Using Reflection for Flexibility and Extensibility in a Metacomputing Environment*

Anh Nguyen-Tuong, Steve J. Chapin, Andrew S. Grimshaw
{nguyen | chapin | grimshaw}@virginia.edu
http://legion.virginia.edu
Department of Computer Science
University of Virginia

Charlie Viles
viles@ils.unc.edu
School of Information and Library Science
University of North Carolina at Chapel Hill

Abstract

Legion is a large-scale metacomputing project at the University of Virginia. Legion users have requirements in many dimensions, including scheduling, security, fault tolerance, programming languages and environments, and performance. Not all users have the same needs. Further, as higher levels of services generally imply higher costs, users should be allowed to make tradeoffs and select the combination of services that is best suited for their purpose. To support diverse requirements Legion presents system developers with a reflective model, the Reflective Graph and Event model (RGE), for building metacomputing applications. The RGE model uses graphs and events to specify computations and enables first-class program graphs as event handlers. We demonstrate the RGE model in several areas of interest to metacomputing using Legion as our testbed. We unify the concepts of exceptions and events; by making exceptions a special case of events. Furthermore, using RGE, we demonstrate how to build generic, composable and reusable components that can be shared across development environments such as MPI, PVM, NetSolve, Ninf, C++, and Fortran.

Keywords: metasystems, metacomputing, graphs, events, exceptions, reflection, reflective architecture, component reuse

---

*This work is partially supported by DARPA (Navy) contract # N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), DOE D459000-16-3C and DARPA (GA) SC H607305A.

1 Introduction

The widespread deployment of gigabit networks will shrink the effective distance between computing resources and enable metasystems—wide-area distributed-object computing systems that consist of many heterogeneous, distributed, and unreliable resources. Without significant software support, metasystem users, e.g., resource owners, administrators, application writers, scientists, language designers, toolkit providers, corporations and government agencies, will not be able to manage the complexity of this environment. Metasystem software must meet users' requirements in several dimensions, including scheduling, security, fault tolerance, programming languages and environments, accounting, and ease-of-use. As higher levels of service generally imply higher costs, a metasystem should allow users to make tradeoffs and select the combination of services that is best suited for their purpose. For example, a bank may opt for a secure system at the cost of performance, a resource owner may wish to provide access to his machines but only between midnight and six o'clock in the morning, and a scientist may demand performance over other concerns. The set of requirements may change over time. The scientist who previously prized only performance may be willing to sacrifice some performance when resource owners decide to charge for usage.

To meet our users' expectations, we provided system developers—compiler writers, library writers, and toolkit developers—with a modular architecture that promotes flexibility, extensibility, reusability, and composability. By extensibility, we mean that developers can extend the functionality of the system within a consistent framework. By flexibility, we mean that developers can accommodate a vast range of user requirements.

By reusability, we mean that developers can encapsulate functionality into modular, reusable components. By composability, we mean that developers can compose components to meet users' requirements, e.g., security and accounting. Our intent is to spur the development of higher-level abstractions, tools and services for application programmers.

Our architecture is based on a reflective model of computation [33]. The basic design philosophy behind a reflective architecture is to expose—instead of hide—the elements that make up the structure of the system to developers. A reflective system is introspective; the system has a representation of itself that it can observe—its self-representation. Often, the self-representation of a reflective architecture is expressed in terms of abstract entities that are manipulated to modify the behavior of the system. Thus a reflective system promotes the writing of generic and reusable components that manipulate the self-representation. Such components may be written by domain experts and incorporated transparently into user applications. For example, Fabre *et al.* used a reflective architecture to incorporate fault-tolerance techniques into non-fault-tolerant applications [10], thereby freeing application programmers from the complex and error-prone task of implementing fault-tolerance algorithms. The versatility of reflective architectures has been demonstrated in several contexts, such as operating systems [32], programming languages [18][19][20][24], real-time databases [26], agent-based systems [8], and dependable systems [1].

In this paper, we present the Reflective Graph and Event model (RGE) and its application in the Legion metacomputing system [12]. RGE enables the manipulation of user computations at an abstract level by representing them as events, event handlers and

program graphs. These data structures are the self-representation of our reflective architecture and manipulating them is the basis for achieving flexibility, extensibility, reusability, and composability. The advantages of using an event-based architecture are well-known: components are decoupled from one another spatially and temporally, and they may be added/removed dynamically. Developers may extend object functionality by registering handlers with the appropriate events and by defining new events. A novel feature of the RGE event mechanism is that handlers may be executable program graphs that specify method invocations on remote objects. Graphs may be bound with their associated events at run-time, enabling the dynamic composition of functionality to objects.

The primary contributions of this paper are to present a computational model and structural framework for designing objects—the Reflective Graph and Event model; to enable the dynamic binding of policies to objects using first-class executable graphs as event handlers; and to unify the concept of exceptions and events.

We provide an existence proof of the applicability of the RGE model to the solution of real problems by demonstrating its use in Legion, an object-based metacomputing system. We demonstrate the versatility of the model by using it in such diverse areas as building a protocol stack for objects, defining a novel event notification model that includes exception propagation as a special case of the model, implementing a simple bag-of-tasks scheduler, and shutting down distributed applications gracefully.

The paper is organized as follows. In order to frame the RGE model within the context of the Legion project, we briefly describe the Legion system model in Section 2. Then, we describe the Reflective Graph and Event model in Section 3. In Section 4, we present

several applications of the RGE model to support our thesis that RGE is a viable technology for building metacomputing applications, incorporating our design goals of flexibility, extensibility, reusability, and composability. In Section 5, we discuss our experiences working with the RGE model. In Section 6, we present related work. We conclude in Section 7 and present areas of future research.

2 System Model

Before discussing the RGE model and its application in Legion, it will help to place the model within context of the overall Legion system.

Legion is an object-based metacomputing system that has been deployed at more than a dozen sites on three continents. Legion objects encapsulate both hardware and software resources. Objects are logically independent collections of data and associated methods with disjoint address spaces. Objects can contain one or more associated threads of control, and communicate via asynchronous method invocations. Objects are named entities identified by a Legion Object IDentifer (LOID). Objects are persistent and can be in one of two states: active or inert. Active objects contain one or more threads of control and are ready to service method calls. Inert objects exist as passive object state representations on persistent storage. Legion moves objects between active and inert states to use resources efficiently, to support object mobility, and to enable failure resilience. For a detailed description of the Legion object model, please see [14].

Legion provides a variety of programming interfaces on several different levels. Some programmers use Legion by writing programs in high-level languages such as parallel versions of C++, e.g., MPL [13]. Other programmers use Legion by specifying an object interface in an Interface Description Language (IDL), using an IDL compiler to generate

client and server stubs, and then providing the method implementations in a sequential programming language, e.g. CORBA [23]. Others use standard message-passing facilities such as PVM or MPI. Still others may use specialized domain toolkits, e.g., NetSolve [7] and Ninf [25]. Finally, another set of users—toolkit and middleware developers—require direct access to reflective aspects of the Legion run-time system, possibly to add new features and encapsulate them in the form of reusable components.

The RGE model targets the last set of users—toolkit and middleware developers—and provides a unifying architecture for developing components. The building blocks available to developers are reflective graphs and events. RGE program graphs are data-flow graphs whose nodes represent method invocations on objects and whose arcs represent data dependencies. The most important property of graphs is that they are first-class entities, and thus may be manipulated and passed as arguments to other objects for execution. Events provide a structuring mechanism for configuring services in a modular fashion—components may be added easily via event or graph handlers to provide new functionality.

3 Reflective Graph & Event Model (RGE)

Graphs and events specify the computation as it unfolds. Graphs represent interactions *between* coarse-grain objects; a graph node (vertex) is either a member function call on an object or another graph, arcs model data dependencies, and each input to a node corresponds to a formal parameter of the member function. Events specify interactions between components *within* an object's address space. A special kind of event, the *exoevent*, allows for graphs as event handlers. Thus, raising an exoevent results in remote method invocations, and enables remote objects to be treated as components. The ability

of events to regulate interactions both inside an address space and across address spaces is the foundation of the RGE model.

Below we first present graphs (Section 3.1), followed by events (Section 3.2) and exoevents (Section 3.3).

3.1 Graphs

Graphs[1] are the mechanisms in Legion for composing and invoking coarse-grained method functions on objects. Graph nodes are called actors and represent method invocation on objects, arcs denote data-dependencies between actors, and tokens flowing across arcs represent data or control information. When an actor has a token on each of its input arcs, it may "fire", i.e., execute its corresponding method, and deposit a result token on each output arc. Figure 1 illustrates a fragment of code written in C++-like syntax and the corresponding graph representation.

```
(1) main() {
(2)   int a = 10, b = 15, x, y, z;
(3)   MyObject A, B;
(4)   x = A.op1(a);
(5)   y = B.op1(b);
(6)   z = A.op2(x,y);
(7)   printf("z=%d\n", z);
(8) }
```

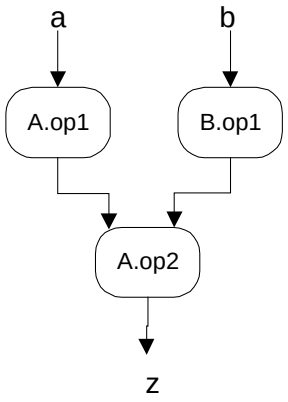


Figure 1. Sample code fragment and corresponding RGE program graph.

[1] Our use of graphs originated in the Mentat project [13][15], a high-performance object-oriented parallel processing system.

Unlike a traditional client/server model, the results from the method invocations on lines 4 and 5 do not return to the `Main` object.[2] Instead they are forwarded directly to `A.op2`. Upon executing the graph, `Main` sends each node a list of objects that should receive the return values and any out parameters.

Graphs are first-class entities and may be assembled at run-time, transformed, passed as arguments to other objects, and executed remotely. The interface to the graph facilities consists of library routines to build graph nodes, add tokens, add arcs, and annotate nodes, arcs, and tokens. Calls to these can be hand-coded or generated by a compiler front-end or other automated tool. The library also provides routines to execute graphs, probe graph status, and wait on return values. A sample of these features is illustrated in Figure 2, where we show a typical compiler transformation of the code in Figure 1.

In addition to the basic graph operations shown in Figure 2, graphs may be annotated with <name, type, value> triples. The name field is an arbitrary string, the type field is a string that indicates the type, and the value field consists of arbitrary data. The name and type fields dictate the interpretation of the value field. Annotations are properties tied to individual arcs and nodes, e.g. to aid in scheduling we may annotate a node with information such as "Architecture=C90", "Memory Usage=20MB", to indicate architectural restrictions and resource requirements. Similarly we might annotate a node with "Semantic Property=Stateless" to indicate that the function is a pure function – the result depends only on the inputs.

Annotations may propagate through the object method invocation chain, in which case we call them *implicit parameters*. For example, if object A annotates its graph with an

[2] A client/server call is a special case of a graph with two nodes: one for the server and the other for the return

implicit parameter, invokes a method on object B, and B invokes a method on object C, A's implicit parameter propagates to C. Implicit parameters provide a mechanism for adding meta-level information transitively.

As a more concrete example, to monitor message flow dynamically, an application can propagate the identity of a logger object to all objects. Subsequently, each object may build a graph and execute a method on the logger object to pass status information pertaining to the message stream. Implicit parameters are similar to CORBA's contexts in that they denote meta-level information and are part of the environment when executing a method.[3] The primary difference with CORBA's contexts is that implicit parameters propagate automatically through the method invocation call chain.

---

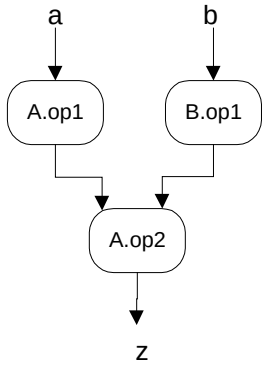
value to the client.

[3] Implicit parameters are also reminiscent of Unix environment variables.

Code Fragment

```
MyObject A, B;
x = A.op1(a);
y = B.op1(b);
z = A.op2(x,y);
```

Graph Representation



Graph Implementation

```
(1) // Declarations
(2) LegionInvocation inv1, inv2, inv3; // graph nodes
(3) LegionParameter parm;
(4) int a = 10, b = 15, z;

(5) // Object creation
(6) LegionLOID A_name, B_name;
(7) A_name = Legion.CreateObject("MyObject");
(8) B_name = Legion.CreateObject("MyObject");

(9) // Create graph and handles
(10)LegionProgramGraph G(Legion.getMyLoid());
(11)LegionCoreHandle _handle(A_name),B_handle(B_name);

(12)// x = A.op1(a);
(13)inv1 = A_handle.invoke("op1", 1, 1);
(14)G.add_invocation(inv1);
(15)parm = make_parameter(a, 1);
(16)G.add_constant_parameter(inv1, parm,1);

(17)// y = B.op1(b);
(18)inv2 = B_handle.invoke("op1", 1, 1);
(19)G.add_invocation(inv2);
(20)parm = make_parameter(b,1);
(21)G.add_constant_parameter(inv2, parm, 1);

(22)// z = A.op2(a,b);
(23)inv3 = A_handle.invoke("op2", 2, 1);
(24)G.add_invocation(inv1);
(25)G.add_invocation_parameter(inv1, inv3, 1, 1);
(26)G.add_invocation_parameter(inv2, inv3, 1, 2);

(27)G.execute(); // Execute program graph

(28)// Retrieve the return value – print value of z
(29)// <buffer> is a data structure to store arbitrary
    data
(30)buffer = G.get_value(inv3, METHOD_RETURN_VALUE);
(31)buffer.get_int(&z, 1);
(32)printf("z=%d\n", z);
```

Figure 2. Example of graph API. Lines 1-32 implement the graph shown on the left. The first step before building the graph is to create and provide handles to our objects (lines 5-11). For each graph node, we specify the method to invoke, and the number of input and output arcs. A graph node is represented by a `LegionInvocation` (lines 13, 18, 23). Then we create the tokens (arguments) and attach them to the appropriate graph nodes (lines 15-16, 20-21, 25-26). There are two kinds of tokens, *constant* tokens and *invocation* tokens. Constant tokens are those for which we have an actual value (lines 15-16, 20-21) whereas invocation tokens are those for which the values are results from other invocations (lines 25-26). Once the graph is built, we execute it (line 27) and block waiting on `z`, the final output value (line 30).

3.2 Events

There are several ways of structuring objects to support a variety of functions, ranging from the *ad hoc* gluing of components to the establishment of well-defined interfaces. A common way of structuring objects is to use a protocol stack—abstractions or functionality are layered on top of one another. If the set of functions supported by such a protocol stack is static, then hard-wiring components is a suitable approach. On the other hand, if the set of functions is expected to change, a flexible approach is required. In metacomputing systems, the latter approach is needed as the set of services supported by objects is driven by a wide set of user requirements.

We adopted an event-based paradigm for structuring objects. Events are introspective and specify the structural implementation of objects. Events provide a unifying mechanism for inter-component interaction; they are conceptually easy to understand and are familiar to programmers; and they allow the development of components in isolation from other components. Finally, they enable the easy addition or deletion of components, providing a basis for extending the functionality of objects. As described in Section 4.1, we use events to configure the Legion object protocol stack.

Our event model is defined by events, event kinds, event handlers and event managers. An *event* contains user-defined data and a tag that denotes its *event kind*. Each event has one or more associated *event handlers* that may be called whenever an event of that kind is announced. Handlers for a particular event kind are given priorities that determine their execution order. Any handler of a particular priority can postpone or prevent the execution of handlers with lower priorities. Events are announced, or raised, in one of two modes, asynchronous or synchronous. In the former case, an *event manager* stores

the event in an internal queue for later delivery. In the latter, the handlers are invoked immediately.

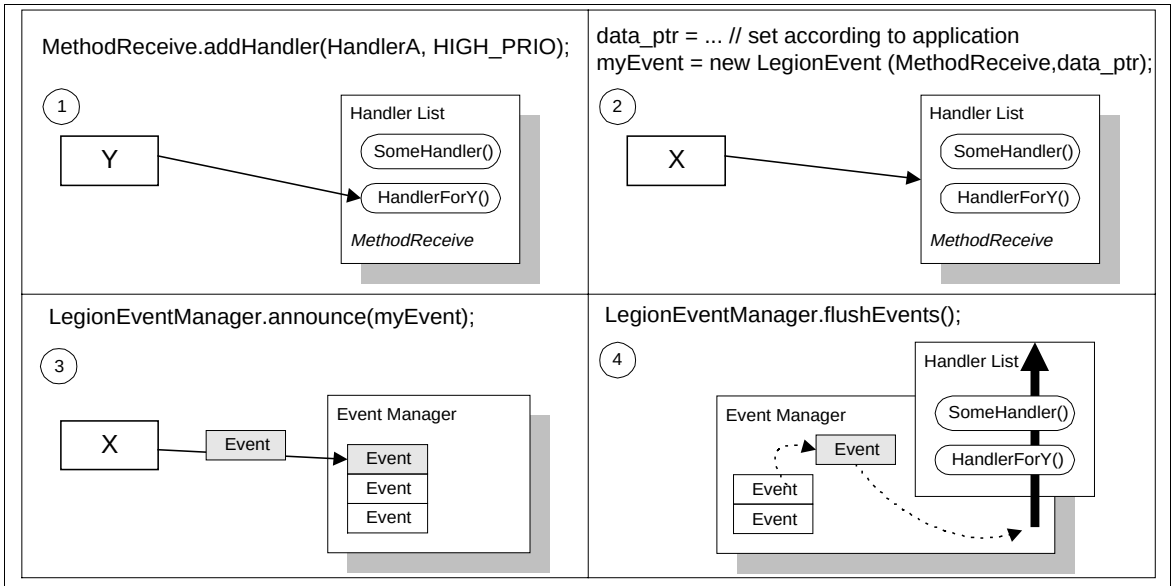


Figure 3. Component communication using events. (1) Y registers HandlerForY() with the *MethodReceive* event. (2) X creates an event. (3) X raises the event by invoking the Event Manager. (4) The Event Manager invokes the handlers.

For example, Figure 3 illustrates communication between two components X and Y within the same object: (1) Y registers a handler for the particular event kind that X will announce. (2) X creates an event using one of the provided event kind as a template. X may attach event-specific data as well. (3) X announces the event to an event manager, which enqueues events and ensures that its handlers are executed in priority order and only if preceding handlers have not prevented further execution. (4) The event manager dequeues and processes the event by calling the associated handlers. Note that apart from application-specific data manipulation, each of these actions requires developers to write only one or two lines of code.

The event facilities enable flexibility and extensibility by allowing components to add, modify and remove handlers. New event kinds may be added, and handler priorities maybe set or reset to affect the order in which handlers are processed.

3.3 Exoevents

In the previous section, events and their handlers resided in the same address space. Exoevents extend events to inter-address space communication. An *exoevent* is an event whose handler is represented by a program graph. Thus, raising an exoevent will result in a set of method invocations on remote objects. The set of method invocations is bound to the event dynamically and is specified by an executable first-class program graph, effectively implementing a "very long jump". The ability to defer the binding of graphs to events until run-time provides flexibility in implementing policy decisions. Object designers need not anticipate all the myriad ways in which their objects will be used. As an example, consider an object that raises a run-time exception. Where should the exception propagate? Should it be to the immediate caller or perhaps to an exception-monitoring object?

To indicate an interest in an exoevent, a remote object associates a program graph with the exoevent. Typically, the graph is a callback graph: when the exoevent is raised, the graph specifies a method invocation back on the remote object. More complex interactions are possible: the graph may specify a method invocation on a third-party object or a sequence of method invocations on several objects. The association of graphs with exoevents is performed at run-time, and may be set on a per object or per method basis. In the per object case, the association of graphs and exoevents persists across method calls. In the per method case, the association is temporary and valid only for the duration of a single method call.

Exoevents form a substantive portion of the Reflective Graph and Event model and embody a "mechanism, not policy" philosophy. RGE provides hooks to attach event

propagation policies dynamically. The benefit for object designers is that they need not anticipate all possible policies when building their objects. This is illustrated in Section 0, in which a single, shared, server object supports multiple exception propagation policies.

Before we proceed any further, we must first define the terms exoevent, exoevent interest, and exoevent interest set (EIS). An *exoevent* is a set of 3-tuple items <item-name, data-type, data-value>. The *item-name* field is a string to identify an item; the *data-type* specifies how to interpret the *data-value* field of an item. Items may be added or removed from an exoevent. Users may search for a specific item by using the name field as a key. By convention, all exoevents contain an item with item-name="ExoEventType". The data-type field is a string describing the type of exoevents. By convention, we classify exoevent types within broad categories and further divide them using a ":" to delineate subcategories, e.g., "Exception", "Warning", "Exception:Security", "Exception:Security:Access Control". An *exoevent interest* is a 2-tuple <exoeventType, notificationGraph> that associates an exoevent type with a computation graph. The exoevent type specifies the kind of exoevent of interest. The notificationGraph is a first-class program graph and specifies a computation to be executed if a match is made between an exoevent and an exoevent interest. An *exoevent interest set* (EIS) is a set of exoevent interests. An EIS may be used to specify interests in multiple exoevents or to specify multiple graphs for the same exoevent.

3.3.1 Specifying interest in an event (per method association)

To raise an exoevent, we use a *RaiseNotification* event. The data field of the event contains the raised exoevent. The handler for the event performs a matching function between the raised exoevent and each exoevent interest in the exoevent interest set. If the

handler finds a match, it extracts and executes the notification graph contained in the exoevent interest. If there are multiple matches, then all graphs are executed.

To specify interest in an exoevent for the duration of a single method call, an object creates an exoevent interest, inserts it into an exoevent interest set, (EIS) and annotates the program graph associated with the computation. Since implicit parameters propagate automatically, the EIS will be available to all objects in the call chain that raise an exoevent, i.e., if `S.service()` invokes methods on other objects, the EIS will propagate to them as well (Figure 4). When an object raises an exoevent, it inspects the exoevent interests contained in the EIS to search for a match based on the exoeventType field. If a match is found, the corresponding notificationGraph is then executed.

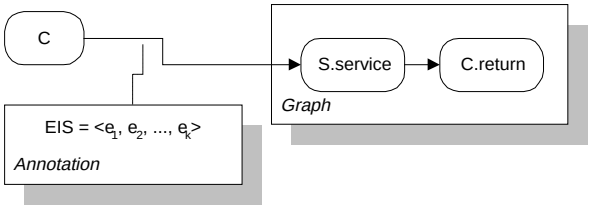


Figure 4. Graph annotated with exoevent interest set (EIS).

Consider the example in Figure 4 that corresponds to the following code fragment (`C` is the client object, `S` is a server object):

At the client: `x = S.service();`

The exoevent interest set specified by `C` is valid during the execution of `S.service()`. This is an important aspect of the model as it enables a single object to support multiple event notification policies, selecting among them on a per method invocation basis.

3.3.2 Specifying interest in an event (per object association)

Exoevent interests may also be specified persistently at the object level and be valid across all method calls to that object. To support this functionality an object uses the methods:

```
RegisterNotification(LOID, ExoeventInterest);
UnregisterNotification(LOID);
```

The Legion Object Identifier (LOID) is used to identify the object that registers an interest and to unregister a previous set of registrations. Note that an object may register more than one ExoeventInterest, each with its own notification graph.

Object-level and method-level scoping of notification interests may be specified simultaneously: a single raised exoevent may result in several graphs being executed—some being specified via graph annotations and others via the `RegisterNotification()` method of the object in which the exoevent occurs.

4 Applications of the RGE model

We have presented events, exoevents and graphs, the basic building blocks of the RGE model. Next, we demonstrate their utility and versatility in:

- designing a configurable protocol stack (Section 4.1),
- supporting multiple exception propagation policy simultaneously (Section 4.2),
- implementing of a bag-of-task scheduler (Section 4.3),
- implementing a distributed application shutdown algorithm (Section 4.4).

The applications described in this section are examples of reusable and composable components. They have been implemented and are deployed in Legion across several development environments, including PVM, MPI, MPL, NetSolve and Fortran. These applications of the model are not meant to be an exhaustive list of the ways we have used

RGE in Legion. Instead, we illustrate the model's applicability to a variety of needs and informally show its application in other domains.

4.1 Configurable protocol stack

One of the primary applications of the RGE model is to implement a configurable protocol stack for Legion objects [31]. A striking feature of the protocol stack is that only a few events are employed. These events may be classified into three broad categories: message-related, method-related and object management-related events. Events reflect the fact that Legion is an object-based system; objects communicate with method invocations, which are implemented at the low level over message passing. Table 1 describes several event kinds used in configuring the protocol stack.

Category	**Event Kind**	**Description**
Message-related events	MessageReceive	Object has received a message
	MessageSend	Object is sending a message
	MessageComplete	Message has been successfully sent
	MessageError	Error in sending message
Method-related events	MethodReceive	Object has received a complete method invocation; all parameters have been received
	MethodReady	A method has passed the security method access control check and is ready to be serviced
	MethodSend	Object is invoking a method on a remote object
	MethodDone	Object is done servicing a method
Object-related events	ObjectCreated	An object has been created
	ObjectDeleted	An object has been deleted

Table 1. Some of the events used to configure the protocol stack of Legion objects.

Figure 5 illustrates the major components of the Legion protocol stack. To invoke a method on a remote object, the `GraphComponent` announces a *MethodSend* event for

each node in the graph that has the sender as a source of an input token. In turn, the `MessageLayerComponent` bundles parameters into a message and announces a *MessageSend* event. Finally, the `NetworkComponent` sends the message over the network.

When an object receives a message from the network, it announces a *MessageReceive* event. The `MethodAssemblyComponent` determines whether the received message is sufficient to form a complete method invocation (recall that in data flow multiple messages may be required to trigger a method execution). If the message results only in a partial method invocation, the object stores the message in an internal database. When the required messages arrive to complete the method invocation, a *MethodReceive* event is raised. At this point, the `MethodInvocationComponent`, stores the complete method in a database of ready methods. Then, a server loop may extract ready methods from the database and execute them.

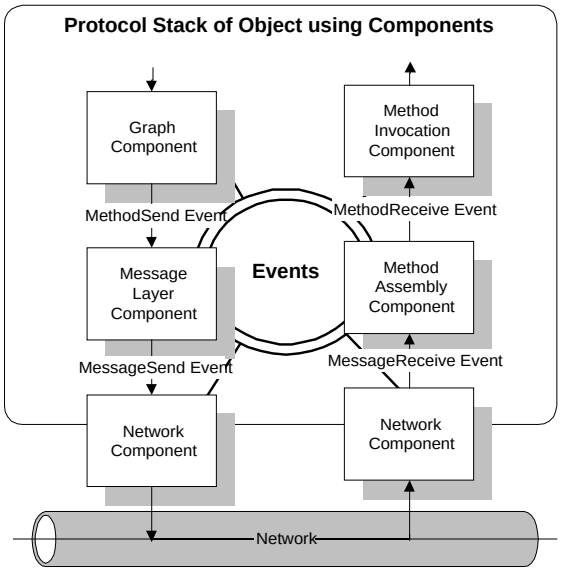


Figure 5. Major components of the Legion protocol stack.

This sequence of events implements our basic requirements for our protocol stack. To extend the basic implementation, e.g., to add security, is simple. For example, adding a security component to encrypt and decrypt the message stream is a matter of registering handlers with the *MessageReceive* and *MessageSend* event kinds. Typically, the encryption handler is the last one registered with *MessageSend* while the decryption handler is the first one invoked as a result of *MessageReceive*. In this example, we assume that the two objects have agreed *a priori* to use compatible encryption/decryption routines. Mechanisms for negotiating protocols are outside the scope of this paper.

4.2 Propagating exceptions

A natural way of exploiting the RGE model is for exception propagation between objects. Note that in RGE, exceptions and events are no longer separate concepts—exceptions are simply a special case of events.

Now we demonstrate how a single server can support multiple exception propagation policies simultaneously (Figure 6). Consider a server object `S` used by two applications, `AppA` and `AppB`. `AppA` specifies that objects propagate exceptions back to `AppA`. `AppB` specifies that exceptions propagate to a third-party object, `ExceptionMonitorB`. Finally, the creator of `S`, `CreatorS`, specifies that all exceptions raised by `S` shall propagate to `CreatorS`. `AppA` and `AppB` use implicit parameters to specify an exoevent interest set (per method association), whereas `CreatorS` registers its interest with S directly (per object association).

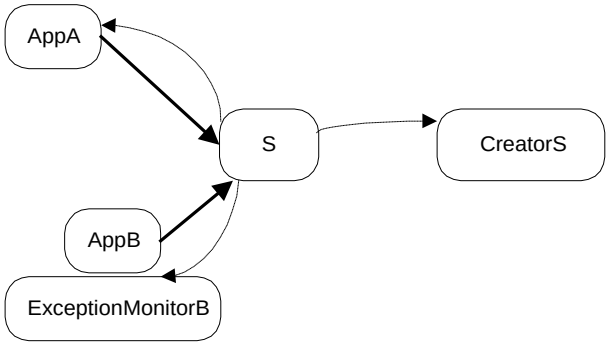
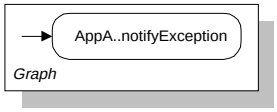
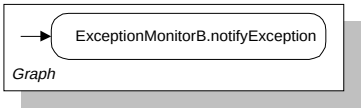


Figure 6. Object `S` supports multiple exception propagation policies. Exceptions raised servicing method invocations (straight arrow) on behalf of `AppA` result in a callback method on `AppA` (curved arrow). Exceptions raised on behalf of `AppB` (straight arrow) result in a method invocation on `ExceptionMonitorB` (curved arrow). Any exceptions raised by S (regardless of the invoker) result in a method invocation on `CreatorS` (curved arrow).

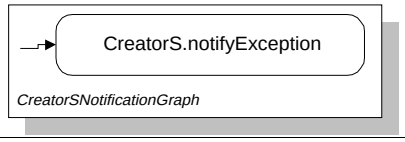
`AppA` and `AppB` insert the following exoevent interest set in their implicit parameters to implement the above policies (note that the difference in policies is expressed by the notification graph):

Exoevent interest	**For AppA**	**For AppB**
ExoeventType	*Exception*	*Exception*
NotificationGraph		

`CreatorS` registers with `S` to be notified of all exceptions raised by `S` using:

```
S.registerNotification(CreatorSLoid, ExoEventInterest);
```

Where `ExoeventInterest` is given by:

ExoeventType	*Exception*
NotificationGraph	

Consider the case when `AppA` or one of its objects invokes a method on `S` that results in an exception by `S`. Since `AppA` inserted an exoevent interest set in its implicit parameters, the interest propagates to `S` automatically. Upon raising the exception, `S` finds the notification graph inserted by `AppA` and executes it. `AppA` is thus notified of the exception via the callback method, `AppA.notifyException()`. Furthermore, since `CreatorS` registered an interest, it too will be notified of the exception via the method `CreatorS.notifyException()`.

Now consider the case when `AppB` or one of its objects invokes a method on `S`. `S` finds the notification graph inserted by `AppB` and executes it. This time, the graph specifies the callback method, `ExceptionMonitorB.notifyException()`. `ExceptionMonitorB` is thus notified of the exception. As in the previous example, `CreatorS.notifyException()` will be invoked as well.

These examples demonstrate the flexibility of binding policy to objects at run-time. Designers of object `S` need not worry about where to propagate exceptions; they need only raise them. Furthermore, `S` is able to support multiple policies simultaneously by virtue of not supporting *any*—the policies themselves are specified dynamically by objects external to `S`.

4.3 Bag-of-tasks scheduling

We next illustrate how we have used the RGE model to implement a self-scheduling policy for stateless objects—objects that embody purely functional method invocations.

We exploit the functional nature of stateless objects to instantiate multiple worker instances and distribute method calls among them (Figure 7).

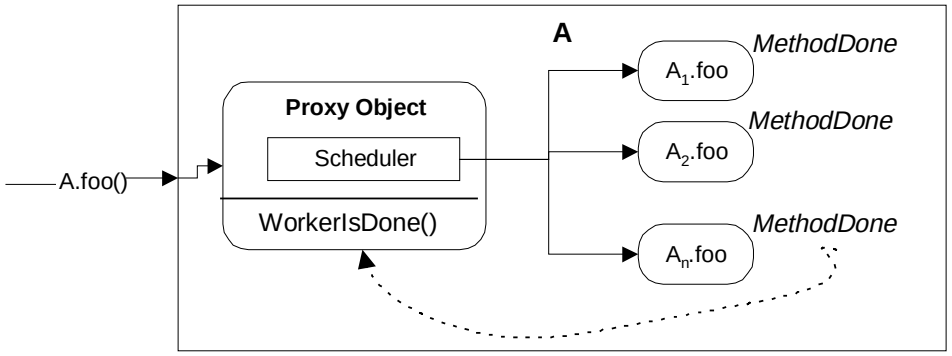


Figure 7. `Proxy` object selects a worker for servicing a method call. When the worker is done, it raises a *MethodDone* event. The graph associated with the *MethodDone* event results in a callback method invocation `WorkerIsDone()` on the `Proxy`.

Calls to stateless objects are routed to a `Proxy` object, which then assigns method call requests to one of the workers. The default scheduling policy for `Proxy` was random placement—a worker was selected at random to service method calls.

The `Proxy` is a natural place for experimenting with different scheduling policies. We illustrate how we modified the scheduling policy using the RGE model. Note that neither clients nor workers were aware of the policy change.

Before dispatching a method to the workers, the `Proxy` object builds a notification graph with node `Proxy.WorkerIsDone()`and associates it with the *MethodDone* event exported by the workers. The data field of the event carries a computation ID used by `Proxy` to keep track of ongoing computations. When a worker finishes its assigned computation, it raises the *MethodDone* event. This results in the execution of the notification graph inserted by the `Proxy.` `Proxy` is notified of the completion of a

method call and dispatches a new method invocation to the same worker, assuming that there is work available, thereby achieving self-scheduling.

The presence of events and graphs supporting reflective objects greatly simplifies the task of developing this scheduler. We replaced the default policy with a more sophisticated one, without having to change or modify the workers themselves. The only additional code required is the `WorkerIsDone()` callback method and minor bookkeeping code in the `Proxy` object, and the graph for the worker to perform the callback into the `Proxy` object.. This was possible because of the dynamic binding of graphs with the *MethodDone* event. Further refinements are possible. For example, we could add fault-tolerance capabilities by having the `Proxy` re-issue lost computations after a timeout interval as we did in an earlier version of Legion [22].

4.4 Shutting down distributed applications

Our last example illustrates an algorithm for shutting down a distributed application, perhaps in response to a keyboard interrupt from the user. At the heart of this algorithm is the use of RGE to keep track of the current set of objects in an application. An object is considered to belong to an application if it was created by a `Main` object—the first object started in an application—or by a child of the `Main` object. As the application progresses, the set of objects may grow/shrink as objects create/delete other objects.

To simplify the discussion, we assume that there are no failures. The `Main` object keeps track of its objects as they are created and deleted by inserting the following exoevent interests in its exoevent interest set:

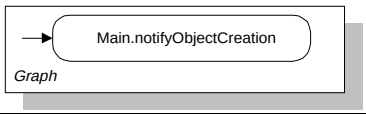
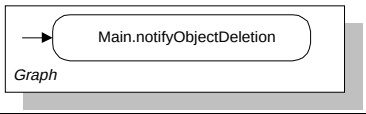
ExoeventType	*ObjectCreated*
NotificationGraph	
ExoeventType	*ObjectDeleted*
NotificationGraph	

Table 2. Keeping track of objects creation/deletion.

The exoevent interest set propagates to all of the `Main` object's children. Whenever an object is created or deleted, the appropriate graph is executed and `Main` is notified via either an `ObjectCreated()` or `ObjectDeleted()` method call (Table 2). Thus, the `Main` object has a current list of all objects. To shutdown the distributed application, the `Main` object invokes the method `DeleteSelf` on each object in its list[4].

The `Main` object must also deal with two potential race conditions:

- In-transit deletion notification. The `Main` object is not yet notified of the deletion of an object and will attempt to invoke a `DeleteSelf` method.
- In-transit creation notification. The `Main` object is not yet notified of the creation of an object and hence does not invoke the `DeleteSelf` method.

In-transit deletion notifications are not a problem. When the `Main` object attempts to invoke the `DeleteSelf` method on an already dead object, the invocation will simply fail. In-transit creation notifications can be handled by requiring that before objects respond to the `DeleteSelf` invocation, they must ensure that they have no pending object creations.

[4] DeleteSelf() is an object-mandatory method that all Legion objects must support.

In this example, we used the RGE model to monitor the set of objects in an application. Except for the `Main` object, objects did not need to be modified to support the shutdown algorithm.

5 Discussion

In the examples described above, the RGE model was used as the basis for supporting and extending object functionality. The default set of events for objects reflects the Legion model of computation—an object-based system implemented over a message-passing layer. We believe this set to be sufficient for most purposes as many algorithms may be expressed in terms of manipulating messages and methods.

Although we target toolkit and middleware developers, we often find it useful and simpler to wrap commonly used policy in higher-level functions. For example, a common way to use the exception propagation model is to propagate exceptions back to the caller. Thus, we provide developers with functions to implement this policy, without requiring them to interact with either the event or graph interface.

While we have discussed the RGE model and its use by developers, we have not discussed the interface provided to application writers. For the most part, application writers never interact with graphs or events. In general, events are raised unbeknownst to application writers. For example, in PVM, MPI, MPL, Fortran, and NetSolve, the distributed shutdown algorithm is triggered when a user hits Control-C. In MPL, the *MethodDone* events used in the bag-of-tasks scheduling example and the *ObjectCreated/ObjectDeleted* events are raised automatically by compiler-inserted code.

The ability to compose policies has been invaluable in meeting our users' requirements and in deploying Legion itself. Based on our experiences, we believe that, provided policies are composable at a high-level, we can map them onto the RGE model.

6 Related Work

The RGE model provides a blueprint for structuring distributed applications based on reflective principles. The concept of reflection is not novel; its use has been advocated in several contexts, including operating systems [32], programming languages [18][19][20][24], soft real-time systems [17], real-time global databases [26], agent-based systems [8], dependable systems [1], and in general, to incorporate non-functional requirements into user applications [27].

To our knowledge, RGE is the only reflective model that uses graphs and events as data structures for representing computations. RGE uses the Macro-Data Flow model to express and specify method invocations between objects [15]. Other data-flow systems include Paralex [2], CDF [3], HeNCE [4] and Code/Rope [9]. Unlike most graph systems, RGE graphs are exposed to toolkit and middleware developers; they can be assembled dynamically and executed remotely. RGE graphs are reflective: graphs are the self-representation of a computation and transforming graphs has a direct impact on the future of a computation. Furthermore, we use graphs in a novel way and allow them to be associated with events. This enables us to encapsulate functionality using graphs and dynamically bind such functionality to objects.

Our model shares many characteristics with projects such as SPIN [5], Coyote [6] and Ensemble [16] that use an event architecture as the basis for flexibility, extensibility, and component interaction. One may view Legion as a "configurable operating system"

specifically designed for metacomputing. However, Legion does not replace the operating system on host machines but provides a middleware layer between the native operating system and applications.

CORBA's Event Notification Service (ENS) [23] and Java's Distributed Event Specification (DES) [29] provide an event-based notification service. In both, objects must export a well-defined interface to be notified of an event. The RGE model is more flexible and enables an arbitrary set of methods. Furthermore, in RGE, the concepts of exceptions and events are unified—exceptions are simply special kinds of events—in contrast to both CORBA and Java.

Java defines two event models: Java DES to specify the propagation of events between objects on different virtual machines, and Java Beans [30], to specify component interaction inside a single virtual machine. Java DES outlines an approach for transforming Java Beans events into Java Distributed events. We take a similar approach in RGE to export internal events and make them visible to remote objects.

Globus is another metasystem project [11]. The primary difference between Globus and Legion is a philosophical one: Globus employs a "sum-of-service" approach for supporting users and specifies standard interfaces for such functions as security and resource management. Legion employs an "architecture" approach—system developers target a unified model that enables component reuse and interoperability. The two approaches are not mutually exclusive. For example, we have already mapped the two standard message-passing APIs, PVM [28] and MPI [21], onto the RGE model and Legion.

7 Conclusion

We have presented a reflective computational model, the Reflective Graph and Event (RGE) model, and demonstrated its use in Legion, an object-based metacomputing environment. While we used Legion as a proof-of-concept, the model is applicable for metacomputing in general. We have chosen *reflection* as the design philosophy behind our model because it has been shown to support extensibility, flexibility, composability and reusability, in other contexts such as extensible operating systems and programming languages. Now, we have applied reflection to metacomputing.

Novel features of our models are the uses of graphs and events to specify and represent computations, to allow executable program graphs as event handlers using exoevents, to enable the late binding of policies to objects, and to present an event propagation model that unifies the concept of exceptions and events.

To show the versatility of the RGE model, we presented four applications of the model that have been deployed in Legion: building a configurable protocol stack for objects, defining a novel event notification model that unifies the concept of exceptions with events, and implementing a bag-of-tasks scheduler and a distributed application shutdown algorithm.

RGE encourages the encapsulation of functionality inside reusable components—components developed by one set of system developers may be reused by another. The RGE model provides a structural framework in which components may be composed together in a unified and consistent manner. Thus, not only are the examples shown in this paper deployed and available to Legion's PVM, MPI, NetSolve, C++, and Fortran users, they may also be used within a single application *simultaneously*.

Future work consists of further developing components along several dimensions: fault tolerance, security, and resource management, to name only a few. Over time, we hope to present system developers with an extensive component library; each component having its own costs and benefits tradeoffs. As components are developed by one set of system developers, they can be made available to others as well. Our model encourages this practice as RGE components are by definition generic in nature—they manipulate the underlying computation at an abstract level. Another research direction is to map additional environments and languages onto our model to increase the range of choices afforded end users.

It will be interesting to observe whether the metacomputing community will converge on a standard set of components or whether many components with varying cost and benefit characteristics will emerge. While it is too early to tell, the RGE model provides an experimental platform for the quick prototyping and deployment of components.

8 References

[1] G. Agha and D. C. Sturman, "A Methodology for Adapting Patterns of Faults", *Foundations of Dependable Computing: Models and Frameworks for Dependable Systems*, Kluwer Academic Publishers, Vol. 1, pp. 23-60, 1994.

[2] O. Babaoglu *et al.*, "Paralex: An Environment for Parallel Programming in Distributed Systems", *Technical Report UBLCS-92-4*, Laboratory for Computer Science, University of Bologna, Oct. 1992.

[3] R. F. Babb, "Parallel Processing with Large-Grain Data Flow Techniques", *IEEE Computer*, pp. 55-61, July 1984.

[4] A. Beguelin *et al.*, "HeNCE: Graphical Development Tools for Network-Based Concurrent Computing", *Proceedings SHPCC-92*, pp. 129-36, Williamsburg, VA, May 1992.

[5] B. Bershad *et al.*, "Extensibility, Safety and Performance in the SPIN Operating System", *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pp. 267-284, Copper Mountain, CO, 1995.

[6] N. T. Bhatti, *et al.*, "Coyote: A System for Constructing Fine-Grain Configurable Communication Services", *Department of Computer Science Technical Report TR 97-12*, University of Arizona, July 1997.

[7] H. Casanova and J. Dongarra, "NetSolve's Network Enabled Server: Examples and Applications", *IEEE Computational Science & Engineering, pp. 57-67, September 1998*.

[8] P. Charlton, "Self-Configurable Software Agents", *Advances in Object-Oriented Metalevel Architectures and Reflection*, CRC Press, pp. 103-127, 1996.

[9] J. C. Browne, T. Lee and J. Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment", *IEEE Transactions on Software Engineering*, pp. 111-120, February 1990.

[10] J. C. Fabre *et al.*, "Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming", *The Twenty-fifth Symposium on Fault-Tolerant Computing*, pp. 489-498, 1995.
[11] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit", *International Journal of Supercomputing Applications*, 1997.
[12] A. S. Grimshaw *et al.*, "Metasystems", *Communications of the ACM*, pp. 46-55, November 1998.
[13] A. S. Grimshaw, A. Ferrari and E. West, "Mentat", *Parallel Programming Using C++*, The MIT Press, Cambridge, Massachusetts, pp. 383-427, 1996.
[14] A. S. Grimshaw *et al.*, "Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems ", *Department of Compter Science Technical Report CS-98-12*, University of Virginia, June 1998.
[15] A. S. Grimshaw, J. B. Weissman and T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing", *ACM Transactions on Computer Systems*, Vol. 14, Num. 2, 1996.
[16] M. Hayden, "The Ensemble System", *Cornell University Technical Report*, TR98-1662, January 1998.
[17] Y. Honda and M. Tokoro, "Soft Real-Time Programming through Reflection", *Proceedings of the International Workshop on New Models for Software Architecture: Reflection and Metalevel Architecture*, pp. 12-23, 1992.
[18] G. Kiczales, J. D. Rivieres and D. G. Bobrow, "The Art of the Metaobject Protocol", MIT Press, 1991.
[19] A. H. Lee and J. L. Zachary, "Reflections on metaprogramming", *IEEE Transactions on Software Engineering*, vol. 21, pp. 883-892, November 1995.
[20] P. Maes, "Concepts and Experiments in Computational Reflection", *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 147-55, October 1987.
[21] Message Passing Interface Forum, "MPI: a message-passing interface standard", May 1994.
[22] A. Nguyen-Tuong *et al.*, "Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System", *Proceedings of the 15th Symposium on Reliable Distributed Systems*, Ontario, Canada, pp. 2-11, 1996.
[23] OMG, "The Common Object Request Broker: Architecture and Specification", *OMG*, 1995.
[24] A. Paepcke, "PCLOS: Stress testing CLOS: Experiencing the metaobject protocol", *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1990.
[25] M. Sato *et al.*, "Ninf: A Network based Information Library for a Global World-Wide Computing Infrastracture", *Proceedings of High Performance Computing and Networking (HPCN '97)*, (LNCS-1225), pp. 491-502, 1997.
[26] J. A. Stankovic and S. H. Son, "An Architecture and Object Model for Distributed Object-Oriented Real-Time Databases", *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'98*), Kyoto, Japan, August 1998.
[27] R. J. Stroud and Z. Wu, "Using Metaobject Protocols to Satisfy Non-Functional Requirements", *Advances in Object-Oriented Metalevel Architectures and Reflection*, Chapter 3, CRC Press, pp. 31-52, 1996.
[28] V. S. Sunderam, "PVM: A framework for parallel distributed computing", *Concurrency, Practice and Experience*, pp. 315-339, December 1990.
[29] Sun Microsystems, "Distributed Event Specification", *http://java.sun.com/products/jini/specs/*, September 1998.
[30] Sun Microsystems, "JavaBeans™", *http://www.javasoft.com/beans/*, September 1998.
[31] C. L. Viles *et al.***, "**Enabling Flexibility in the Legion Run-Time Library", *International Conference on Parallel and Distributed Processing Techniques*, Las Vegas, NV, 1997.
[32] Y. Yokote, "The Apertos Reflective Operating System: The Concept and its Implementation", *Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 414-434, 1992.
[33] C. Zimmermann (Ed.), "Advances in Object-Oriented Metalevel Architectures and Reflection", *CRC Press*, 1996.