An Event-Based Software Integration Framework

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

CMPSCI Technical Report 94–47
June 1994

Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Defense Advanced Research Projects Agency under grant MDA972-91-J-1009.

An Event-Based Software Integration Framework*

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

June 10, 1994

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, match user requirements with the capabilities of event-based integration products, and so on. We attempt to address these shortcomings by specifying a *generic, event-based integration framework* that provides a flexible model for discussing and comparing event-based integration approaches. This model supports dynamic and static specification, composition and decomposition, and can be instantiated to describe the features of most common event-based integration approaches. To demonstrate this, mappings to several popular products are presented.

Keywords: integration, event-based notification

1 Introduction

A serious concern in the construction of large software systems is *integration*, in which multiple software *modules* (programs, subprograms, collections of subprograms, etc.) are made to cooperate. 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 the 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., [11, 14, 18, 21, 28, 29, 30]).

Unfortunately, no consistent model for describing event-based integration exists; in fact, there is not even a consistent vocabulary for discussing it. As a result, it is hard to capture an application'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 not "yet another" loose integration mechanism; rather, it is a high-level, general, and flexible model with several purposes:

- To identify *common components* found at the heart of event-based software integration and define a precise and consistent *vocabulary* for discussing them.
- To serve as a basis for *comparison* of specific event-based integration mechanisms.
- To facilitate *interoperability* among different integration mechanisms.

*This work was supported by the Defense Advanced Research Projects Agency under Grant MDA972-91-J-1009.

- To provide *insight* into what is required for high-level communication among software modules.

The framework is intended to support event-based software integration, as found in **SoftBench** [18], the **CORBA** specification [11], and many other approaches. It is not intended to support 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** [23]), and stream-based (e.g., pipes), though it is capable of supporting each of these in limited ways. It is unrealistic to expect a single model to subsume all possible integration models. As we demonstrate, however, the framework does support many common and useful integration models.

We begin in Section 2 by describing related work in event-based software integration. In Section 3, we give a brief overview of the framework, followed by a detailed description. In Section 4, we demonstrate the framework's effectiveness by mapping representative examples of several integration mechanisms onto it. Finally, we draw conclusions about the framework and its potential.

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, dynamic versus static behavior, and naming issues.

Methods Of Communication: Two primary methods of communication among software modules are *point-to-point* and *multicast*.[1] Point-to-point means that data is sent directly from one software module to another. Two common point-to-point approaches are *procedure call*[2] and *application-to-application*. A procedure call sends procedure parameters from a software module known as the *caller* to another known as the *callee*, optionally returning a value to the caller. In the application-to-application approach, application programs have unique ID's and transmit messages to one another via these ID's. This approach is common on personal computers, since the simplifying assumptions of "one user per machine" or "one invocation of a given program at a time" may often be made. Examples include **Apple Events** [2] on the Macintosh and **ARexx** [9] on the Commodore Amiga.

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 kinds of data that are then routed, usually by a central server, to the appropriate recipients. Implicit invocation, originally called *selective broadcast*, was pioneered by **Field** [30]. Today, implicit invocation is used in many commercial products such as Hewlett-Packard's **SoftBench** [18], Sun's **ToolTalk** [21] and DEC's **FUSE** [14]. It has also been added to some programming languages (e.g., [27]). 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 may be rearranged without modifying the tools. An example of a software bus is **Polylith** [15, 29].

Another primary method of communication is *shared repository*, in which modules communicate by storing and retrieving data to and from a central database. While this is not a loose integration method, one particular kind of shared repository, the *sequential shared repository*, is sometimes combined with an event-based approach to provide loose integration. A sequential shared repository is a database that disallows concurrent access to its contents. *Blackboard systems* [20] are a popular integration mechanism using this approach.

As we demonstrate in Section 4, our framework supports the point-to-point, multicast, and sequential shared repository approaches.

Specification Of Module Interactions: Various integration approaches allow interactions between software modules to be specified at different levels of abstraction, ranging from low-level to high-level. RPC stub generation is an example of a mechanism that allows simple, low-level connections between caller and callee to be specified. **ToolTalk** [21] wraps modules in a layer of software with input and output interface specifications. **Polylith** [29] takes this one step further and allows a somewhat higher-level specification of the

[1] *Broadcast* is a special case of multicast in which data is sent to all possible recipients, so we do not discuss it separately.

[2] Procedure call is not properly an event-based mechanism, but it is often used to communicate events between modules in event-based mechanisms.

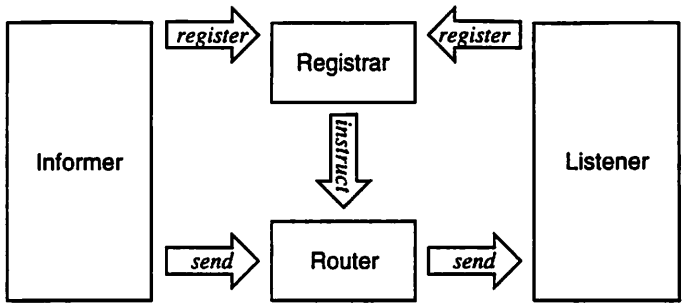

Figure 1: Relationships among informer, listener, registrar, and router.

connections between wrapped modules' interfaces. **SoftBench** [18] provides a programming language rich enough to specify the interactions of wrapped modules without any modifications to the original source code. Finally, **Meld** [22] permits a high-level specification of inter-module interactions via constraints rather than imperative statements. A related approach is to attach formal semantics to modules [13] or the connections between modules [1] and reason about the correctness of module interactions. As shown in Section 3.2.11, our framework supports specification of module interactions at many levels of abstraction.

Dynamic vs. Static Behavior: Different integration approaches permit varying levels of static and dynamic specification of their behavior. For example, **Polylith** version 2.1 [29] requires that modules be terminated and restarted to change their connections. The **SDL BMS** [3] allows modules to register and unregister for service dynamically. **ToolTalk** [33] supports both static and dynamic specification of interactions. As shown in Sections3.2.10 and 3.2.11, our framework supports both dynamic and static specification.

Naming Issues: Interacting software modules may be named in various ways. At the most basic level, software modules are contacted directly by name, as in **ARexx** [9]. **CORBA**'s [11] object-oriented approach and **DCE**'s name server support dynamic location of modules by name. **ISIS** [7] allows multiple modules to be *grouped* under a single name so they may be accessed together. Dynamic location and grouping are both examples of the use of *aliases*. Aliases support loose integration since a module may be exchanged for another without changing the name that other modules use to refer to it. Aliases may transparently represent individual modules, groups of modules, or distributed modules on multiple physical machines. As shown in Sections 3.2.7 and 3.2.8, our framework supports several naming models via attributes and groups.

3 The Generic, Event-Based Integration Framework

3.1 Overview of the Framework

Components of the Framework: 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*.

Before a participant may transmit or receive any messages, it must *register* its intent to do so via a *registrar*. Participants may register (and unregister) information such as simple intent to communicate, synchronization preferences, events in which they are interested, mappings of events to messages, and more (as will be described in Section 3.2). The registrar then contacts a component called a *router* to create communications channels between registered informers and listeners and transmit messages to their intended recipients.

The framework allows the content and/or routing of messages to be modified after the messages are sent but before they are received. A *message transforming function* (MTF) may dynamically convert one sequence of messages intended for a set of listeners into another sequence intended for the same or another set of listeners. A common, simple example of an MTF is a *filter*, which selectively removes messages from a listener's incoming message stream if they do not match certain criteria. Another is an *aggregator*, which transmits a new message whenever a particular sequence of incoming messages is detected.

In addition to altering messages, one may also control the rules of message delivery by imposing *delivery constraints*. These are constraints on the delivery of messages: order of delivery, timing of delivery, and so on. For example, one may specify that all messages must be received in exactly the order they are sent, or that a particular listener must receive all messages before any other listener does.

The framework supports the description of *groups* and *attributes* which are useful for capturing composition and decomposition capabilities. Groups are sets of participants and/or framework components that may be logically treated as individual participants or framework components. For example, a registrar, router, and set of participants may be treated as a single participant, facilitating the description of a hierarchical, event-based model. Attributes are named, typed values that may be associated with participants, framework components, messages, and events. Attributes provide a flexible approach for defining selection capabilities.

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 participants that are interacting from those that are not. Routing transmits data among participants. Message transforming functions allow the data to change dynamically. Finally, delivery constraints permit control over the flow of information.

Using The Framework: The generic framework does not specify details such as:

- Particular message transforming functions and delivery constraints, the number of registrars and routers, and other details.
- How to implement framework components, messages, and events.
- How to turn software modules into participants.
- Particular interactions between participants.

Instead, using a three-phase process, the framework may have these details imposed on it dynamically or statically and thereby model software integration mechanisms with specified properties and participant interactions. The phases are called *instantiation*, *adaptation*, and *configuration*.

The *instantiation* phase is the process of imposing additional details and semantics on the framework and its components, as listed in Section 3.2.10. We call such an instantiated framework a *mechanism*, and the user responsible for this phase is called a *mechanism engineer*. The description of a mechanism is called a *mechanism specification*. Mechanism specifications may be defined statically or applied dynamically to a mechanism. The actual implementations of framework components, messages, and events are not explored in this paper. The separation of instantiation from implementation is important because it allows us to discuss the differences between mechanisms that affect their behavior versus those that do not. As we shall see in Section 4, the behaviors of many software integration products may be described by mechanisms instantiated from the framework.

The *adaptation* phase is the process of describing how software modules are turned into participants that interact via a particular mechanism. Adaptation may or may not require modifications to the modules. Adapted participants are said to be *compatible* with the mechanism.

The *configuration* phase is the process of defining interactions between participants in a particular mechanism. That is, we may specify which participants are registered, how they interact with one another, which message transforming functions they use, how participants adhere to the delivery constraints, and so on. Such a description is called a *configuration specification*, and it defines a *configuration* of the mechanism. Configuration specifications may be defined statically or applied dynamically to a mechanism. The *configuration state* is the set of states of all participants in the configuration at a moment in time. The user responsible for the adaptation and configuration phases is called a *configuration engineer*.

3.2 Detailed Description

3.2.1 Participants, Informers, and Listeners

A *participant* is a software module that has been adapted to be compatible with an integration mechanism. An *informer* is a participant that transmits messages, and a *listener* is a participant that receives messages. A participant may be an informer, a listener, or both. Each listener has one or more incoming *message ports* at which incoming messages are held until the listener handles them. The semantics of each message port are definable; for example, one port may queue messages, and another may emit an event on message receipt.

Every participant has a unique name or identifier that distinguishes it from all other participants. We call this the *participant ID*. Since a group of participants can be treated logically as a single participant (see Section 3.2.8), this implies that a group of participants has its own ID. Also associated with every participant is a *participant state*. This may contain any information about the participant including, but not limited to, state information about the adapted software module, and it allows the participant to respond to a mechanism-specific set of queries. For example, it may include a description of the messages currently waiting at the participant's message ports but not yet processed.

Before a module can be used within a given integration mechanism, it must be turned into a participant that is compatible with the mechanism. This process is called *adaptation*. Adaptation may be performed either as the module is being written or after it has been written. Adaptation may involve modifying the module's source code manually or automatically, linking with custom libraries, wrapping (sometimes called *encapsulating*) the module in another layer of software, or other means.

Formally, let I be the set of all integration mechanisms, P be the set of participants, and X be the set of all software modules. Given a module $x \in X$ and an integration mechanism $i \in I$, an *adaptation* $a(x, i)$ is the relation $a : (X \times I) \rightarrow P$. That is, $a(x, i)$ is a participant compatible with an integration mechanism. a is a relation, not a function, because there may be many methods of adapting modules for a given mechanism.

3.2.2 Events and Messages

An *event* is a notable occurrence such as the invocation of a program, modification of a file, change of a participant's state, or sending of a message. Each event has a name and a type.

A *message* is information emitted by an informer in response to an event or events.[3] Each message has a name and zero or more *message parameters*, analogous to the name and parameters of a subprogram. Messages and message parameters are typed. A message's parameter types and number of parameters are collectively called the *message signature*.

3.2.3 Registrars

Before a participant may transmit or receive any messages, its intent to do so must be *registered* via a *registrar*. Conceptually, a registrar is a front-end for the creation of router connections. Information about a participant can be registered by that participant or by other means, such as by another participant or by a configuration engineer. Registration may be done dynamically or statically.

An informer registers its intent to send messages, either in general (e.g., broadcast) or with particular listeners. It also registers associations between events and messages; that is, on the occurrence of a particular event, which messages the informer will emit. A listener registers only simple intent to receive messages. Finer-grained control over the selection of messages is achieved by allowing participants to associate message transforming functions with their incoming and outgoing message streams. For example, a listener interested only in messages from a particular informer or pertaining to a particular event can use an MTF that filters out all incoming messages not matching those criteria.

Particular delivery constraints, discussed in Section 3.2.6, may be registered. Delivery constraints may apply per participant, per message, or both; for example, "All messages of type t with attribute a, sent by Informer i_3 in response to event e, should be received first by Listener l_9." Synchronization and concurrency control preferences, discussed in Section 3.2.9, may also be registered and are described using the same levels of granularity.

When a registration is requested, some consistency checking may be necessary to insure that the registration does not conflict with any other registrations. For example, if two different listeners both attempt to register a delivery constraint to receive all PRINT messages before any other listener, both requests cannot be satisfied simultaneously.

Associated with each registrar is a *registrar state*. The state contains all registered information and additional information as specified by a mechanism engineer. Registrars may respond to queries about their state, such as which participants are registered, what are the attributes of participant p, which participants are interested in message m, and so on. Not all queries may be answerable directly from the registrar's state, so the registrar may need to obtain router state information as well.

[3] Note that "I am about to send a message" and "a message has been sent" are both events. Therefore, messages may effectively be sent on demand or in response to other messages.

3.2.4 Routers

A *router's* purpose is to receive messages from informers and make them available to listeners. Routers create communication channels between participants in response to requests from registrars. 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. Routers are distinguished from registrars in that they are time-critical components; that is, one of their major tasks is the optimization of message transmission between participants.

Routers may make a message available to listeners by two methods. The first is to send the message immediately to the intended recipients. The second is to hold the message until some event occurs, such as a request from a listener or the passage of a certain amount of time. The first method corresponds to the traditional view of message passing, and the second method gives routers very limited repository semantics, as demonstrated in Section 4.4.

Communications channels may exist only long enough to transmit the message, or they may remain in existence until the participants using it indicate that they no longer need it (i.e., they unregister). Such "long-lasting" channels are useful for several reasons. First, they may be used to optimize point-to-point communication, since the router can create the channel and then not have to be involved in the sending of every message along the channel. Second, they may be used to model some aspects of *stream-based* integration mechanisms, such as pipes, in which an unspecified amount of data is transmitted in between "begin" and "end" events.

Let L be the set of all listeners and M be the set of all messages. The complete path of a message in the framework is:

1. An informer emits a message $m \in M$.
2. The message m is received by a router.
3. The router applies all applicable message transforming functions, delivery constraints, and synchronization preferences to message m and the set of registered listeners, resulting in a sequence $((m_1, l_1), (m_2, l_2), \ldots, (m_k, l_k))$, 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 $\{m_1, m_2, \ldots, m_k\}$.
4. The router makes each message m_i available to listener l_i.

Associated with each router is a *router state*. The state includes a description of all connections to and from participants, a description of all undelivered messages that have been emitted by informers but have not yet been sent to the message ports of intended listeners, and other information as specified by a mechanism engineer.

3.2.5 Message Transforming Functions

Given a sequence of messages intended for a particular set of listeners, an *message transforming function* (MTF) transforms it into another sequence of messages intended for another set of listeners. An MTF may also have an internal state that changes, thereby changing the system state. Formally, let M be the set of all messages, L be the set of all listeners, and S be the set of all system states. An MTF is a function

$$f : M^* \times 2^L \times S \rightarrow M^* \times 2^L \times S$$

where M^* is a sequence of messages in M, and 2^L is the power set of L.

For the purposes of transforming messages into other messages and/or re-routing them to other listeners, M^* and 2^L are in the domain and range of an MTF so they can be changed by the MTF. We use a sequence of messages, rather than a single message or a set of messages, so the MTF can use the ordering of the messages to affect its computation. In addition, including the system state in the domain and range increases the power of an MTF by allowing it to maintain an internal state and modify its behavior based on that state. Since the system state includes the states of all the framework components, including MTF's, changing an MTF's internal state changes the system state. Of course, a simple MTF function could ignore this state information.

Most integration products support some message transforming functions; most commonly, filters. **Field** [30] participants have regular expressions (filters) that specify which kinds of messages they receive. **ToolTalk** [21] and the Amiga **Exec** [10] allow participants to filter messages based on message "classes." The **Pilgrim**

Event Notifier [6] provides "subscription lists," and participants receive all messages sent to those lists. **Bart** [4] uses "declarative glue" and relational algebra to allow participants to select relevant data. Policies in **Forest** [16] provide aggregation. A more subtle example of aggregation is that of **Odin** [8], 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.

3.2.6 Delivery Constraints

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).

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 may 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, "Listener l_7 should receive all messages before any other listener does."

Some examples of useful delivery constraints are:

- All messages are received strictly in the order they are sent.
- Messages must be received within a certain amount of time.
- Only one (or a subset) of a designated group of listeners need receive a particular message. (Perhaps all of these listeners are capable of responding to it, but only one response is needed.)

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

Delivery constraints are found in some integration products. **ISIS** [7] has "message delivery orderings" that enforce causal relationships between messages. The Amiga **Exec** [10] allows listeners to be prioritized for receiving messages. **Consul** [26] has an "order protocol" that enforces orderings on received messages. Listeners in Garlan and Scott's Ada extensions [17] receive messages in the order in which the messages are defined statically in the source code. Real-time systems have hard and soft deadlines. **ISIS** [7] and **Consul** [25] support atomic broadcast of messages.

3.2.7 Attributes

Participants, framework components, messages, and events may have *attributes*, which are named, typed values. An attribute is referred to by an *attribute name*, and its value is called an *attribute value*. Defining an attribute has three parts:

- Specify a name and a type for the attribute.
- Define a function that, given an attribute and an attribute value, assigns the value to the attribute.
- Define a function that, given an attribute, returns the attribute's value.

The framework does not prescribe any particular use of attributes, but they are provided so that participants and framework components may be identified as having certain properties or capabilities. For example:

- Some useful message attributes are a `participant ID` attribute, indicating the ID of the participant that sent the message; a `timestamp` attribute, indicating the time at which the message was sent; and an `event` attribute, indicating the event that triggered the message.
- A listener may have `operation` attributes indicating that the listener is capable of performing particular operations. For example, a listener that can print Postscript files might have an `operation` attribute with the value `print-postscript`. Informers can query the registrar to learn which listeners provide operations that they desire. Listeners can register to receive messages having a particular `operation` attribute. Thus, attributes can be used to support participant location (directory) services for implicit invocation mechanisms.

- A group of participants, messages, etc., can share a `classification` attribute that describes something that the group's members have in common.
- A `priority` attribute may indicate the relative importance of a participant, event, message, etc.

3.2.8 Groups

Items in the framework, such as participants and framework components, may be *grouped* recursively to form a hierarchy of items. 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. Groups have many uses:

- They may be used to support *composition*. A set of registrars, routers, and participants may be grouped and treated logically as a single participant, which may then be a member of other groups.
- A group of participants may share a set of message attributes.
- A group of events may be considered logically equivalent so that a participant responds the same way to any of them.

Formally, let G be the set of all groups. A group of items is composed recursively of items and groups of items. Given a group $g \in G$, the items and groups that g comprises are called the *members* of g. Groups may be empty. A group may not contain itself directly or indirectly.

It is possible to model groups indirectly using attributes; that is, items that share an attribute can 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. Groups appear in various integration products, though generally only participant groups are supported. **ISIS** [7] provides support for several kinds of participant groups, in which the members have varying degrees of awareness of each other. **Field** [30] supports participant grouping implicitly via message patterns: all participants registering a given message pattern form a group, and messages matching the pattern are multicast to the group members. **Schooner** [19] and **Cronus** [31] have multiple servers, each serving a group of participants. **CORBA** [11] objects and **SoftBench** [18] tools may encapsulate multiple participants, implicitly forming a group.

3.2.9 Other Issues

Other issues associated with the framework are synchronization, concurrency control, and access control. While the enforcement of these issues is often the responsibility of lower-level subsystems (e.g., the operating system or the hardware), a mechanism specification may describe them and define policies for their enforcement.

Messages may be sent via the router from informers to listeners either synchronously or asynchronously. Which method is used depends on the *synchronization preferences* registered for the informer and listener involved. A participant's synchronization preferences may vary based on the messages' content, type, or attributes, or the participant's state, or they may apply to a participant as a whole. Among integration products, a variety of synchronization methods are supported. For example, **ToolTalk** [33] permits asynchronous message-passing only, Garlan and Scott's Ada extensions [17] permit synchronous only, and **Polylith** [15] permits both.

Concurrency control preferences may also be registered for participants. A participant may register a *compatibility matrix* that indicates which messages the participant can accept concurrently. For example, a participant that handles a printer queue might accept only one `Print-File` message at a time, but allow concurrent `Printer-Status` messages querying the status of the queue. Concurrency control may impact other aspects of the mechanism, such as synchronization.

In an integration mechanism, one may wish to permit or restrict access to particular messages, participant attributes, message transformation functions, shared data objects, and so on. Some integration products explicitly support access control. The **Pilgrim Event Notifier** [6] supports the access control provided by **DCE** [5]. **Zephyr** [12] uses **Kerberos** [32] for authentification and access control. The vast majority of integration mechanisms, however, do not explicitly support access control.

3.2.10 Specification of Mechanisms

Having defined the framework components, we now turn our attention to the framework itself. An *integration mechanism* is an instantiation of the generic framework. When it is unambiguous to do so, we use the

simpler term *mechanism*. A *mechanism specification* is a description specifying instantiated properties of a mechanism. After examination of many integration products, we have developed the following list of factors that seem to be important in differentiating integration mechanisms. We would not claim that the list is comprehensive, since different applications may have different integration requirements, but we believe that it represents the core of a mechanism specification.

- Participant specification:
  - *Participant state*. For example, a **CORBA** [11] object has a states that may affect its behavior.
  - *Listener message port semantics*. For example, **SDL BMS** listeners access pending messages as a queue, and **Apple Events** [2] listeners have random access to pending messages.
  - An *adaptation method* for converting modules into participants compatible with the mechanism. For example, **SoftBench** comes with a product called `Encapsulator` for this purpose. Many integration products do not provide such tools, and some do not specify the adaptation process explicitly.
- Event and message specification:
  - *Message, message parameter, and event types*. For example, **Field** [30] messages are strings, but **ACA** [34] messages are abstract data objects.
  - *Message passing capabilities*. For example, **Field** uses multicast, but **ARexx** [9] uses point-to-point communication.
- Registrar and router specification:
  - *Registrar and router states*. For example, a router state could include a history of all messages that the router has handled.
  - *Registration methods*. For example, **Polylith** [29] supports static registration only, **SDL BMS** [3] dynamic registration only, and **ToolTalk** [33] both.
  - *Support for multiple registrars and routers*. For example, **SoftBench** [18] has only one of each, but **Schooner** [19] has multiple servers/routers.
- Message transforming function specification:
  - *Particular message transforming functions*. For example, **Field** supports message filtering and **Forest** supports message aggregation.
- Delivery constraint specification:
  - *Particular delivery constraints*. For example, **Consul** [25] allows messages to arrive in a different order than they were transmitted, but Garlan and Scott's implicit invocation for Ada [17] makes this impossible.
  - *Policies for handling delivery constraint inconsistencies*. For example, if a participant attempts to register a constraint that is inconsistent with another that was previously registered, the registrar could reject the new constraint or override the old constraint.
  - *Policies for handling delivery constraint violations*. For example, real-time systems specify that the violation of a time constraint indicates complete or partial failure of the system.
- Other specification:
  - *Policies for forming groups*. For example, **ISIS** [7] permits the definition of groups with members that may or may not be aware of one another.
  - *Synchronization capabilities*, including policies for resolving mismatches. For example, the **SDL BMS** supports asynchronous communication only, and **Polylith** supports both synchronous and asynchronous communication.
  - *Policies for concurrency control*. For example, blackboard systems [20] do not permit concurrent access to the blackboard.

- *Policies for access control*, including policies for handling violations. For example, **Zephyr** [12] uses the **Kerberos** [32] authentification system for access control.

In addition, attributes and groups may be applied to events, messages, and framework components and used to achieve desired semantics and flexibility in each of the above areas. For example, important messages may be given a `high-priority` attribute, as in **Q** [24], and a registrar and router may be grouped to appear as a single entity, as in **Field**'s `Msg` message server.

An *initial mechanism specification* statically describes the mechanism upon instantiation. As the mechanism evolves over time, other mechanism specifications may be applied dynamically to change the mechanism. For example, one might add new message transforming functions or change the number of routers.

The *mechanism state* is a snapshot of the mechanism at a moment in time. It includes the current mechanism specification information and the states of all registrars, routers, and message transforming functions in the mechanism.

3.2.11 Specification of Configurations

The *configuration* of a mechanism is the set of participants interacting in that mechanism. A *configuration specification* describes participants and their interactions. In particular, it specifies which participants are registered/unregistered, any information to be registered by participants (as described in Section 3.2.3), and information about interactions between participants. Finally, attributes and groups provide a flexible means for specifying additional participant semantics.

An *initial configuration specification* statically describes the configuration of the initial mechanism specification. Systems such as **Polylith** [29] provide this feature. As the mechanism evolves over time, other configuration specifications may be applied dynamically to change the interactions of the participants. For example, when a participant unregisters, the configuration specification changes. Systems such as the **SDL BMS** [3] and **ToolTalk** [33] support such dynamic reconfiguration.

A *configuration state* is a snapshot of a configuration at a moment in time. It includes the current configuration specification information and the states of all participants in the configuration. The configuration state and mechanism state are known collectively as the *system state*.

4 Mappings

Many existing integration products may be mapped to the generic, event-based integration framework. This section describes possible mappings of the underlying mechanisms of **Field** [30], **Polylith** [29], the **CORBA** [11] specification, and the classical blackboard system [20].

4.1 Implicit Invocation: Field and Forest

Field [30] is based on a client-server architecture. Client programs, called tools, broadcast messages among themselves via a central message server called `Msg`. All messages are ASCII strings. Using *message patterns* similar to regular expressions, clients specify which classes of messages they want to receive. `Msg` delivers to each client only the messages that match the client's message patterns.

Forest [16] is an extension to **Field** that allows one to specify how tools should respond to certain messages, without having to modify the source code of the tools. These specifications, called *policies*, are external to the tools. An example policy is, "exiting the text editor should cause an automatic recompilation of the program that was edited."

Field and **Forest** have the following integration requirements. Tools need to broadcast messages via a central message server, register their message patterns with the server, and receive only those messages that match their patterns. Policies need to examine messages in transit and send other messages in response. Finally, policies may have priorities that determine the order in which they are applied.

We now define a mapping for **Field**. Tools are mapped to participants. `Msg` is mapped to a group containing a registrar and router. As a registrar, `Msg` handles message pattern registration. As a router, `Msg` performs broadcasting. All of the messages in **Field** are of a single type `Message` with the message string as its sole parameter. Message patterns are mapped to message transforming functions which examine the content of the messages and determine if the message should be delivered to the participant (filtering). **Forest** policies are mapped to message transforming functions that perform aggregation. Messages sent as

a result of policies are tagged with a `policy` attribute, and delivery constraints are used to enforce policy priorities.

4.2 Software Bus: Polylith[4]

In **Polylith**, individual programs, called tools, connect *input and output ports* to a *software bus* and send and receive messages on named *bus channels*. **Polylith** is not a bus in the traditional sense, since channels connect tools' output ports directly to other tools' input ports. A *module interconnection language* or *MIL* specifies tools and their connections.

Polylith has the following integration requirements. Tools need to connect to the bus and send and receive messages on named bus channels. MIL specifications define each tool's interface, attributes (called "algebras"), synchronization preferences, bus channels used, and connections of those channels to other tools' channels. These specifications are modifiable without modifying the tools involved, but they are static, so tools must be terminated and restarted in order for changes to take effect.

We now define a mapping for **Polylith**. Tools are mapped to participants. MIL is a language for static configuration specifications defining tool input, output, attributes, synchronization preferences, and connections to bus channels and other tools. The software bus is mapped to a group containing a registrar and router. As a registrar, the bus permits tools to register their MIL specifications. As a router, the bus creates the specified connections and transmits messages along them. Each set of tools that share an input or output channel is mapped to a participant group, and the bus channel name is mapped to the ID of that group. Sending a message on a channel is mapped to multicasting that message to the corresponding group of participants.

4.3 CORBA

The Common Object Request Broker Architecture, or **CORBA** [11], is a specification of a software integration mechanism with an object-oriented architecture. In this architecture, programs, called *clients*, transmit messages, called *requests*, to uniquely identifiable, encapsulated entities, called *objects*. In response to these requests, the objects provide *services* to the clients. Objects have *interfaces* that describe their services to the outside world. Interfaces are written in *interface definition language* or *IDL*. Connecting these clients and objects is a server called the *object request broker* (ORB) that provides transparent object location and delivery of requests.

CORBA is a very large and complex specification with more integration requirements than can be addressed here, so we focus on clients, objects, ORB's, and IDL. Clients and objects map to participants. Object ID maps to participant ID and object state to participant state. Object services map to participant attributes; for example, an object providing a printing service has a `Print` attribute. Requests map to messages. An ORB maps to a grouped registrar and router. An object registers with an ORB via the ORB's registrar, permitting the ORB to perform object location and deliver requests to that object. Finally, IDL is used to describe the interfaces that objects provide to clients, providing capabilities similar to those of configuration specifications.

4.4 Blackboard Systems

A classical blackboard system consists of a global data repository called a *blackboard*, one or more software modules called *knowledge sources* (KS's) that access the blackboard, and a centralized control mechanism or *scheduler*. Whenever the blackboard is modified by a KS, an event is broadcast to notify all KS's of the change. Based on that event, some KS's may become interested in accessing the blackboard. An *activation record* (AR) is created for each interested KS and stored in a second global repository called the *agenda*. Based on some criteria, the scheduler chooses one AR from the agenda, permits the KS associated with the AR to access the blackboard, and the cycle repeats. Note that there is no concurrent access to the blackboard.

The classical blackboard system has the following integration requirements. KS's need to specify their interest in certain events. AR's must be placed onto the agenda, and the scheduler must select one. The KS's must read from and write to the blackboard. Finally, the scheduler must ensure that at most one KS accesses the blackboard at a time.

[4]This mapping is for **Polylith** version 2.1, the current release. Documentation for later versions indicates support for more dynamism.

We now define a mapping for the classical blackboard system by describing each step of the blackboard algorithm in terms of the framework.

1. *Whenever the blackboard is changed by a KS, all KS's are notified of the change.*

   KS's map to participants, and the blackboard maps to a group containing a registrar and a router. The registrar records which KS's are interested in which messages. The router acts as a sequential shared repository that stores blackboard data, as described in Section 3.2.4. The blackboard broadcasts a `BB-Change` message whenever the blackboard is modified by a `BB-Write` message. The message has a `Change-Kind` attribute describing the kind of blackboard change.

2. *An AR is created for each KS interested in the change.*

   KS's register to receive `BB-Change` messages and have an MTF that filters those messages based on the `Change-Kind` attribute. On receipt of a `BB-Change` message, a KS sends a `KS-Activate` message to the scheduler, indicating that it wants to operate on the blackboard, and then waits for permission.

3. *For each interested KS, the scheduler adds an AR to the agenda.*

   The scheduler is a participant that maintains the agenda and AR's internally. On receipt of a `KS-Activated` message, the scheduler adds a corresponding AR to the agenda.

4. *If there is a KS operating on the blackboard, the scheduler waits. Otherwise, the scheduler gives one KS permission to operate on the blackboard.*

   On receipt of a `KS-Done` message, indicating that the current KS is finished operating on the blackboard, the scheduler selects a new AR from the agenda. The scheduler sends a `KS-Start` message to the associated KS to signal it to execute. Note that the selection of the "best" AR on the agenda is the famous blackboard *control problem.*

5. *The selected KS operates on the blackboard.*

   After receiving a `KS-Start` message, the KS operates on the blackboard by sending it `Read-BB` and `Write-BB` messages. Blackboard data is represented by the parameters of these messages. On receipt of a `Write-BB` message, the router holds it until some KS, say k, sends a matching `Read-BB` message, at which time the router sends the `Write-BB`'s parameters to k. Matches between `Read-BB` to `Write-BB` messages may be determined by message attributes, types, parameters, or other means.

   Note that each `Write-BB` message causes the blackboard to broadcast a `BB-Change` message, causing steps 1–3 to occur.

6. *The selected KS finishes operating on the blackboard. The process repeats from step 4.*

   When the KS finishes, it sends a `KS-Done` message to the scheduler.

5 Conclusions

By defining a generic, event-based integration framework, we have identified common components of event-based integration and provided a means for understanding, discussing, and comparing event-based integration approaches. The framework provides characterizations and partial formalizations of participants, registrars, routers, message transforming functions, delivery constraints, and the software module adaptation process. It supports dynamic and static specification of the mechanism and configuration and can model many useful integration approaches.

In the future, we plan to continue formalizing the model. For example, message transforming functions are well-defined, but delivery constraints are much harder to capture formally because they vary widely. Two other issues that bear investigation are *scalability* and *interoperability*. The framework supports composition via groups and attributes, allowing the model to scale, and we plan to investigate to what extent composition translates to actual scalability of implementations. The framework also provides the ability to identify similarities and differences between mechanisms, which aids in understanding interoperability issues between mechanisms. We plan to examine the extent to which this information can be used directly to support interoperability.

References

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

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

[3] 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.

[4] 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.

[5] 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.

[6] Nehru Bhandaru. Pilgrim Event Notifier. Project Pilgrim working draft, November 1991.

[7] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of The ACM*, 36(12):37–53, December 1993.

[8] Geoffrey Clemm and Leon Osterweil. A Mechanism for Environment Integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

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

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

[11] 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, 1991. OMG Document Number 91.12.1, Revision 1.1.

[12] 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.

[13] Dewayne E. Perry. Version Control in the Inscape Environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 142–149, March 1987.

[14] Digital Equipment Corporation. DEC FUSE: An Open, Integrated Software Development Environment. Product Information Sheet EC-F1550-48, Digital Equipment Corporation.

[15] 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.

[16] David Garlan and Ehsan Ilias. Low-cost Adaptable Tool Integration Policies for Integrated Environments. In *SIGSOFT Proceedings*, December 1990.

[17] David Garlan and Curtis Scott. Adding Implicit Invocation to Traditional Programming Languages. In *Proceedings of the 15th International Conference on Software Engineering*, 1993.

[18] 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.

[19] 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.

[20] V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors. *Blackboard Architectures and Applications*, volume 3 of *Perspectives in Artificial Intelligence*. Academic Press, Inc., 1989.

[21] Astrid Julienne and Larry Russell. Why You Need ToolTalk. *SunEXPERT Magazine*, pages 51–58, March 1993.

[22] Gail E. Kaiser and David Garlan. Melding Software Systems from Reusable Building Blocks. *IEEE Software*, pages 17–24, July 1987.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Paul B. Patrick, Sr. CASE Integration Using ACA Services. *Digital Technical Journal*, 5(2):84–99, Spring 1993.

[29] James M. Purtilo. The Polylith Software Bus. Technical Report UMIACS-TR-90-65, University of Maryland, May 1990.

[30] Steven P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, July 1990.

[31] 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.

[32] 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.

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

[34] Andy Wilson and Bob Travis. Application Control Architecture Specification. Digital confidential and proprietary document, May 1990.