1992

Architecture of the RELAX Problem Solving Environment

H. Scott McFaddin

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
92-081
ARCHITECTURE OF THE RELAX
PROBLEM SOLVING ENVIRONMENT

H. Scott McFaddin
John R. Rice

CSD-TR-92-081
August 1992
Architecture of the RELAX Problem Solving Environment

Scott McFaddin        John R. Rice

August, 1992

Abstract

RELAX is a problem solving environment for collaborating partial differential equation (PDE) solvers and editors. RELAX provides both a computational and user interface environment. The computational environment coordinates teams of single domain PDE solvers, which collaboratively solve mathematical systems called composite partial differential equation problems. The user interface environment coordinates multiple interactive user interface components, called editors, which display or alter any feature of a composite PDE problem. Editors may be both text-oriented (e.g., equation editors) and graphics-oriented (e.g., solution plotters).

The RELAX architecture is based upon a set of inter-communicating software components. Editors and single-domain PDE solvers are examples of RELAX components – these particular ones are externally supplied (perhaps from libraries). RELAX also provides several intrinsic system components which drive the environment. RELAX provides a message passing mechanism for supporting inter-component communication – this mechanism provides one-to-one message transmission as well as a form of message broadcasting. This paper describes the architecture of the RELAX environment. We outline the function and structure of each component as well as the message passing protocol which integrates these components into a working system.

1 Introduction

The design of RELAX has three goals. The first goal is to build a system which would be capable of integrating existing scientific software for partial differential equations (PDEs) into a broader problem solving environment. PDE problems are very important in scientific computing because they provide useful models of a variety of physical phenomena (e.g., heat distribution, electrical and magnetic potential, fluid flow). Recent decades have witnessed great progress in this branch of scientific computing, culminating in general PDE problem solving systems such as ELLPACK [8] and DEQSOL [5]. These systems are highly functional, convenient and complete, but are typically focused upon solving one single, isolated PDE problem (e.g., the temperature distribution in a single heated metal plate).

RELAX first encapsulates the functionality of these systems – rather than re-inventing or duplicating their technologies, RELAX focuses on integrating them into a larger composite problem. Adapting these systems for RELAX requires a relatively small amount of coding (compared to re-implementation). Once encapsulated, RELAX extends the functionality of these systems – whereas they solve single domain PDE problems, RELAX solves multi-domain "systems" of PDE problems (e.g., the temperature distribution in a system of attached metal plates). In meeting this goal, RELAX makes use of pre-existing PDE problem solving software without re-implementing it, and with a relatively small degree of embellishment.

The second design goal of RELAX is that it have the capability of using pre-existing display and interaction components to form a flexible, dynamic
user interface. This goal is motivated by the fact that user interface software, like PDE modelling software, is extremely expensive to implement [7]. RELAX supports user interface components which are structurally separated from the underlying application feature which they are displaying or editing. Such user interface components do not share data structures with the application, instead they operate by passing messages to and from the application. Separation of the application from the user interface is widely recognized as an important goal of user interface design [9].

The third design goal is that the first two design goals be achieved in a uniform, seamless fashion. Consequently, RELAX employs a single internal operational paradigm – message passing. This paradigm is applied to all functions of the system. Message passing is used to operate both the RELAX user interface environment as well as the RELAX computational engine.

Whereas RELAX is an interactive system, its application domain has some characteristics which are unusual in interactive systems. Many of the computational steps (e.g., solving a PDE or displaying a color surface) are inherently time-consuming; there may be dozens or hundreds of such steps in solving a modestly complex problem. Thus, really "fast" response is unlikely even with very high performance computers. There are, however, single editing steps where the user expects very fast response. Even so, the overall size of the computational problem makes it feasible to use rather complex data structures and protocols without impacting system performance. The highest priority is to greatly enhance the problem solving capabilities and user convenience.

1.1 Layout of this Paper

This paper describes the architecture of the RELAX environment, and the mechanisms which RELAX uses to meet the above design goals. The remainder of this introductory section defines the class of mathematical problems which RELAX may be applied to. Section 2 presents the basic architectural arrangement of RELAX and describes the message passing mechanism of the system. Sections 3, 4 and 5 describe the operations of each part of the RELAX system. Section 6 summarizes the ways in which RELAX meets its design goals, and suggests directions for future work.

1.2 Composite PDE Problems

RELAX is used to solve composite PDE problems. These are multi-domain, multi-operator PDE problems which model physical systems involving the interaction of multiple local phenomena. A composite PDE problem is derived from a collection of primitive PDE problems. A primitive PDE problem is comprised of (1) a domain – an open set in space enclosed by a set of interconnecting boundary elements, (2) an interior equation – a partial differential equation which is enforced over the interior of the domain, and (3) a set of boundary equations – individual equations which are independently defined on each boundary element of the domain. Figure 1 illustrates the definition and solution of a primitive PDE problem.

Primitive PDE problems are good for modelling physical objects – the domain typically models the shape of the physical object, the interior equation describes some phenomenon such as the distribution of heat or electro-magnetic potential throughout the object, and the boundary equations dictate how the object interacts with its surrounding environment. RELAX restricts primitive PDE problems to the two-dimensional plane – there are only two space variables in all equations, and the boundary elements are one-dimensional curves.

A composite PDE problem is built from a set of geometrically joined primitive PDE problems. The primitive problems are joined at various intersecting boundary elements, called geometric interfaces. (In RELAX, geometric interfaces are restricted to be line segments.) Additionally, these primitive PDE problems are coupled by special interface equations. Interface equations dictate how the local phenomena are transmitted from one domain to the next. Examples are:

\[ U = V \]
\[ U_n = -V_n \]
Here, the symbols $U$ and $V$ represent the solutions of two primitive PDE problems on neighboring domains. $U_n$ and $V_n$ represent the outwardly pointing normal derivatives along the boundary element where the two domains are joined. These interface conditions specify that the neighboring solutions agree in value along the shared boundary, and that the solutions blend smoothly across the boundary. However, RELAX allows any algebraic combination of $U$, $V$, $U_n$, and $V_n$ to be used as interface equations. Figure 2 illustrates a two-domain composite PDE problem, as well as the local solutions (without interface conditions) and the solution of the composite problem. In general, composite PDE problems can model complex multi-material systems, where each domain represents a single body in a conglomerate system. These systems are typical of mechanical designs, an extreme example of which would be an engine.

2 Architectural Overview

RELAX is structured as a set of message-passing software components. A software component is a collection of data, along with a collection of procedures for operating upon that data. Essentially, components are objects in the terminology of object-oriented computing. However, RELAX is not an object-oriented computing system in that it does not become involved in the generation of objects, and does not provide a class- or inheritance mechanism. Components do not access the internal data of one another. Instead, components interact with each other, and the system, by sending and receiving messages. RELAX provides a special message handler for addressing and routing message transmissions.

2.1 Message Passing in RELAX

RELAX components send two types of messages: edit requests and change notifications. An edit request is a message asking that another component change the state of one of its features. A change notification is a report sent from a component (to no other component in particular) that
the state of one of its features has changed. Essentially, edit requests are point-to-point commands and change notifications are broadcasts.

Editable features are represented by symbolic identifiers called targets. Each target has a name and a type. Target identifiers (integers in our implementation) are generated by the system upon request by a component. This is called registering a target, and the requesting component is called the target owner. Any RELAX component can register a target. Registering a target makes some feature of the owner visible to the other components of the system. Any RELAX component may be a target owner, an owner may register any number of targets, and target registration may be done at any time during the lifetime of the system. Upon registering a target, the owner must immediately be prepared to receive edit request messages for the target. The owner component is "blind" in this form of communication – when it receives an edit request message, it does not know which component transmitted the message.

In addition to receiving edit request messages, a target owner must also generate change notification messages whenever the state of the targeted feature changes. These messages are duplicated and broadcast to any other components which are interested in a target. Other components express their interest in a target by monitoring the target. The system handles the registration of target monitors and automatically distributes change notifications to them. The owner component is again blind in this communication – change notification messages are broadcast without regard to the number or nature of the monitoring components. Target owners may also withdraw their targets at any time. In this case, each target monitor is sent a TARGET_WITHDRAWN message.

2.2 RELAX Components

There are three categories of components in RELAX: (1) primitive PDE problem solving objects ("primitive objects" for short), (2) user interface components, called editors, and (3) intrinsic system components. The first two types of components are supplied by the user and may be dynamically created, attached, and detached from
the system. Figure 3 illustrates the arrangement of the components of the RELAX architecture.

RELAX components are represented as sets of data and procedures. In the current implementation, the data and procedures are all contained in a single address space, and the system runs as a single process. It is possible to operate RELAX in a distributed fashion, by substituting communication stubs in place of various components in the master RELAX process. We have implemented such stubs and the RELAX prototype currently operates in this distributed mode. The externally supplied components, editors and primitive objects, are represented by state records — these are dynamically created data structures which contain private data and a procedure table. The procedure table contains pointers to the procedures used by the editor or primitive object. Messages are passed to components by invoking procedures owned by that component.

We briefly outline the function of each type of component:

- **Primitive Objects**: These are externally supplied components which model and solve primitive PDE problems. Primitive objects are responsible for all aspects of solving a single-domain PDE problem, including the generation of numerical meshes, discretization of the PDE, and solving systems of equations.

- **Editors**: These are externally supplied components which provide an interface between the user and some feature of the system. The editor component is responsible for the complete presentation of the user interface, including all communication with the window system and/or graphics package.

- **Message Dispatcher**: This is a system supplied component which handles all transmission of messages within the system. The message dispatcher also registers targets and can assist editors in locating targets.

- **Composite Problem Platform**: This is a system component which maintains the data structures defining a composite PDE problem. For example, the composite problem platform stores topological information about which primitive objects share geometric interfaces, as well as equations defining the

![Figure 3: Architectural arrangement of RELAX. The elliptic shapes represent components which are supplied externally, by the user or from a library. The rectangular shapes represent system components.](image-url)
interface conditions along those interfaces. Additionally, the composite problem platform maintains data structures defining a global solution iteration, and is capable of executing such iterations. Finally, the composite problem platform is capable of defining composite PDE problem hierarchically.

- **Object Support Platform:** This is a system component responsible for integrating primitive objects into the system. The object support platform provides the attachment point for primitive objects - recall that they are external components and must be dynamically attached to the running system. The object support platform relays messages between primitive objects and the message dispatcher. Another feature of the object support platform is a virtual object mechanism which allows various primitive objects to filter the messages intended for other primitive objects.

- **Editor Support Platform:** This is a system component which provides an attachment point and communication interface for editors. The editor support platform relays messages to and from editors, and is also capable of parameterizing and controlling the message flow, for example, by copying and buffering messages.

### 3 Primitive Objects

Primitive objects are externally supplied RELAX components, each of which manages the affairs of a single primitive PDE problem. This section describes the functions of primitive objects, the external message passing interface of each primitive object, and then gives several examples of primitive objects which we have implemented.

#### 3.1 Functions of Primitive Objects

- **Cloning.** The first function of a primitive object is that it be capable of duplicating itself. This is called cloning, and is the mechanism for object origination in RELAX. RELAX does not have an “object system,” as is found in general purpose object-oriented computing systems [3]. (These systems automatically generate new objects, usually through a class inheritance mechanism). Instead, object classes are represented by “iconic objects” - these are representative objects, loaded into the system at startup, and are used exclusively for cloning. The author of an object class must provide an iconic object for that class.

- **Communication Interface.** Primitive objects must present a communication interface to the outside world. This means they must register targets which represent their major features and then exchange messages for these targets. This makes the primitive object visible to the other components of the system. For further details about this interface, see Section 3.2.

- **Space Allocation.** A primitive object is responsible for allocating all space necessary for its operation. Since primitive objects solve PDE problems, this often involves significant amounts of memory. A primitive object must communicate directly with the underlying operating system (e.g., with malloc()) to acquire its space.

- **Geometric Interfaces.** A primitive object must distinguish one or more of its boundary elements as geometric interfaces. Geometric interfaces are the only boundary elements along which the primitive object may be joined to others. In the current two-dimensional design of RELAX, geometric interfaces are line segments. The remaining boundaries can be of any shape the object chooses. A primitive object must capable of enforcing arbitrary first order linear boundary equations of form

\[ aU + bU_n = c(x, y) \]

along these interfaces. Here, \( a \) and \( b \) are constant coefficients and \( c \) is variable, taking its value from an evaluation function. RELAX uses these boundary equations to transmit boundary data among neighboring primitive PDE problems during a global solution process.

- **Geometry Management.** A primitive object must manage its own geometry. RELAX does
not specify any shape constraints for primitive objects, relying instead upon objects to define their own geometry. Geometrically, RELAX is concerned only with the special geometric interfaces. Objects may be of any shape, so long as they meet at valid geometric interfaces. Internally, the object is responsible for maintaining a geometric configuration that is consistent with its other functions, for example, the object should check that there are no sharp corners which it cannot handle. Editing messages which request such geometric configurations should be rejected by the primitive object (i.e., a return value of NULL).

- **Mesh Management.** Primitive objects are responsible for solving PDE problems, which usually means that numerical approximation methods are used. In this case, the object must generate a mesh for its domain. This involves internal coordination, for example, the primitive object must check with its internal geometry handler that the shape of the domain can be meshed.

- **Equation Management.** A primitive object must maintain the equations which define the primitive PDE problem – the interior and boundary equations. These are not stored or managed by RELAX. Of course, the primitive object must check the equations for consistency with its internal solvers – if special purpose software is used, then general equations should be rejected by the primitive object.

- **Solver Management.** Each primitive object must provide the capability of solving any primitive PDE problem it has accepted. The primitive object must provide solution data, on demand, for its most recent configuration. There are three operational modes regarding the time of the actual solution. In *default mode*, RELAX places no constraints upon solution time, and internal solver invocation may be deferred until solution data is actually requested. The object may be requested, however, to operate in a *synchronous mode* in which any change in its configuration (for example, a boundary equation is edited) results in immediate re-invocation of the primitive object’s solver. A primitive object must also respond to a one-time synchronization message, called SOLVE, which asks that the local PDE problem be immediately solved. An additional mode, *unsynchronized mode*, asks that the primitive object intentionally wait until solution data is requested or a SOLVE message is sent.

- **Interprocess Communication.** Some primitive objects establish communication links with external processes for carrying out the actual numerical operations of the primitive PDE problem. This is useful for primitive objects which are derived from stand-alone, single domain PDE problem solving systems (e.g., ELLPACK [8]) or for computationally intensive local PDE problems which must run on special hardware. The primitive object is responsible for allocating all process and network resources which are necessary, for example, setting up Unix pipes or TCP/IP connections. RELAX does not automate this process.

### 3.2 The Primitive Object Message Interface

Each primitive object must present a message passing interface to the system through which it can be manipulated and through which its features and configurations can be displayed. The primitive object message passing interface is comprised of a set of standard targets which every primitive object must register, along with a set of standard messages for those targets. This section lists and describes the types of standard primitive object targets, and a few of the important messages for these targets. A complete listing of the targets and messages is found in [6].

- *object*: Each primitive object registers one target of this type. This target is for messages concerning the primitive PDE problem as a whole, rather than one particular feature. For example, the CLONE edit request message is sent to this target of the object when a duplicate object is to be created.

- *scalar*: These targets represent parameters or numerical attributes of the primitive PDE problem. There are many uses for scalar targets – for example, the object can advertise the area of its domain as a scalar target. The value of such scalars are often
needed externally, for example, in parameterizing global solution algorithms. The choice of scalars made visible through this target is left to the object — there are no required scalars. Primitive objects can register as many scalar targets as they need. A scalar target may be sent the \texttt{SET\_SCALAR\_VALUE} and \texttt{GET\_SCALAR\_VALUE} edit request messages, and the primitive object must generate the \texttt{SCALAR\_VALUE\_CHANGED} change notification message whenever the attribute which the scalar represents changes.

- "geometric\_interface": These targets represent the boundary elements of the domain, along which the primitive object is prepared to share solution data. Messages can be sent to these targets to request boundary data (\texttt{GET\_BOUNDARY\_VALUES}) or to impose the first order linear boundary conditions described above (\texttt{SET\_BOUNDARY\_CONDITION}). In addition, messages may be sent which request linear transformation operations on the geometric interface (i.e., to move or stretch it). The primitive object is free to react to these operations as it sees fit — rigid objects would move the entire domain as an interface element is moved, others may allow free deformation of the object's boundaries.

- "domain": This target represents the domain of the primitive object. Messages sent to the domain target of an object can request geometric transformations on the domain, or may request data for displaying the domain (e.g., \texttt{SHOW}).

- "boundary": Primitive objects register a boundary target for each independent boundary element enclosing their domain. Messages sent to boundary targets request information about its geometry, and also request the imposition of boundary condition equations on the segment. The geometric interface boundary elements are also represented by a boundary target — the difference is that the primitive object is not required to share boundary data along ordinary boundary elements, and the shape of ordinary boundary elements is unspecified by the system.

- "solution": Each primitive object registers a solution target. This target represents the solution of the primitive PDE problem; i.e. the local mathematical or physical phenomenon being modelled. Query messages (\texttt{GET\_SOLUTION\_VALUE}) may be sent to a solution target to retrieve the solution value at a point. Other query messages (\texttt{SHOW}) ask for display data for the solution (there is a common format for this — see [6]). The most important message, however, is the \texttt{SOLVE} message, asking that the primitive object carry out a local solution invocation. Currently the system supports the registration of only one solution target per primitive object. Future designs of RELAX could support multiple solutions — this would be useful for modelling local phenomena such as conductive-convective heat flow, where there are multiple solutions (flow and temperature) which are internally coupled.

3.3 Examples of Primitive Objects

We have implemented several classes of primitive objects and have supplied them with the RELAX prototype. This section summarizes their features.

- \textit{Helmholtz box objects.} The domains of primitive objects in this class are rectangular. These objects solve the Helmholtz equation,

\[
U_{xx} + U_{yy} + \mu U = f
\]

using a rectilinear mesh and a "5-point star" discretization of the interior equation. The local linear system generated by this specification is solved using a Gauss-Seidel style of iteration. Helmholtz box objects test all geometric transformation requests to ensure the rectangular integrity of the domain is preserved. (Rotation, translation, and axial scaling is permitted.) Additionally, the primitive object accepts non-standard "meshing" messages which allow an editor to set the resolution of the vertical and horizontal meshes.
Each side of the domain is a straight-line segment, and the Helmholtz box registers a boundary and a geometric interface target for each side. Helmholtz box objects also allow boundary equations to be independently set along each side of the rectangular region.

- **Annulus objects.** These are annular-shaped primitive objects which solve a polar version of the Helmholtz equation

\[ U_{rr} + U_{tt} + \mu U = f. \]

Here, \( r \) and \( t \) are the radial and polar space variables of the domain. Boundary conditions may be set along the interior and exterior faces of the annulus. Annulus objects use a radial mesh, discretizing the polar interior equation with a polar form of the "5-point star" finite difference discretization technique. The internal solution method is Gauss-Seidel iteration. To preserve its symmetric nature, the annulus object disallows all linear transformations on its domain, except for translation, rotation, and axisymmetric scaling.

- **Quadrilateral objects.** These primitive objects have arbitrary quadrilateral regions as their domain, and can solve a general second order linear partial differential equation on their interior:

\[
\begin{align*}
  a(x, y)U_{xx} &+ b(x, y)U_{xy} + c(x, y)U_{yy} + \\
  d(x, y)U_x &+ e(x, y)U_y + f(x, y)U = g.
\end{align*}
\]

Currently these coefficients are restricted to be constants. The underlying solution technology is from the ELLPACK [8] system. The mesh is rectangular, and the discretization is interior collocation with Hermite bi-cubic basis functions. The object inspects all linear transformations for integrity (no self intersection allowed), and also checks with the ELLPACK domain processor to ensure that the domain may be meshed (i.e., no excessively sharp corners). Because of the implementation of the ELLPACK system, each quadrilateral must run as an independent process, so that the actual code for the quadrilateral is merely stub code for performing interprocess communication. Currently this communication takes place over Unix pipes, and both the RELAX process and the quadrilateral process must run on the same machine.

- **Joint objects.** The domains of joints are subtended by a collection of boundary elements which are alternately line segments and cubic splines. The line segments form the externally accessible geometric interfaces of the joint – other objects may be attached here. The cubic splines interpolate the endpoints of the line segments, and are perpendicular to them. Because of their perpendicular fashion of meeting the line segment, these objects are useful as joining regions in physical models. The solver is similar to the quadrilaterals – it is capable of solving general second order linear elliptic operators, and a separate ELLPACK-generated solver process is used.

## 4 System Components

This section describes the components of the RELAX environment which are provided by the system. Although they are intrinsic, and not user-contributable, the system components operate in a highly independent fashion, and communicate through the same message passing mechanism as used by primitive objects and editors. System components do not share data. The RELAX design is free of intrinsic data dependencies. For this reason, the RELAX system itself can be easily extended and modified.

### 4.1 The Message Dispatcher

This component provides the central message passing mechanism of the RELAX environment. The message dispatcher handles creation and registration of targets, transmission of edit requests to target owners, registration of target monitors, and dispersal of change notification messages from the target owner to all target monitors. The message dispatcher operates independently of the other system components, and does not assume any particular number, names, or location of other system components. It does this by using **callback procedures** for all message handling. For example, when an owner component registers a target with the message dispatcher, it must supply the address of one of its procedures to be used as a callback procedure.
When an edit request message is sent to the target, the owner's callback procedure is invoked and the message body is passed as an argument to the callback procedure. Similarly, when components wish to monitor a target they must provide a callback procedure for delivery of their copy of all change notification messages.

The message dispatcher also assists RELAX components with target navigation. In general, an instantiation of the RELAX computing environment can involve hundreds or thousands of targets (each boundary element of each domain is a target, etc.). Components must have some mechanism for identifying targets in the first place. This is handled, at a global level, by providing database-style navigation operations which allow any component to request that the message dispatcher search the global space of targets for one or more targets matching a given pattern. The search keys upon the name and type of a target.

The message dispatcher handles this task in an interesting way. Rather than providing special purpose subroutine calls for performing navigation operations, the message dispatcher creates a special navigation target for this purpose. A navigation target is a target representing targets. Messages sent to this target request the invocation of navigation operations. The bootstrapping problem (how do I get the identifier of this target) is handled by distinguishing a special identifier, which is given the value of the symbolic constant MASTER-TARGET, defined in system header files. Some of the edit request messages which the master navigation target accepts are GET.TARGETS.OF.TYPE, GET.ALL.TARGETS, and GET.SPECIFIC.TARGET. Components may also monitor the master target - change notification messages TARGET.REGISTERED and TARGET.WITHDRAWN are dispersed whenever a new target is created or an existing target is destroyed.

4.2 The Object Support Platform

The object support platform is an intrinsic system component which is dedicated to integrating primitive objects into the RELAX system. There are three major functions of the object support platform in fulfilling this role. The first function is to provide low level object integration. The object support platform registers all of the standard primitive object targets with the message dispatcher. The object support platform also records the address of the primitive object's state record, and uses this information to relay all messages flowing between the primitive object and other system components. The object support platform provides primitive objects with the following communication abilities: (1) primitive objects may receive edit requests for their targets, (2) primitive objects may generate change notification messages for their targets, (3) primitive objects may send edit requests to any target in the environment, (4) primitive objects may monitor any target in the environment. Thus, primitive objects are provided with all communication capabilities of other RELAX components. However, all communication with primitive objects flows through the object support platform.

The second function of the object support platform is to provide navigation operations for locating primitive objects. These navigation operations are similar to the global navigation operations of the message dispatcher – the object support platform maintains a database of information about the primitive objects available in the system, and other components can send query messages to locate specific primitive objects. The object support platform, at system startup time, automatically registers a navigation target which represents the set of primitive objects. Samples of valid query messages are: GET.ALL.OBJECTS, returning a (long) list of object identifiers, GET.OBJECTS.OF.TYPE, returning a list of objects of a particular type (e.g., helmholtz.box). This navigation target can also answer queries about the targets of an object. Sample messages in this category are: GET.TARGETS.OF.OBJECT and GET.TARGETS.OF.OBJECT.OF.TYPE. The object support platform also generates change notification messages for this target – for example, OBJECT.REGISTERED, OBJECT.DELETED, TARGET.REGISTERED, and TARGET.DELETED.

The third function of this platform is to provide a virtual object mechanism. This is a message passing mechanism which allows one object (a virtual object) to intercept and/or filter the messages sent to another object. This mechanism is made possible by the architectural placement of the object support platform – all messages for targets of primitive objects must first flow through it.
The virtual object mechanism has many uses. For example, it can be used to impose artificial constraints on the behaviour of another object. By first sending all "linear transformation" messages to a virtual object, the virtual object can impose geometric constraints (e.g., snapping, gridding, frozen position) on the original object. Another important use of a virtual object is as an adaptor. These are virtual objects which filter and pre-process equations and other important configuration parameters. By using an adaptor, existing primitive objects do not need to be re-implemented or modified as the system evolves.

The virtual object mechanism is rather complex, and is more completely described in [6]. One of the important cases which is handled by the virtual object mechanism is the propagation of messages. In the case that a virtual object is itself overridden by another virtual object (e.g., an adaptor of an adaptor, etc.) a message might be passed to many intermediate virtual objects before it reaches a terminus. The propagation feature of the virtual object mechanism allows virtual objects to "preprocess" intercepted messages before they are themselves intercepted by other virtual objects.

4.3 The Composite Problem Platform

The composite problem platform is a system component which is responsible for defining and solving composite PDE problems. There are three major functions in this regard – we describe them in this section. First, the composite problem platform must define a topology for the composite PDE problem. This is information about the geometric and equational coupling of primitive problems. The second function of the composite problem platform is to provide data structures and mechanisms for solving composite PDE problems. Solution algorithms are not automatically generated by RELAX – instead, solution algorithms are editable features, and are represented by targets. The third function is to provide mechanisms for hierarchically defining complex composite PDE problems. This means that a complete composite PDE problem may be treated as a single primitive in a larger composite PDE problem.

The composite problem platform is an intrinsic system component, but it is structurally separated from the others (see Figure 3). All interaction with the composite problem platform is done by passing messages through the message dispatcher. The composite problem platform represents all of its features and services as targets, just as primitive objects represented their features through targets. Again, communication throughout RELAX is done in a uniform fashion, by passing messages to targets. This section describes the targets of the composite problem platform and some of the messages for those targets.

Separating the composite problem platform from the other RELAX components has an important advantage. Should the design of the system be extended to encompass broader definitions of composite PDE problems, or their solution algorithms, only the composite problem platform would require re-implementation. The other RELAX components would operate through the same message passing protocol.

4.3.1 Composite Problem Topology

The first major function of the composite problem platform is to define the topology of composite PDE problems. Topological information is embodied in abstractions called adjacencies. An adjacency is a data record that contains information about the coupling of two primitive PDE problems. A set of primitive PDE problems, along with a set of adjacencies which couple them, completely defines a composite PDE problem (Section 1.2).

Each adjacency identifies two primitive PDE problems, along with two specific geometric interfaces of those primitive problems. These geometric interfaces are symbolically coupled by the adjacency. Adjacencies record two types of coupling information. First, adjacencies record geometric information, which specifies exactly how the geometric interfaces are aligned. In the current design of RELAX, all geometric interfaces are line segments in the two dimensional plane, and they are aligned by identifying two representative points on the coupled geometric interfaces (the line segments may not perfectly coincide, numerically). The second type of information is equational. Each adjacency records two inter-
face equations to be imposed along the portions of the two geometric interfaces which are coincident (This is only a specification of the interface equations – the actual enforcement of the interface equations is the responsibility of the global solution algorithm – see Section 4.3.2).

The composite problem platform makes its topological information globally viewable and editable by creating several types of targets. First, the composite problem platform creates an adjacency target each time a new adjacency is created. This target is used to externally edit and inspect the corresponding adjacency. Sample edit request messages for adjacency targets are: SET.ALIGNMENT, SET_INTERFACE.EQUATIONS, GET_ALIGNMENT and GET_INTERFACE.EQUATIONS.

The composite problem platform also creates a navigation target for the set of adjacencies. As with the other navigation targets, edit request messages sent to this target make database-style queries about the global set of adjacencies, and change notification messages are generated whenever a new adjacency is created or an existing one is deleted. Messages sent to this target can also request the creation of new adjacencies and the deletion of existing ones. Sample messages in this category are CREATE.ADJACENCY, DELETE.ADJACENCY, and CHECK.ADJACENCY. The last message asks the composite problem platform to check a potential adjacency for validity (are the geometric interfaces close enough to be coupled, etc.).

The composite problem platform also monitors targets. In particular, the geometric.interface targets coupled by an adjacency are monitored. If one of the geometric interfaces is deleted (for example, a primitive object decides to no longer share data along the interface and withdraws it as a globally visible target), or if one of the geometric interfaces is simply moved (breaking the geometric bonds of the adjacency), then the composite problem platform automatically withdraws or adjusts the adjacency. Notice that this action results in the generation of change notification messages for the adjacency target itself.

### 4.3.2 Execution of Composite PDE Problems

The second major function of this platform is to create and maintain data structures which define a method of solving composite PDE problems. The method used by the current design of RELAX is iterative interface relaxation. This method is formally specified in [6], and examples of its use are found in [2]. In a nutshell, this iteration consists of (1) visiting a sequence of primitive PDE problems in the composite PDE problem, (2) importing slope and boundary data from each neighboring primitive PDE problem (coupled to a geometric interface of the current primitive), then (3) solving the local primitive PDE problem using the imported data in boundary conditions along the coupled interfaces. The philosophy of this iteration follows that of all iterative algorithms – the solution invocation locally improves the estimate of a global problem, and the improved data is then incorporated into other local solutions. A convergence argument is beyond the scope of this paper and, in general, convergence is not guaranteed by RELAX. RELAX is only a programming system for specifying iterations (i.e., RELAX is no more responsible for convergence than is FORTRAN). The key programmatic elements of interface relaxation iterations are: (1) a schedule which dictates the sequence of primitive PDE problems to visit, and (2) a set of equations which describe the importation of boundary data along each coupled geometric interface.

This programmatic information is embodied in structures called solution scripts. A solution script contains three types of information:

- First, a solution script contains scalar quantities called script variables which may appear in various equations used by the solution script. The value of a script variable can be set in one of three ways: (1) it can be set explicitly, (2) it can take its value from attributes of specific geometric interfaces, or (3) it can take its value from scalar targets of objects (see Section 3.2).

- The second type of information in a solution script is an iteration schedule. This is a sequence of primitive objects to repeatedly visit during the iteration. The default iteration schedule is a random round-robin listing of primitive objects in a composite PDE prob-
• The third type of information is a set of smoother equations. These are pairs of equations assigned to adjacencies among primitive objects. (The assigned adjacencies must couple geometric interfaces of primitive objects listed in the iteration schedule.) An example smoother equation is

\[ U + aUn = V - aVn. \]

Funaro et al. [21] use pairs of smoother equations such as:

\[ V = U \]

\[ Un = \theta Un + (1.0 - \theta)Vn \]

Here, \( \theta \) is a parameter used to throttle the convergence (it is also related to the geometry of the domains). Values of iteration parameters such as \( \theta \) can be taken from script variables.

Script variables can also be used on a per-domain basis. For example, in the smoother equation

\[ U + 0.5 \times \text{width}(U) \times Un = \text{width}(V) \times (V + Vn) / 2 \]

the value of the width scalar is derived from the primitive objects representing local solutions \( U \) and \( V \). Since the values of script variables can be taken from local PDE problems, and since script variables can appear in smoother equations, the interface relaxation driver can be easily regulated by local characteristics.

The composite problem platform registers a target for each solution script, and accepts messages which allow the script to be inspected (e.g., GET_SCRIPT_INFORMATION), as well as edited (SET_SCRIPT_VARIABLE, SET_ITERATION_SCHEDULE, SET_SMOOTHER_EQUATION, etc.). Change notification messages are also generated whenever some feature of the script changes. Additionally, a navigation target is created for the global set of solution scripts.

The composite problem platform monitors primitive object and adjacency targets to ensure the consistency of solution scripts. When an adjacency is deleted, for instance, the solution script is adjusted to remove the assignment of smoother equations to the adjacency.

4.3.3 Hierarchical Design of Composite PDE Problems

The third major function of the composite problem platform is to provide mechanisms for hierarchically structuring composite PDE problems. Hierarchical problem building is necessary because even ordinary problems can involve hundreds or thousands of domains, adjacencies, etc. – editing them would be an unwieldy task. The basic notion behind hierarchical problem building in RELAX is to encapsulate a completely defined and executable composite PDE problem, treat it as if it were a primitive PDE problem, and to include that primitive PDE problem in a larger composite PDE problem.

The composite problem platform represents encapsulated composite PDE problems by structures called assemblies. An assembly contains

• a list of primitive PDE problems called the members of the assembly,
• a set of distinguished geometric interfaces of members of the assembly, called the external interfaces of the assembly, and
• a solution script assigned to the assembly.

The external interfaces are used to share boundary data between the encapsulated assembly and any other primitive PDE problems to which the assembly may become coupled. The solution script is used to solve the composite PDE problem defined over the assembly – this solution script is used as the "local" solution method of the assembly.

This representation is hierarchical – assemblies can contain assemblies which can contain assemblies, etc. The composite problem platform represents these containment relationships as an assembly tree. Each node in the assembly tree represents an assembly – the children of this node represent assemblies contained in the parent assembly. The root of the assembly tree is called the root assembly. When new primitive objects are created, they are made members of the root assembly. Thereafter, the containment relationships among assemblies and primitive objects may be altered with assembly operations.
4.4 The Editor Support Platform

The editor support platform is an intrinsic system component that provides an interface between editors and the rest of the system (specifically between editors and the message dispatcher). The architectural location of the editor support platform is illustrated in Figure 3. The editor support platform fulfills three primary functions: (1) it provides basic operational support for editors, (2) it provides delivery of all messages to and from editors, and (3) it provides higher level "environmental" operations.

The first function of the editor support platform is to provide basic low level integration of editors into the overall RELAX environment. The editor support platform maintains the data structures which link editors to the system, for example, storing the address of editor state records. The editor support platform provides several conveniences to editors, for example, automatically allocating X window system [4) resources such as display connections, an application context structure [1], and a set of initial colors (indices of colormap entries) for displaying solution plots. The editor support platform also supplies a navigation target which allows editors to obtain lists of editors attached to the system.

The editor support platform is responsible for handling the relay of all messages to and from editors. Through this mechanism, editors may send edit requests and monitor targets – editors do not register targets themselves (although each is represented by a special system defined "focus" target).

The editor support platform also provides several mechanisms that yield a high level "editing envi-
Editors are externally supplied, dynamically attached interaction components which display

...environment." The direct access to the message dispatcher provided above is a low level form of interaction. The editor support platform supports three forms of high level editing operations:

- **Selections.** A selection is a named list of target identifiers. This list is stored in the editor support platform, and can be accessed by any editor. Selections may also be created and deleted at any time. This provides a limited form of inter-editor communication, although it cannot be used for generalized exchanges of data, like the cut-and-paste style of window system selection mechanisms [⁴]. Instead, a selection is a mechanism by which editors can operate as visual "choose" interactors. Such editors present the user with a display of items, such as the domains of a set of primitive objects, and allow the user to interactively choose certain of the items. The targets representing this list of items are placed into a selection, and read by another editor, which most likely invokes an operation on the selected items (e.g., set the interior equations of all "chose" objects to \( U_{xx} + U_{yy} = -1.0 \)). This allows specialized editors to collaboratively interact in solving problems.

- **Editor Focus.** The focus of an editor is the set of targets it is interested in. This term is intentionally left vague, and its interpretation is left to the editor. RELAX allows editors to alter the focus of other editors. This is rather helpful in navigating through the space of available targets. Generic "navigation editors" can be built which allow the user to first select the target to edit. Then, the focus of a "special purpose" editor can be set to the desired target. This frees the editor programmer from coding navigation software.

RELAX handles focus editing in an interesting fashion. Editors are only loosely coupled to their focus list. Instead of requiring an editor to create and maintain a focus list, the editor support platform automates this for each editor. First, it creates a special "focus target" for each editor. Then, it creates and maintains a focus list corresponding to the focus target. Edit request messages can be sent to the focus target to inspect or edit the focus list (the messages are GET_FOCUS and SET_FOCUS, respectively. The editor support platform also generates change notifications whenever the focus list is changed (FOCUS_CHANGED). If editors are interested in their focus lists, they may manipulate as well as monitor their own focus target. Of course, external editors can reset the focus list as well, and the editor will be kept informed when other editors (e.g., navigation editors) wish to alter their focus. This method also permits editors to refuse external manipulation of their focus. They simply do not monitor their focus target. The focusing protocol is completely voluntary.

- **Attachments.** The editor support platform can be asked to regulate the flow of change notification messages being sent to editors. This is necessary in some cases, when highly interactive operations (e.g., linear transformations) cause a flood of change notifications from one (or possibly many) targets.

This regulation is done through mechanisms called attachments. An attachment is a logical connection between a set of editors and a set of targets. The parameters of the attachment dictate what happens when a change notification from any of the targets is to be sent to any of the editors.

An attachment can specify that change notification messages for an editor be buffered. In this case, the attachment specifies (1) conditions under which a change notification message is accepted (i.e., added to the buffer), and (2) conditions under which a buffered message is forwarded to the editor. By default, all change notifications are accepted from any target the editor is monitoring, and messages are forwarded immediately. In addition to the automatic message delivery through these buffers, the object support platform also provides operations which allow editors to explicitly extract change notifications from the buffer (much like event extraction routines in window systems [⁴]).

5 Editors

Editors are externally supplied, dynamically attached interaction components which display
and alter the features of other RELAX components. RELAX separates the operation of editors from the underlying application though its message-target communication model. RELAX is not a closed system, operating through intrinsic system-supplied user interface components. Instead, RELAX editors are user contributed.

As a convenience, we have constructed several editors which may be used with RELAX (see Figure 4). These editors each serve special purposes but are collectively used to construct a composite PDE problem. For example, the editor at the top of the screen is dedicated to textual interactions – it is being used to set the interior operator of a "joint" object (see Section 3.3) to $U_{xx} + U_{yy} = 0.0$. This editor monitors the target representing the interior operator of the joint. If the operator is changed through other means (e.g., by another text editor), a change notification is generated by the joint, replicated and sent to the editor, and the editor updates its internal data structures and display.

The "composite PDE editor" in Figure 4 is a complex editor which simultaneously displays several objects. This editor is capable of displaying the level contours of the interior solutions on those objects as well. At the bottom of this editor is a set of iconic objects – displaying the choices of primitive objects from which composite PDE problems may be constructed.

The "general object viewport" is a display-only editor which can display the local solution from a single object. It handles two dimensional domains of any shape. This editor can also "freeze" its display, serving as a snapshot device for looking at local solutions.

The "composer" editor seen at the bottom right of Figure 4 can be used for geometrically manipulating the domains of individual primitive objects, forming adjacencies among their geometric interfaces. This editor is also capable of grouping primitive objects, performing geometric operations on several of them at once (a common feature in general purpose geometric editors).

Figure 5 illustrates another special purpose RELAX-compatible editor. This one is used to edit targets associated with adjacencies among geometric interfaces. The top part of the editor displays (and edits) the interface equations assigned to an adjacency. The bottom part handles the smoother equations assigned to that adjacency. The central portion of the editor describes some automatic "smoother-derivation" operations of the editor. For example, the interior linearization operation performs a simple discretization of the interface equations at interior points "projected" inside the neighboring domains of the adjacency. These equations are then used as smoother equations for the adjacency (more details of this feature are found in [6]). The other methods base smoother derivations on a "Newton iteration" scheme and upon a user defined linearization (i.e., derive the coefficients of the smoother equations from the coefficients of the interface equations based upon a user-definable linear transformation of their coefficients). Only the first method has been implemented at this time.

Figure 6 illustrates the use of a "focus editor" – it is used to adjust the focus of the top equation editor (GeneralEquationEditor1). This editor also can be used to parameterize other interactions between the top equation editor and the system – for example, freezing any change notifications coming from the system (labelled "equation update capabilities"). This prevents the system from changing the equation stored by the editor. By attaching to one equation, freezing the update capability, refocusing on another equation, and transmitting the frozen equation in a SET.EQUATION edit request message, the original equation is effectively transferred from one target to another, without being overwritten.

6 Summary and Future Work

We have constructed a prototype of the RELAX design. This prototype supplies all of the primitive objects described in Section 3, all of the editors seen in Section 5, as well as several more. The prototype is implemented in C, Fortran, and the ELLPACK source language. The current (August 1992) implementation of the prototype has approximately 30,000 lines of source code. Each primitive object has approximately 3,000 lines of code. The amount of code for editors can vary dramatically, depending upon the extent of their
functionality. The "composer" editor seen in Figure 4 has approximately 6,000 source lines, while the "interface editor" seen in Figure 5 has only 700 lines.

There are several directions for future work. A few are implementation matters – for example, we plan to extend the set of primitive objects to handle more shapes (e.g., curved bodies), local phenomena (e.g., heat convection), and solution methods (e.g., finite elements). There are other, more strategic improvements needed in the design. The design is currently oriented toward a single-process model, which means that all communication is synchronous. This will soon be extended to an asynchronous model in which components can be distributed among multiple processes and can even run on different machines. The composite problem platform could be extended to handle more general geometric interfaces (currently they are always line segments) as well as more complex iterations (currently the solution script is a single sequential schedule). Also, more research needs to be done about how to handle general interface conditions (we have primarily focused on the simple case of $U = V, \quad U_n = V_n$). Ultimately, the system will be extended to handle three dimensional problems.
Figure 4: The RELAX multi-editor environment. A typical RELAX screen showing the user interfaces of some of the editors built to test and use the prototype system.
Interface Editor

- U: solution of HEAT on box1
- V: solution of HEAT on joint2

Interface Conditions:

\[ U = V \]

\[ + U_n = - V_n \]

Symbolic Discretization Method:

Smoothing Functions for Iteration Process:

Set U, Un by: \[ U + 0.5U_n = V - 0.5V_n \]

Set V, Vn by: \[ V - 0.5V_n = U + 0.5U_n \]

Figure 5: The user interface of an editor modifying the parameters of an adjacency. The adjacency is assigned to the intersection of geometric interfaces of primitive objects called box1 (a rectangular region) and joint2 (a curved region).
Intelligence of a focus editor operating on the focus of a feature function editor.

Figure 6: Illustration of the operation of a focus editor. The dialog window at the right is the user interface of a focus editor.
References


