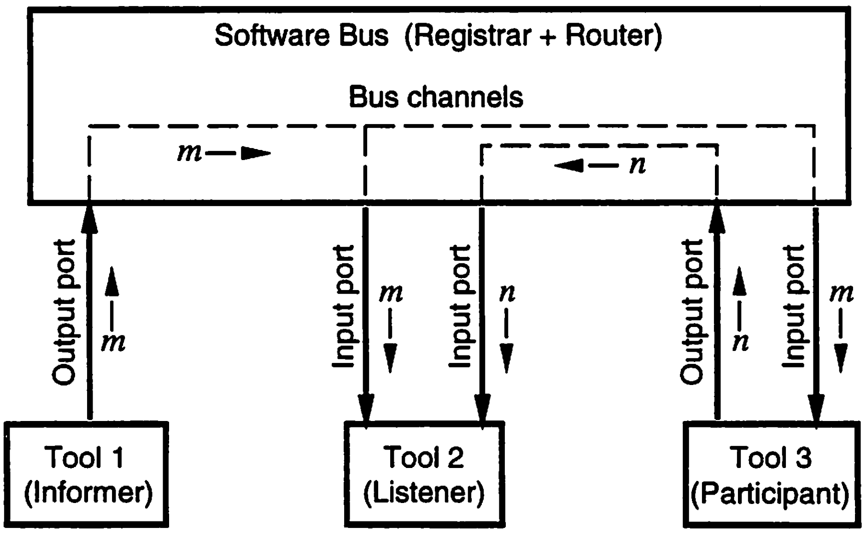

Figure 6: The participants and components of **Polylith**. A message *m* sent by Tool 1 on a given bus channel is received by Tools 2 and 3 that are listening on that bus channel. Similarly, a message *n* is sent from Tool 3 to Tool 2 along a different bus channel.

as tools, and then bind the output ports of tools to the input ports of other tools.[10] Messages may be of simple, structured, and pointer types.

In terms of our framework, **Polylith** tools are participants (subtypes of type `Participant`). The bus incorporates the functionality of both a registrar 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).[11] 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 MTF's 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; the participants connected to a bus channel could be considered an implicit "group," but such groups cannot then be members of other groups. 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.

6.3 CORBA

CORBA, like **FIELD**, is a specification of a client-server architecture for software integration. Unlike **FIELD**, however, all communication is point-to-point, not via multicast. **CORBA**'s specification is much larger and more detailed than that of **FIELD** or **Polylith**, so the description of its mapping to our framework is correspondingly longer.

In the **CORBA** model, shown in Figure 7, programs, called clients, transmit messages, called requests, to uniquely identifiable, encapsulated entities, called objects. Objects respond to requests by executing operations (also called services). 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 that programming language and IDL. IDL supports various basic and structured types, multiple

[10]**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.

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

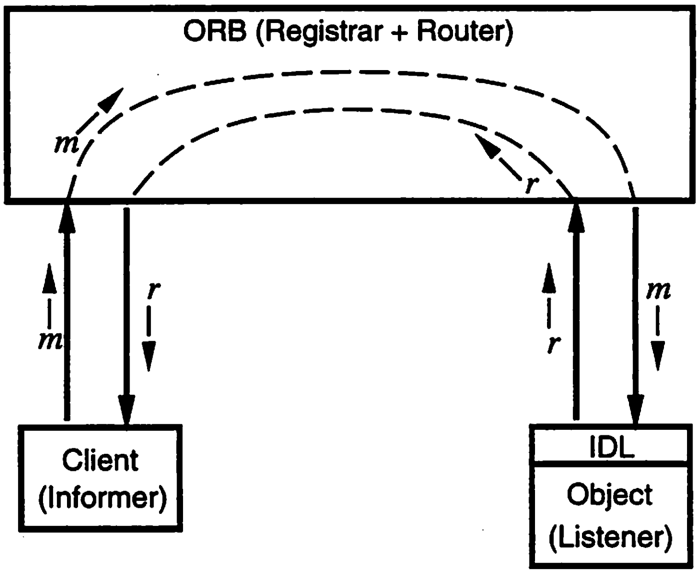

Figure 7: The participants and components of **CORBA**. A client sends a request, m, which is routed to the intended object by an ORB. The object optionally returns a value, r, which is routed back to the client by the ORB.

inheritance, and exceptions.

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.

In terms of our framework,[12] **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" [CCC+93].[13]

[12]**CORBA** does not specify most operation names, leaving them to be determined by each implementation independently, so few mappings to **CORBA** functions are presented.

[13]It is unclear why request contexts are distinguished from operation parameters. This decision probably reflects

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. A software developer either instruments the module's source code to use **CORBA** types and library functions, or writes a second set of stubs to interface the module with the IDL-generated stubs.

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

CORBA provides no MTF's. 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:

1. *At Most Once.* If a request is successful, it is guaranteed to have been received by the intended object exactly once. If a request is unsuccessful, an exception will be raised, and the request is guaranteed not to have been received more than once (i.e., the object did not receive duplicate requests). All `OUT` parameters of an unsuccessful request have undefined values. "At Most Once" requests can be sent synchronously or asynchronously.
2. *Best Effort.* The request is sent asynchronously from client to object with no return value. No guarantee is made that the request will be delivered successfully.

Delivery constraints are registered using the `Register_Informer` operation.

CORBA 1.2 has no explicit support for grouping objects. Implicitly, an ORB can be viewed as a group of the objects that it directly manages. **CORBA** supports no operations, however, that address all of the objects managed by an ORB. Alternatively, all **CORBA** objects of a particular type — termed the extension of the type — can be viewed as a group. Any object in the group can substitute for any other in a type-compatible manner.[14] There appear to be no operations supported by **CORBA** that operate on the extension of a type, however (e.g., multicasting a request to all objects of a given type, or queries over all objects of a given type). **CORBA** 2.0

UNIX biases; i.e., as environment variables are distinguished from command-line arguments.

[14]This substitution may not be transparent, however, since the client may select an object (obtain an object reference) using the object's unique name.

provides explicit support for grouping using domains. A domain is a "set of objects sharing a common characteristic or abiding by common rules" [LCC+95]. A domain may itself be an object and a member of other domains, thereby supporting composition.

Configuration specifications for **CORBA** are written in IDL, and they specify only **CORBA** object (participant) types.[15] 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: MTF's and delivery constraints for participants, instances of participants, or groups.

6.4 Comparison

The features of **FIELD**, **Polylith**, and **CORBA** described in the case studies are summarized in Table 1. The three integration mechanisms have a number of notable similarities, some of which are apparent in Figures 4–7. All three mechanisms have readily identifiable participants that are adapted by some combination of source code modification and wrapping. 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. We characterize the differences by noting major features of the framework that each mechanism does not support. **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 MTF's 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. 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 don't support pointers, and how to handle structured types so **FIELD** can interpret them as strings.

[15]There is only a single Client type.

Feature	FIELD	Polylith	CORBA
Participants	Msg clients (executable programs)	Bus tools (executable programs)	Clients and objects (executable programs)
Adaptation	Wrap or modify source code	Wrap and modify source code	Wrap and modify source code
Registrar	Msg message server	Bus	ORBs
Router	Msg pattern matching engine	Bus	ORBs
MTF's	Filters, policies	Filtering by bus channel	None
Delivery Constraints	Policy priorities	Not user definable	At most once, best effort
Grouping	Participant groups	None	Participant (object) and router (ORB) groups, called domains
Message Types	String	Simple, structured, and pointer types	Simple, structured, and interface types
Message Sending	Multicast	Point-to-point, multicast	Point-to-point
Message Delivery	Non-polling (passive)	Polling (active)	Unspecified
Configuration Spec	Not user definable	MIL specification	IDL specification
Specify instances?	No	Participants	No
Config Violation Handling	NULL return value	None	Predefined exceptions

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

- 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

By defining a generic, event-based integration framework, we have identified common components of event-based integration and provided a means for understanding and discussing event-based integration approaches. The framework provides characterizations and formalizations of participants, messages, registrars, routers, message transforming functions, delivery constraints, and groups. It supports the specification of mechanisms and configurations and can model the dynamic and static behavior of many diverse event-based integration approaches.

The framework is a powerful tool for comparing and contrasting integration mechanisms. We have demonstrated this by modeling three different, popular integration approaches. More such mappings are in progress. By highlighting significant similarities and differences between mechanisms, the framework provides guidance for choosing the most 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 framework is a reference model, not an implementation guide. An implementation would need to elaborate important issues beyond the instantiation of the mechanism, such as an instance model, runtime system semantics, and languages for MTF's, delivery constraints, and queries. In addition, while our ADT model provides a well-understood description of objects and their behavior, it implies that many kinds of objects are first-class. This view may not always be desirable and may need to be addressed directly in an implementation. We plan to investigate this tradeoff further.

We also intend additional exploration of scalability. The 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.

Acknowledgements

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

A The Type Model

This appendix lists our abstract data types and defines the signatures and exceptions of their operations. In Figure 2, the types in the `Object` subtree are instances of the types in the `Type` subtree. Type `Type` contains most of the operations related to manipulating types. Subtype `Framework_Type` adds role-related operations (for `Participant`, `Registrar` and `Router` types), and subtype `Participant_Type` adds message-related operations (for `Participant` types). All operation parameters are "in" parameters unless otherwise stated. Common exceptions such as "permission denied" have been omitted.

1. `MetaType` (supertypes: `none`)

   Type `MetaType` is introduced as the root of the hierarchy to avoid a loop in the hierarchy: types have fields, and fields have types. 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 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. `t` is always 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 type/signature.

  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. Framework_Type (supertypes: Type)

- **Add_Role**(t: IN OUT Framework_Type; super: Type);

  Add type super as a role of the type t.

  Exceptions: super is not a known type, super is already a role of t.

- **Delete_Role**(t: IN OUT Framework_Type; super: Type);

  Remove type super as a role of the type t.

  Exceptions: super is not a role of t.

- **Roles_Of**(t: Framework_Type) return set of Type;

  Return all of t's roles. If t has no roles, then the empty set is returned.

4. Participant_Type (supertypes: Framework_Type)

- **Add_Message**(t: IN OUT Participant_Type; msg: Message);

  Add the given message to the type t. The same message cannot be added twice, and a type cannot have two messages with the same name and the same attributes. A type may have multiple messages with the same name and different attributes; i.e., overloaded messages.

  Exceptions: msg is already a message of t, t already has a message with this name and attributes.

- **Delete_Message**(t: IN OUT Participant_Type; msg: Message);

  Delete the given message from the type t.

  Exceptions: msg is not a message of t.

- **Messages_Of**(t: Participant_Type) return set of Message;

  Return all messages of the given type. If t has no messages, then the empty set is returned.

- **Message_From_Name**(msg_name: Message_Name; msg_type: Type) return Message;

  Return the message of the type t 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.

5. `Object` (supertypes: `MetaType`). 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 fields.
  Exceptions: Mechanism specification violation.
- **Get_Type**(`obj: Object`) return `Type`;
  Get the type of object `obj`.

6. `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.

7. `Operation` (supertypes: `Object, MetaType`)

- **Create**(`n: Operation_Name; t: Operation_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));
```

8. 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));
```

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

9. Participant (supertypes: Object). Type Participant is an instance of type Participant_Type.

- **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.

Subtypes of `Participant`: `Informer` and `Listener`. (No additional operations.)

10. `Registrar` (supertypes: `Object`) Type `Registrar` is an instance of type `Framework_Type`.

- **Register_Informer(**`r: Registrar; i: Informer; roles: set of Type; msgs: set of (Message, Delivery_Constraint, Synchronization, Access_Control)`**)** `return Router`;
  Register an instance of an informer. The informer instance can choose to inherit operations from only a subset of its type's roles by specifying them as parameters. The informer can also choose to support only a subset of its type's messages and can vary those messages' delivery constraint, synchronization, and access control information.
  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; roles: set of Type`**)** `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. Like informers, listeners can choose to inherit from only a subset of their type's roles. If no roles are specified, then by default, the listener inherits from all of its roles.
  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; roles: set of Type; 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. Like informers, listeners can choose to inherit from only a subset of their type's roles.
  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 ith 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 ith, (i+1)st,... MTF's of that participant (if they exist) are automatically incremented by 1.

  The order of evaluation of a participant's MTF's 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 MTF's associated with the given participant. This is necessary so a participant can know the order in which its MTF's 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.

11. Router (supertypes: Object). 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; i: Informer; m: Message`);
  Send the message. Its delivery will proceed in whatever manner has been specified via the caller's registration information.
  Exceptions: Listener not found, delivery constraint violated, message type mismatch.
- **Send_To_Router**(`r1: Router; r2: Router; m: Message`);
  Send a message from router `r1` to router `r2`.
  Exceptions: Router r2 not found, send failed, delivery constraint violation.
  Provisions: Only a router can invoke this operation.

12. `MTF` (supertypes: `Object`)

    - **Transform**(`f: MTF; m: Message; l: Listener; m_new: OUT sequence of Message; l_new: OUT set of Listener`);
      Apply the MTF to `m` and `l`, returning `m_new` and `l_new`. The MTF's state may change as a result.

13. `Delivery_Constraint` (supertypes: `Object`)

    Delivery constraints have no operations of their own. **Router.Send** raises an exception when a delivery constraint is violated.

14. `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.

15. Other types

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

References

[AG94] Robert Allen and David Garlan. Formal connectors. Technical Report CMU-CS-94-115, Carnegie Mellon University, School of Computer Science, 1994.

[App91] Apple Computer, Inc. *Inside Macintosh*, volume VI. Addison-Wesley Publishing Company, Inc, New York, 1991. Chapters 4-8.

[Bar84] J. G. Barnes. *Programming In Ada*, page 153. International Computer Science Series. Addison-Wesley Publishing Company, second edition, 1984.

[Bar93] Daniel J. Barrett. SDL BMS: A simple broadcast message server. Arcadia Document UM-93-03, University of Massachusetts, Software Development Laboratory, Computer Science Department, October 1993.

[Bea92] Brian W. Beach. Connecting software components with declarative glue. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 120–137, Melbourne, Australia, May 1992.

[BGH+93] M. Bever, K. Geihs, L. Heuser, M. Mühlhäuser, and A. Schill. Distributed systems, OSF DCE, and beyond. In A. Schill, editor, *Lecture Notes in Computer Science Volume 731: DCE — The OSF Distributed Computing Environment.* Springer-Verlag, 1993.

[Bha91] Nehru Bhandaru. Pilgrim event notifier. Project Pilgrim working draft, November 1991.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of The ACM*, 36(12):37–53, December 1993.

[CCC+93] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The Common Object Request Broker: Architecture and Specification.* Object Management Group and X/Open, 1993. Revision 1.2.

[CO90] Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

[Com91] Commodore-Amiga Incorporated. *Programmer's Guide to ARexx and Disk*, August 1991. Part number AREXX01.

[Com92] Commodore-Amiga Incorporated. *Amiga ROM Kernel Reference Manual: Libraries.* Addison Wesley Publishing Company, third edition, 1992.

[Dig] Digital Equipment Corporation. DEC FUSE: An open, integrated software development environment. Product Information Sheet EC-F1550-48.

[DKE+89] C. Anthony DellaFera, John T. Kohl, Mark W. Eichin, Robert S. French, David C. Jedlinsky, and William E. Sommerfeld. Zephyr notification service. Technical plan, Project Athena, June 1989.

[FHC+93] Charles Falkenberg, Christine Hofmeister, Chen Chen, Elizabeth White, Joanne Atlee, Paul Hagger, and James Purtilo. *The Polylith Interconnection System: Programming Manual for the Network Bus.* University of Maryland, College Park, 3.0 edition, September 1993. Draft.

[Fro89] Brian Fromme. HP Encapsulator: Bridging the generation gap. SoftBench Technical Note Series SESD-89-26, Hewlett-Packard, November 1989.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In David Wile, editor, *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 175–188, New Orleans, LA, December 1994. ACM Press. Published as SIGSOFT Notes 19(5).

[Ger89] Colin Gerety. HP SoftBench: A new generation of software development tools. SoftBench Technical Note Series SESD-89-25, Revision 1.4, Hewlett-Packard, November 1989.

[GI90] David Garlan and Ehsan Ilias. Low-cost adaptable tool integration policies for integrated environments. In *SIGSOFT Proceedings*, December 1990.

[GLST95] Bob Gautier, Chris Loftus, Edel Sherratt, and Lynda Thomas. Tool integration: Experiences and directions. In *Proceedings of the 17th International Conference on Software Engineering*, pages 315–324, 1995.

[GS93] David Garlan and Curtis Scott. Adding implicit invocation to traditional programming languages. In *Proceedings of the 15th International Conference on Software Engineering*, 1993.

[Hen66] F. C. Hennie. On-line Turing machine computations. *IEEE Transactions on Electronic Computers*, EC-15(1):35–44, February 1966.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, October 1993. Published as *ACM SIGPLAN Notices* volume 28, number 10.

[HS92] Patrick T. Homer and Richard D. Schlichting. A software platform for constructing scientific applications from heterogeneous resources. Technical Report 92-30, University of Arizona, Department of Computer Science, November 1992.

[JR93] Astrid Julienne and Larry Russell. Why you need ToolTalk. *SunEXPERT Magazine*, pages 51–58, March 1993.

[KG87] Gail E. Kaiser and David Garlan. Melding software systems from reusable building blocks. *IEEE Software*, pages 17–24, July 1987.

[Koy92] R. Koymans. *Specifying Message Passing and Time-Critical Systems With Temporal Logic*. Springer-Verlag, 1992.

[KR90] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, second edition, 1990.

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

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 1995. To appear.

[LLC+94] Tin-Peng Liang, Hsiangchu Lai, Nian-Shing Chen, Hungshiung Wei, and Meng Chang Chen. When client/server isn't enough: Coordinating multiple distributed tasks. *Computer*, 27(5):73–79, May 1994.

[LM93] Fred Long and Ed Morris. An overview of PCTE: A basis for a portable common tool environment. Technical Report CMU/SEI-93-TR-01, Carnegie-Mellon University Software Engineering Institute, March 1993.

[MOS90] Mark J. Maybee, Leon J. Osterweil, and Stephen D. Sykes. Q: A multi-lingual interprocess communications system for software environment implementation. Technical Report CU-CS-476-90, University of Colorado, Boulder, June 1990. Submitted to Software Practice and Experience.

[MPS91] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report 91-32, University of Arizona, Department of Computer Science, November 1991.

[MPS92] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Modularity in the design and implementation of Consul. Technical Report 92-20, University of Arizona, Department of Computer Science, August 1992.

[MR90] Naftaly H. Minsky and David Rozenshtein. Configuration management by consensus: An application of law-governed systems. In Richard N. Taylor, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 44–55, Irvine, CA, December 1990. ACM Press. Published as SIGSOFT Notes 15(6).

[MS92] Shivakant Mishra and Richard D. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report 92-19, University of Arizona, Department of Computer Science, August 1992.

[NGGS93] David Notkin, David Garlan, William G. Griswold, and Kevin Sullivan. Adding implicit invocation to languages: Three approaches. In Shojiro Nishio and Akinori Yonezawa, editors, *Object Technologies for Advanced Software: Proceedings of the First JSSST International Symposium*, pages 489–510, Kanazawa, Japan, November 1993. Springer-Verlag.

[OH90] Harold Ossher and William Harrison. Support for change in *rpde*^{3}. In Richard N. Taylor, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 218–228, Irvine, CA, December 1990. ACM Press. Published as SIGSOFT Notes 15(6).

[Pat93] Paul B. Patrick, Sr. CASE integration using ACA services. *Digital Technical Journal*, 5(2):84–99, Spring 1993.

[Per89] Dewayne E. Perry. The Inscape environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 2–12, 1989.

[PH91] James Purtilo and Christine Hofmeister. Dynamic reconfiguration of distributed systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 560–571, 1991.

[Pur90] James M. Purtilo. The Polylith software bus. Technical Report UMIACS-TR-90-65, University of Maryland, May 1990.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the FIELD environment. *IEEE Software*, July 1990.

[Rei95] Steven P. Reiss. FIELD: A friendly integration environment for learning and development. Book manuscript, 1995.

[SN92] Kevin J. Sullivan and David Notkin. Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[SNS88] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Usenix Conference Proceedings*, pages 191–202, Dallas, Texas, February 1988.

[STB86] Richard E. Schantz, Robert H. Thomas, and Girome Bono. The architecture of the Cronus distributed operating systems. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 250–259, Cambridge, MA, May 1986.

[Sun92] SunSoft, Inc. *ToolTalk 1.0.2 Programmer's Guide*. Sun Microsystems, Inc., 1992.

[WT90] Andy Wilson and Bob Travis. Application control architecture specification. Digital confidential and proprietary document, May 1990.