1991

RELAX: A Software Platform for PDE Interface Relaxation Methods

Scott McFaddin

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

Report Number:
91-018
RELAX: A SOFTWARE PLATFORM FOR PDE INTERFACE RELAXATION METHODS

Scott McFaddin
John R. Rice

CSD-TR-91-018
March 1991
RELAX: A SOFTWARE PLATFORM FOR PDE INTERFACE RELAXATION METHODS

Scott McFaddin*  John R. Rice†

Abstract

We describe the design of the RELAX system for programming interface relaxation techniques for partial differential equations (PDEs). The PDEs are solved over composite domains. Each domain is encapsulated with its own geometric modeling system and local PDE solvers. This encapsulation presents a challenging software problem, for it necessitates PDE algorithms which speak a language of "data interchange among objects", rather than the usual language of grid points and linear systems. RELAX is geared towards rapid prototyping, allowing the user to sketch the composite domains on the screen and then to use this sketch to demonstrate the appropriate relaxation schedule visually. The editing of the relaxation program is closely intertwined with the editing of the domain sketch.

1 Introduction

RELAX is a system designed for solving partial differential equations on composite domains. RELAX is especially suited for modeling physical phenomena of mechanical designs, such as heat flows and electromagnetic fields. Many mechanical designs are described in terms of an aggregation of parts brought together in various ways - chain saws have their teeth, turbines have their blades, and spinal columns have their vertabrae. As ubiquitous as the notion of composite design may be in the physical world, it is not as widely reflected in mathematical algorithms for solving partial differential equations, nor in the body of existing PDE software. The design of RELAX flows from ideas in several areas. We distinguish the assumptions made from each area.

Geometric Modeling: The goal of geometric modeling is to accurately represent the space occupied by physical objects. There are many approaches to these representations - constructive solid geometry (CSG) modelers [Brown 82], for example, assume objects are formed from unions, intersections, and differences among a small set of primitive shapes. Boundary modelers, on the other hand, represent an object by a collection of curves-and-surfaces which enclose it. Some allow curved boundaries [Alphal 88], others polyhedral ones [Vanacek 89]. These modeling assumptions are based upon tradoffs between many factors - for example, execution efficiency, representational power, human factors, and assumptions about the physical world. In general, mechanical objects are not built from any one single flavor of part - they are formed from a heterogeneous set of parts, and it is unlikely that any one modeling scheme would be sufficient. What is needed is a system which can make use of multiple geometric models in solving problems.

Assumption A1: Practical applications require multiple modelers.

Although RELAX is not a geometric modeling system per se, it does require input from geometric modelers, namely in drawing and answering
queries about the elemental domains which make up the composite domain over which PDEs are solved. The design of RELAX is oriented towards domain-independence, that is, the system tries to make as few assumptions about the PDE domains as possible.

**Object Oriented Programming:** The high cost of software development has, in recent years, focused a great deal of attention on the benefits of object oriented programming [Johns 88, CACM 90]. Although there seems to be no good definition of exactly what is meant by object oriented computing, we focus on the issue of modularity:

**Assumption A2:** Encapsulated objects are a good vehicle for combining multiple geometric models.

We assume that as much individual domain information as possible is put in a single place. This means that geometric modelers, which create domains, come bundled together with PDE solvers which solve problems on those domains. Along with the advantages in software costs of object oriented structuring, this also induces an appropriate factorization of expertise, for the persons creating the geometric modeler can also create the PDE solvers which operate on that class of model. Most importantly, this means that the overall RELAX system itself does not have to become as complex as $N^2$ geometric modeling systems to use $N$ different types of domains. Note that this requires interaction between geometric domains to be cast in terms of object-object data transmission.

**Numerical Methods for solving PDEs:** A great deal of attention has been paid to solving PDEs on single, and often simple, domains. There has recently been more and more work, however, on the case of composite domains [Funaro 88, Lynch Rice 89]: We contrast three general numerical methods for solving problems on composite domains. (These are formulated for linear, steady-state problems but they may be extended easily for non-linear, dynamic problems.)

- **Method M1:** Place a grid over the whole object. Discretize all equations, interior, boundary, or interface, at the grid points. Use a general linear system solver to get a solution. This is the most direct way to approach the problem. However, this method presents difficulties when the composite objects do not meet up in a regular fashion. More importantly, the global discretizer may not understand the geometry of the sub-parts.

- **Method M2:** Have a grid generator and discretizer for each domain. These modules will supply the global system with linear equations. The global system will generate equations for the interface conditions. Solve the resulting linear system. This avoids the global discretization problems of the first method but it still present difficulties with discretizing the interfaces. Also, it must still solve a large linear system with an unknown structure.

- **Method M3:** Assume each component domain comes equipped with an individual, local solver. Solve the whole problem by successively and repeatedly asking each sub-domain to "solve itself" with varying boundary conditions. Numerical methods are known which operate like this [Funaro 88, Lynch Rice 89]. The central problem here is scheduling the invocation of the various solution modules.

RELAX is a system for taking the third approach.

**Assumption A3:** Multi-domain problems in PDEs can be modeled, mathematically, in terms of relaxations at domain interfaces.

**Visual Programming:** The arguments for visual programming are simple and compelling: Users wants to see what they are computing about.

**Assumption A4:** Programs for composite objects should be written using as much visual specification as possible.

This goal is surprisingly elusive. It is quite natural to build composite domains on the screen: the user
chooses a constituent domain from a "corral", then
drags it to a spot on the composite domain being
assembled. But then to begin specifying iterations
— essentially programming — is difficult to do in a
visual fashion. Among other things, the structure
of the iteration program is hard to see from a direct
animation of the steps involved. We discuss some
of our solutions to these problems below.

2 System Tour.

We have built a prototype of the RELAX de-
design which works for one- and two-dimensional
domains. In this section we demonstrate how this
prototype may be used to construct a composite
PDE problem and then to specify a relaxation.

The user programs PDE computations on as-
semblies of composite objects by building an ex-
ample assembly and then demonstrating computa-
tions on this example. The system watches the
user's actions and builds a program from them.
This program is not just a typescript or event his-
tory of user actions. It is a general purpose exe-
cution plan which may be applied to other assem-
bles. In this section we will show how a simple
relaxation program can be written for an assem-
bly which represents the heat dissipation in a "two
dimensional" engine block.

When the user invokes the system, the interface
in Figure 1 appears. The options window at the
top controls minor features of the interface. The
interface uses a spectrum of colors to represent the
range of values in PDE solutions. The scale bounds
window at the top left gives the upper and lower
bounds of this range and allows these bounds to be
frozen or edited. The mouse window below that
tells the user what the mouse clicks mean at var-
ious phases in the operation. The programming
window is more important: it allows the system
to build the relaxation into a program. The navi-
gation window will allow the program to symboli-
cally traverse the shared boundaries of composite
domains. The large window in the middle is the
sketch window, where composite domains will be
drawn.

The icons at the bottom of the screen repre-
sent the parts from which the user may construct composite domains. There is nothing particularly canonical about these parts – the choice was quite ad-hoc. This reflects a major design principle of this system – it is independent of the particular domains involved and allows new objects to be easily added. The current prototype includes:

- **Helmholtz Box** Geometrically these are rectangular regions which come equipped with special solvers for the two dimensional Helmholtz equation. These solvers allow mixed boundary conditions. These objects only understand the Helmholtz equation.

- **Grommeted Weld** These objects attach a grommet to a Helmholtz box or a polygon. A grommet is basically a symbolic attachment device – geometrically it is only a point in space. However, the weld has the functionality of taking solution data (think heat) and spreading it along the boundary to which it is attached. A weld does not solve any differential equations although it does solve algebraic equations for the spreading.

- **Strut** A Strut is a one dimensional object with two end grommets. It has a one dimensional ordinary differential equation (ODE) solver and uses boundary values from the grommets at each end.

- **Grommeted Beams** This is a one dimensional object which has multiple loci for making grommet attachments. The idea behind this part is to simulate skeletal objects. It has a complicated ODE solver which must solve interior value problems.

- **Regular Polygons** These are similar to Helmholtz Boxes in that they model two dimensional regions. These parts allow multifaceted objects to be simulated, such as gears and sprockets.

Each of these objects comes encapsulated with its own solvers, drawing routines, mouse query routines, etc. The global system draws nothing.

It solves no equations. Instead it sends messages to objects, asking them perform these operations on themselves. The part objects also have their own editors. When a Helmholtz box is created, for example, it allows the user to invoke an editor tailored to the Helmholtz box. An example of this is seen in Figure 2. This editor allows three dimensional viewing of the PDE solution and editing of the specially formatted interior equation.

Complex objects can be built from these primitives. The engine block in Figure 3 was built by creating and placing a series of Helmholtz boxes. When the boxes are placed side by side, they build a data structure for the adjacency connection. These data structures later allow programs to traverse the assembly. Objects obey a simple protocol as they are being created which allow them to query other objects for adjacencies (the details are omitted).

After creating and placing the 19 boxes of the engine block we wish to define a small program for solving the heat equation on the engine block. The strategy in our mind is:

1. Solve the heat equation on box 0 using guesses as boundary values.
2. Solve the heat equation on each of the small
domains using the dirichlet values just computed at the boundary with box0 and heat radiation boundary conditions on the other three sides.

(3) On box0 solve the heat equation again, using derivative values from objects 1-18 on shared portions of the boundaries and radiosity conditions on the unshared portions.

(4) Repeat steps 2 and 3 until convergence.

We begin programming by clicking on box0. We point the mouse at the box and with the left button select it. The interface now appears as in Figure 4. The drawing for box0 is now highlighted (it was sent the show message with highlighted as the argument). Also, box0 has taken over several parts of the interface. The mouse buttons now mean different things, as indicated in the mouse window. The box0 object has also filled in the navigation at the upper right window with several choices.

The navigation window contains the names of command symbols (strings) which may be sent to the object, causing different actions. Command symbols are object defined. Some command symbols cause solution actions to take place. Others, known as navigation symbols, have the object return lists of other objects – usually those which have been joined to it during the construction. If an object does not understand a command symbol, or if it can not return the desired lists, the application of the command symbol is said to fail.

The command symbols for box0 are on display in the navigation window. The first eight command symbols are navigation symbols, and will cause it to return a list of various neighboring objects. The all_neighbors choice will cause box0 to return a list of all its neighbors – in this case boxes 1-18. The last command symbol – solve_HELMHOLTZ – causes Helmholtz boxes to solve PDEs. The system gathers symbolic values as they are selected and place them into the program being constructed.

The idea in our mind is to set up a loop where we solve a PDE on box0 using neumann values from the neighbors, then solve PDEs on all neighboring boxes using dirichlet values from box0. The very first solve on box0 corresponds to step (1) above and will use default values since it has no data from neighbors. We need to do two things, then, with box0 – solve a PDE and then navigate to its neighbors. We will group these actions in a BLOCK construct. We start programming by pressing the begin block button in the programming window. Every action we build until we enact the corresponding end block button will be collected and executed in a sequence.

Having clicked on begin block we are "inside" the block. We click solve_HELMHOLTZ in the navigation window. The system collects this command symbol and enters it as the first item in the block. The program we now have is

BLOCK 1
COMMAND <solve>

The number 1 simply tells how many entries are in the block – it is simply an alternative syntax to parenthesesization. We now wish to move to the other boxes. We click on the all_neighbors button. The system collects this navigation symbol
and invokes it. The reaction to this symbol is to return a list of other objects - box1 - box18. The object box0 is now "unselected" and the system regains control of the interface.

The system notices that more than one object was returned and assumes that in the general case what will follow will be actions to be committed over a set of objects. The system records this as a FORALL construct and enters it as the second item in the above BLOCK. To suggest that one of the neighboring boxes be selected next, the set of neighboring objects is now flashing.

The system assumes the next construct built will be the action taken in the FORALL. If an entire sequence of actions is to be taken, a BLOCK should now be initiated. Since we wish to perform only one action - solve the PDE on the cooling fins - we may immediately demonstrate the action and it will be attached to the FORALL. We select one of the flashing boxes, the same menu of choices appears in the navigation window (since it too is a Helmholtz box), and we select the *solve HELMHOLTZ* command from this menu.

The system records this as

```
FORALL <all_neighbors>
  COMMAND <solve>
```

The system checks to see that the action <solve HELMHOLTZ> can be performed on each object in the FORALL list and then closes the construct. We now close the entire block by clicking on **end block**. The program is

```
BLOCK 2
  COMMAND <solve>
  FORALL <all_neighbors>
    COMMAND <solve>
```

Since we wish this fragment to be repeated (iterated) we once again select box0. Here we can select from among all program fragments built with box0. In this case there is only one such fragment, the BLOCK above, so we may click on **make loop** to wrap a loop around the program fragment. We now have

```
LOOP
  BLOCK 2
    COMMAND <solve>
    FORALL <all_neighbors>
      COMMAND <solve>
```

Notice the theme of bonding between program fragments and parts of the assembly. This is a major theme of the system. We have found this necessary to aid the user in visualizing program actions and in traversing program fragments. This
theme of bonding also leads us to the notion of editing programs by editing their example. Suppose we wish to extend the engine block assembly above to include attachments to a superstructure. This will involve adding new parts to the assembly. This process is taking place in Figures 5 and 6.

We first attach two welds to the central Helmholtz box by selecting the Weld icon and dragging a special welding torch cursor (not seen in the Figures). When the torch gets reasonably close to the Helmholtz box, this is noticed by the Helmholtz box and through the attachment protocol (mentioned earlier), the weld may be attached. The weld and helmholtz boxes build their own data structures for this attachment. They will consult this later to allow traversals during programming and execution. We also create a beam object and place it in the scene. In Figure 6 we have attached struts between the grommets provided by the weld and the grommets in the beam.

At this point we may ask the program fragment above to be reapplied. This time, when the navigation symbol all_neighbors of the FORALL is applied to box0, it will return a set with the two welds mixed in. Everything goes well until the solve command action is invoked on a weld. The weld knows nothing about this and the application of the action has failed. There are two types of failure in this system — structural failure like this, and mathematical failure, which happens when an iteration does not converge or an object has invoked a test action of some sort, such as measuring the size of PDE residuals. Since we have reached a structural failure, the system locates the problem spot in the program and enters insertion mode at that point. The action COMMAND <solve>, which had stood alone as the application of the FORALL structure, will now be replaced by a CHOICE structure — our version of conditional. The welds are now flashing, encouraging the user to select one. The action entered at the weld will be inserted into the CHOICE construct. The program now appears:

```
LOOP
  BLOCK 2
  COMMAND <solve>
  FORALL <all_neighbors>
```

The programmer can also explicitly induce a CHOICE with the [new choice] button.

We conclude this tour by pointing out that what we have contructed here are actual computer programs. The code above applies not only to the example assembly of Figure 6 but to a wide array of similar assemblies. The program does not care about the number of neighbors of the central box, nor which side they are on. In fact, the system does not even care that they are Helmholtz boxes — only that they understand the appropriate command symbol, in this case solve. It is possible, using this model, to make the system care about such things — by programming in commands which have the object check its type, for example. This comes at zero cost to the system, for the response to commands is the private affairs of the object to which they are applied.

### 3 System Design

In this section we will present the current design model of our system. The system is under active development, which means that nothing in this design is sacred. We only give an idea of the design of the current prototype system.

To create computer programs, the user must bear in mind two mental models: a model of the virtual execution machine and a model of the virtual programming machine. The execution machine consists of the body of atomic actions the system can perform and the language used to program those actions. The programming machine is the mechanism by which such programs are written. This dichotomy exists in all programming systems — even when writing ordinary C code one has such models in mind. The execution machine is whatever it is that does things like call procedures, add numbers, clear the screen, traverse pointers, etc.; The programming machine is composed of things like text editors, compilers, and program loaders.
3.1 The virtual execution machine

This subsection will be a brief description of the base language and the primitives in the system. As we remarked above, every programming systems has such a foundation, and those of higher level systems such as ours are especially interesting. Our system deals with object lists and failure actions. An object list is a list of objects returned by some object in the system in response to a command symbol. Each primitive action is given an object list when it is invoked and returns one when it is done. If the primitive action fails, it returns a NULL object list. This might represent, say, a failed attempt at navigation. A program fragment is always given a starting object as the initial object list. In the example program of Section 2 box0 is the starting object.

The system, at present, provides five programming primitives. We describe these below in terms of their return values and failure logic.

- **COMMANDs.** A command is basically a place to store a command symbol (defined in Section 2). Commands appear in the form COMMAND <command symbol> in the saved program. The command, like all primitives, receives an object list to execute upon. The command primitive sends its command symbol to the first object in the object list. The object returns an object list, which the command gives as its return value. Commands may be applied to objects for several purposes: for navigation (left, right, etc.), to invoke solution modules (e.g. solve) and for queries about objects (e.g. test.residual). If the application of the command symbol to the object fails, (such as sending solve to a weld) the command fails too.

- **BLOCKs.** A block stores a list of pointers to other primitives. The input object list is passed to the first object in the object list. The object returns an object list, which the command gives as its return value. Commands may be applied to objects for several purposes: for navigation (left, right, etc.), to invoke solution modules (e.g. solve) and for queries about objects (e.g. test.residual). If the application of the command symbol to the object fails, (such as sending solve to a weld) the command fails too.

- **FORALLs.** A FORALL stores a command symbol and a pointer to another primitive, usually a BLOCK. It receives as input a single object, then applies the command symbol to this object (such as when all.neighbors was sent to box0). It expects to receive a generated object list. A FORALL repeats its action for each object in the generated list. If its action fails for any of the objects, the FORALL fails and further applications stop. The returned object list from the last application is the return value of the FORALL.

- **LOOPs.** are used to indefinitely repeat actions. A LOOP stores a pointer to another primitive, usually a BLOCK. There are two types of LOOP. The first type repeats it action on the same input object list. The outputs from invoking the stored primitive are ignored until failure. These LOOP are used primarily for in-place iterations, i.e. doing the same action repeatedly on the same object until some condition is met. The second type of LOOP passes the output object list from one invocation on to the next invocation. These types of LOOP are used for indefinite walks over assemblies. For example, the code

```plaintext
LOOP
  COMMAND <right>
```

walks to the rightmost neighbor of an object until it can go right no further. When the invocation of the stored primitive on the input object list finally fails, the LOOP returns the last successful return list. If no invocations succeed, the LOOP fails. The stored primitive is usually a BLOCK with a query COMMAND as its first member.

- **CHOICES.** These store a list of alternative primitives. The CHOICE attempts to invoke the primitives on the input list until one succeeds. If all fail, the CHOICE fails.

There is nothing particularly new about these structures. They are found in just about all programming languages. They may be combined,
though, in interesting ways to represent traversals of geometric structures. For example, a fragment such as

```
CHOICE 2
   BLOCK 2
       FORALL <all_neighbors>
       COMMAND <check_type weld>
       ... action A ...
       ... action B ...
```

carries out action A on objects having only welds attached to it and action B otherwise.

The virtual execution machine not only uses object lists in the "blind" fashion above, it can be directed to store and manipulate them directly. The execution machine maintains a stack of object lists which can be extracted and added to. The motivation for adding this feature is that many actions in solving PDEs involves the notion of sweeping data, or wavefronts. Also, there is often the need for "colored" relaxation, such as PDE analogues of the Red-Black Gauss Seidel method for solving linear equations. By explicitly commanding the system to insert objects into an object list, wavefronts and coloring sets can be built and then referenced.

3.2 The virtual programming machine

This describes the way the user arrives at programs written in the primitives above.

3.2.1 Co-Editing Heuristics

We have found the use of menu-based program generation to be very tedious. The program provides a set of heuristics for converting edits on the example assembly into program edits. We list three for example:

- **H1**: Insertion checks. This was done when the weld was added to the helmholtz box above. The system rechecks the applicability of the program to the inserted object. To arrive at the point in the program which applies to the inserted object may require re-navigation of the assembly.

- **H2**: Repeated patterns in example creation lead to repetitions FORALLs in the generated relaxation program.

- **H3**: Attach-Delete-Attach cycles suggest conditionals in the generated relaxation program.

3.2.2 Visualization

The user needs a reasonable way to "see" algorithms. There are three avenues of attack:

1. **Text**. Here, the user is given a textual version of the program he/she has just created. This text can usually be edited. Our system falls into this category because a textual "save" file is generated and can be edited with a text editor. However, because of the heterogeneity of the objects it will not always be possible for an arbitrary user to understand what this text means. Consider the traversal command `<follow 5th_principal_line_of_curvature>`.

2. **Replay**. Here, the user views the program as it executes. The use of action replays are important in leading the user to consider alternative cases, when an traversal fails on a structure. Replay has its drawbacks – the user can see sequential bursts of action but can discern nothing about the way the program is structured.

3. **Geometrically Keyed Grouping**. When an iteration technique is deeply nested, we have found that it quickly becomes difficult to understand the structure of a program. The user can ask the program to run itself in certain modes which display actions on groups of objects all at once. This is reasonable in our paradigm because grouped programming actions, such as those found in block structures, often correspond to geometrically neighboring objects, and it is quite intuitive to the user as to what is going on when they are drawn together. When the user wishes to witness such grouped actions in more detail, the system can "zoom in on" the area of interest.
We have previously alluded to the strong association between program and example in our system. This association is extremely important in allowing the user to visualize algorithm execution.

### 3.2.3 Program Construction Techniques

The system has several non-obvious approaches to building programs:

- **Integration with failure logic.** The system can detect failures in partial program fragments as they are being built. By moving to the failure point and going into an insert mode, the system assists the user in building correct programs by finding the correct place to begin inserting alternative code. Remember that the user may not be able to see or understand exactly where he/she is in the program.

- **Co-editing of program and example.** When new items are inserted into an assembly, the programming mechanism can locate points at which insertions or changes need to be made in a program. Consider the example in Section 2 when the welds were added. This led the system to find a point where new code was needed (in the body of the `FORALL`).

- **Hierarchical grouping of objects.** This allows the user to construct program fragments on different assemblies and then to unite the two. This is not yet implemented.

### 3.2.4 Disambiguating User Actions

The example of Section 2 made the assumption that the user was willing to make movements around the assembly very explicit. However, the user may be facing an attractive display of the assembly and be anxious to simply point at where she/he wants to go. This problem is illustrated in Figure 7. Suppose, for example, that the current action in programming the system is located at box17. To move to a right neighbor of box17, say box21, the user is expected to click the `move_right` button in the navigation window, rather than simply pointing at box21. This

![Figure 7: An ambiguous programming situation.](image-url)
could be incredibly annoying to the user. The system needs some way of allowing the user to point where he wants and then deciding what the user meant in the general case. The system needs to decide to program the move_right choice versus, say, the largest_neighbor choice. This can be a difficult problem. We intend to provide a small user defined language for helping the system to disambiguate user actions. The user will provide a "profile" of himself, written in this language, helping the system to decide what the user means. (It would be interesting if the system could itself be used to write these profile problems. We do not, however, intend to attempt this.)

4 Conclusions and Further Research

This project is under "active development" which means that it is far from being complete. There are many questions that need to be addressed. We list a few in this section.

- We have found that visual programming yields astonishing speed at building example assemblies and specifying simple relaxation schemes. Programs which would require hundreds of lines of, say, FORTRAN/ELLPACK code take only minutes to write visually. Because the end-programmer can see complicated geometric models, he/she need not be concerned with their private specification – programming parameters buried deep inside user manuals are often made obvious on the screen. What the system offers in terms of speed, it seems to lack in power to specify intricate iteration sequences. Only simple cases are easily programmed visually. For this reason we see the system being most suitable as a rapid prototyping tool, rather than a production solution.

- Is there a good mechanism for permuting and refining a visual prototype into a production model? For example, can we convert a RELAX program into a FORTRAN model in order to achieve execution efficiency and detailed control of the computation? Can we compose visual prototypes, to handle complexity well? Can we define and use libraries of relaxation patterns to handle more intricate assemblies?

- The heuristics in RELAX are hardwired. We need to provide a general language for specifying example-edit — program-edit conversions.

- The system could be extended to any application having networks of "local relations" as found in PDEs. We would like to test this design on (1) Box and Arrow flowcharting programs, and (2) Constraint Networks for graphical user interfaces.

References


