RC23908 (W0603-069) March 9, 2006
Computer Science

IBM Research Report

Relational Blocks: Fully Declarative Visual Application Assembly

Avraham Leff, James T. Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Relational Blocks: Fully Declarative Visual Application Assembly

Avraham Leff
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
avraham@us.ibm.com

James T. Rayfield
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
jtray@us.ibm

ABSTRACT
Dramatic improvements in productivity might be achieved if programmers could fully define applications declaratively. Successful approaches exist for declarative definition of application Views and Models. However, the inability to similarly define an application's business logic in a way that is compatible with the View and Model definitions has stymied progress towards the goal of fully declarative application assembly.

We present *Relational Blocks* as an attempt to solve this problem. *Relational Blocks* expresses business logic in relational algebra, and interfaces to an application's View and Model through a relational API. Early results, from small-scale applications, show that this unified approach enables applications to be defined in a purely declarative fashion. Furthermore, we have exploited this behavior by building a editor that supports the visual construction – and immediate execution – of these declarative applications.

Keywords
relational blocks, declarative programming, relational algebra, relational model, application assembly, visual application design.

1. INTRODUCTION
Declarative programming promises to increase productivity by allowing programmers to define applications (or parts thereof) based on *what* the application should do, rather than on *how* the application should do it [12]. This is in contrast to *imperative programming*, which envisions programs as a sequence of commands (statements) applied to the program state. Although declarative and imperative programming are often described as opposites, in practice they form a spectrum. At one extreme, fully imperative programming specifies all the low-level details about the implementation of an application, such as "load the value 5 into register zero". At the other extreme, fully declarative programming specifies no details about the implementation of an application: "Computer, build me an order-entry application!".

In reality, high-level languages such as C and Java avoid the necessity of programming at a fully imperative level; for its part, declarative programming has not yet advanced to the point of allowing single-sentence descriptions of complex applications. Even so, computer scientists seek to move the level of abstraction further towards the declarative end of the spectrum, because a more concise description of the application should lead (in general) to higher productivity. In other words, if low-level details can be omitted by the programmer, she should be able to write programs more quickly. Productivity is enhanced because programmers are free to concentrate on defining the application's function, and are not distracted by considerations of how that function is achieved.

Declarative programming has been successfully applied to *View* construction such as Web pages that are written in HTML. Programmers describe only what the Web page should look like: web-browsers are responsible for providing the algorithms that render the page onto a display's pixels. Application *Models* can also be described declaratively. For example, relational database schema can be described by the DDL subset of SQL [3], and XML document structure can be specified by schema [14]. Finally, application logic (*Controllers*) can be described in declarative fashion using functional languages (e.g., Haskell and Lisp), logic-based languages (e.g., Prolog), and constraint-based languages (e.g., Oz). XML document instances can be manipulated in declarative fashion using languages such as XSLT [17], and navigated using query languages such as XPath [15] and XQuery [16].

Despite such examples, and despite the promise of increased productivity, most applications continue to be built using imperative programming techniques. This may be because humans tend to think and plan in a sequential style. Imperative programming allows an "incremental" approach to implementing an application since one can start coding without fully understanding everything that needs to be accomplished. Alternatively, it may be because most introductory programming courses are taught with imperative programming languages. More fundamentally, there may be problems with existing declarative approaches that make them less productive than imperative approaches. For example, popular declarative languages only cover part of the application development space. HTML (without scripting) cannot

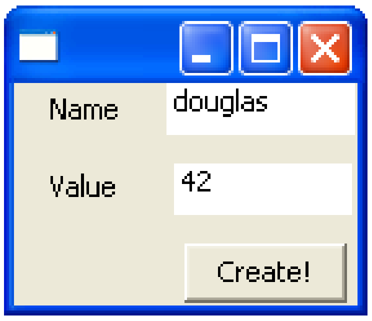


Figure 1: ***Create Example*** **View**

be used to build spreadsheets, and XSLT cannot be efficiently used to build messaging systems. Although separate declarative technologies may exist for different portions of the application space, there is no existing way to integrate the different portions into a single coherent application.

Relational Blocks addresses the inadequacies in existing declarative approaches to interactive-application design. It provides a fully declarative visual design paradigm, including encapsulated Model, View and Controller blocks. All blocks have a common interface, so they can be combined in any way which is meaningful. The visual Controller design takes place in a two-dimension canvas, which allows the designer more freedom to express the system interconnections than the standard text editor allows in one dimension. Interconnections between blocks need not be labeled, so that no effort is expended on naming data flows whose semantics are apparent from the visual layout. The two-dimensional layout also allows the designer to more clearly express the construction of encapsulated composite blocks out of the basic M/V/C blocks. Application construction may also take place in an incremental fashion. Only a small set of blocks is required to bootstrap a working application. M/V/C blocks may be added, removed, or rewired at any time, and the application immediately verified and re-executed.

1.1 Create Example

The following application fragment can make the above issues more concrete. *Create Example* allows a user to create {*Name, Value*} records (rows) in a database system. Figure 1 is the View which is seen by the user. The user enters a *name* in the Name text box, a *value* in the Value text box, and presses the Create button to create the record.

The "conventional" (all imperative) approach to building applications (e.g., by coding in Java) uses widget-technologies such as Swing or SWT to define the UI panels, text-entry fields, and button. Event-handler code is then linked to the widgets so that the following sequence is executed when the user clicks the Create button:

1. Read the values of the Name and Value text boxes
2. Use these values to issue a JDBC INSERT to the database Model

The most obvious problem with the "all-imperative" approach is that an intrinsically visual activity (defining the application's View) is done in a non-visual medium. The developer is typically required to hand-code the size of the windows, the placement of all the widgets, the widget modifier flags (e.g., "resizable"), and so forth. Often they resort to drawing the View by hand on graph paper in order to determine the correct parameters for the imperative code!

```
<def:Code>
<![CDATA[
  void ButtonClick(object el, ClickEventArgs cea)
  {
    Button btn = (Button) el;
    FlowPanel parent = (FlowPanel) btn.Parent;
    parent.Children.Remove(btn);
    parent.Children.Insert(1, btn);
  }
  ]]>
</def:Code>
```

Figure 2: Non-visual imperative code accessing declarative visual components

Motivated by this observation, visual approaches to View implementation use tools such as Dreamweaver [11] and IBM Rational Application Developer [9] to declaratively define Views such as that shown in Figure 1. This approach is satisfactory for static views which are not coupled with Model and Controller logic (e.g., static HTML pages). However, it does not extend well to dynamic Views, where the contents of the View has a signficant dependency on the current state of the Model. For example, visual HTML tools do not allow the displayed table size to be based on the current contents of the Model. This requires integration of declarative (View) and imperative (Model and Controller) approaches.

Mixed declarative/imperative approaches such as Microsoft's XAML [1] provide a means to integrate declarative definition of the View with an imperative definition of the Controller. XAML allows programmers to declaratively define a View layout of text, images, and controls, using a visual editor or XML. However, the Controller logic must be implemented in a standard imperative language such as C#, either embedded in the application XML or in a "code behind" file. The View widgets must call imperative event-handlers when interesting events occur; the imperative code must access the View widgets using labels and graph navigation. Both of these patterns are shown in Figure 2 (excerpted from [1], Figure 7), which illustrates an event-handler for a Button click. In this example, clicking the Button causes it to move into the second position in the View. Note that the XAML only declares the *initial* View; after a Button is pressed, the View must be updated using imperative C# code, and no longer corresponds to the View described by the XML.

The impedance mismatch caused by mixing declarative and imperative programming is eliminated if Model and Controller design use a visual paradigm, just as visual View design does. Visual design of Model components based on the relational Model is fairly straightforward: relational tables look like tables, with a column (or row) for each attribute (e.g., see [9]). Unfortunately, visual design of Controller logic is not well understood. *Relational Blocks* is an attempt to address the problem of visual declarative Controller design in a way that integrates with existing visual declarative paradigms for View and Model design.

1.2 Relational Blocks

In this paper we propose the *Relational Blocks* approach for building *entire* applications using only a visual declarative approach. Section 2 introduces the *Relational Blocks* approach, and explains how it addresses the problems discussed above. We also discuss the integration of transactions and exceptions into the programming model. Section 3 discusses the *Relational Blocks* visual editor, the technologies used in its implementation, and illustrates its use in the assembly of a small sample application.

We have just begun to explore the use of *Relational Blocks* in visually assembling applications. We describe some of the challenges that we are actively addressing in Section 4.

2. RELATIONAL BLOCKS APPROACH

The *Relational Blocks* approach for declarative visual design of Controller components is based on relational algebra. The particular formulation we have chosen is based on *Relational A* ([4], chapter 4). Relational algebra is in many ways a perfect match for a Model represented by a relational database, since relational algebra provides a declarative description of the data that should be extracted from the relational Model and how it should be manipulated [2]. Also, relational algebra operations are reasonably simple in isolation, small in number, and can be easily composed to form more complex operations. Relational algebra also maps nicely onto a visual representation of interconnected blocks, similar to an electronic circuit. The operations defined by Relational A are: NOT (set complement), REMOVE (remove an attribute), RENAME (rename an attribute), AND (natural Join), OR (generalized union), and TCLOSE (transitive closure).

One problem is that relational algebra by itself does not provide a means to *update* a relational Model, since relational algebra expresses a set of time-invariant relationships between outputs and inputs. The UPDATE and INSERT operations found in SQL are therefore not found in a relational algebra such as Relational A.

One approach to handling updates is to use an imperative programming language ([4], Chapter 5, *Tutorial D*). *Relational Blocks* takes a different approach: it treats the Model as the state of a Relational State Machine. Thus, the execution of the State Machine causes the Model (and View) to be updated. The State Machine is typically "clocked" (makes transitions) on View events. Thus, the application state is specified up front, and *Relational Blocks* supplies a well-defined model to update this state

In order to express the complete application using relational algebra, the View components must also be integrated with the relational algebra Controller and relational Model. *Relational Blocks* therefore encapsulates View components within a relational API that is compatible with the Model and Controller components. This allows the entire *Relational Blocks* application to be expressed as a directed graph of Model, View, and Controller components, all defined in a declarative fashion:

- *Model*: the Model is expressed as a set of relations (tables). Visually, the Model takes the form of a mathematical table, as in existing visual tools for relational database design. The Model may consist of persistent and/or transient portions (this is an application design issue; the *Relational Blocks* paradigm makes no distinction between persistent and transient Models). The Model has an output, which is the current state of the database, and an input, which is the desired next state of the database.
- *View*: the View is expressed visually, by laying out widgets to form the desired user interface screen. Program-writeable widgets have an input, expressed as a relation. For example, a label might have a single tuple with *text* and *font* attributes. Program-readable widgets have an output, expressed as a relation. For example, a slider might have a single tuple with a single attribute *value* in its output. Note that read/write widgets (e.g., text boxes) effectively become part of the Model, since they act as a mini database. More complicated widgets such as tables and lists are multi-tuple relations. The View widgets are thus directly compatible with the Model and Controller components.
- *Controller*: the application's controller logic is described declaratively using relational algebra. The input to the Controller is the current state of the Model and the current values of the readable widgets. The output of the Controller is the next state of the Model and the next state of the writeable View widgets.

These relational components are assembled in a way that is analogous to hardware-chip assembly. A component is described syntactically solely in terms of its *input* and *output* terminals, and its interconnection to other components. Component semantics are described mathematically (using the relational algebra), or approximated in natural language. Continuing the hardware analogy, the assembled components form a directed graph. The *Relational Blocks* runtime maintains this graph, whose state changes only when a "clock tick" occurs. The functional (Controller) portions of this graph must be acyclic; however, cycles may pass through the Model and View components, because the cycles are "broken" by the clocked nature of the Model and View.

2.1 Relational Blocks Example

In this section we illustrate the basic concepts of *Relational Blocks* using the *Create Example* introduced in Section 1. The *Create Example* View is shown in Figure 1, and its Model is shown in Table 1. It consists of a single table, with *name* and *value* attributes, each of type string.

The full example (Model, View, and Controller) is shown in Figure 3. (Figure 3 is an annotated version of the *Relational Blocks* visual editor canvas; Figure 8 is the actual screenshot.) The Figure shows the Controller blocks in addition

Table 1: Example Model

name string	*value* string
.	.
.	.

to the View widgets and Model definition. The output of the Name text-box is a single-tuple relation with the attribute *text*. This output flows to the block titled REMOVE+RENAME #1. In the current implementation, the REMOVE and RENAME algebra functions are combined into a single block, since they are often used together. In this case, the block renames the *text* attribute to *name*, and removes any other attributes. Similarly, the block titled REMOVE+RENAME #2 renames the *text* attribute of the Value text-box to *value*, and removes all other attributes.

The two relations from the REMOVE + RENAME blocks flow to the JOIN block, which performs an AND operation. This results in a single-tuple relation with the attributes *name* and *value*, the values of which are taken from the Name and Value text-boxes, respectively. (Note that AND reduces to a Cartesian product when there is no common attribute name in the joined relations, as in this case.) Thus the relation header from the AND operation is exactly compatible with the relation header of the Model block, which is required by the *Relational Blocks* implementation.

The "Create" button produces a no-attribute (degree zero) relation. Degree-zero relations are used to indicate TRUE and FALSE in Relational A. TRUE contains a single no-attribute tuple, and FALSE contains no tuples. TRUE is the identity relation for AND (any relation AND TRUE yields the original relation). Any relation AND FALSE yields the empty relation.

The ENABLED INSERT block has two inputs: `insert` and `enable`. This is actually a macro block which contains the blocks shown in Figure 4. `insert` specifies the new tuple(s) to be inserted into the Model at the next state transition, iff the `enable` input is TRUE. In the macro block, `insert` is AND'ed with `enable` to form a relation which is either equal to `insert` or empty, depending on the value of `enable`. The resulting relation is OR'ed with the current value of the Model, so that the next value of the Model is either augmented by the `insert` tuple(s), if `enable` is true, or unchanged, if `enable` if false.

State transitions of the application in this example are triggered by user actions on the View. Thus, when the "Create" button is clicked, the contents of the Name and Value text boxes are merged into a single tuple, which is then inserted into the Model. If a different View event occurs (e.g., the user clicks on one of the text boxes), no change is made to the Model, since the `enable` value is false.

2.2 Relational Blocks Runtime

More generically, the *Relational Blocks* runtime execution is event-driven, typically by View events (e.g., clicking the "Create" button). (Another event source might be external events, such as database triggers, but this has not yet been studied.) Each event executes the following algorithm:

1. Evaluate the inputs to all model blocks. Typically this is a recursive process, because the inputs depend on the outputs of other algebra blocks, View widgets, and Model block outputs. In the example above, evaluating the input to the Model block embedded in the ENABLED INSERT macro requires evaluating the OR block inside the macro, the AND block inside the macro, the AND of the REMOVE + RENAME blocks, and the outputs of the text boxes. Also, the `selected` output of the "Create" button must be evaluated.

2. Each model block then updates its state, using the values just calculated in step 1. In this step, the Relational State Machine transitions to the next state. In the example above, the Model will insert the new {*Name, Value*} tuple iff the "Create" button was pressed. Future evaluations of Model outputs will equal this new state.

3. All writeable View blocks update their state based on their current inputs. Note that the View is thus updated synchronously by the event-handler, and asynchronously by the user.

Imperative languages almost always have a model of computation based on the concept of a *program counter*. The program counter indicates exactly where the flow of execution is at all times. (Note that multithreaded models will have one program counter per thread.) This is made explicit by the way that imperative-language debuggers allow developers to set *breakpoints* for an application's execution so that the application is suspended when the flow of execution reaches that point. In contrast, *Relational Blocks*, in common with other declarative approaches, does not have a concept analogous to a program counter. Instead, *Relational Blocks* uses an event-driven state-machine model: on each event, the next state is evaluated functionally, and the state machine is then advanced.

2.3 Flexibility/Power

One capability which is needed for more complex applications is a general function-evaluation mechanism. For example, suppose the type of the *value* attribute is changed to type integer instead of string. The application design must be enhanced to convert the string value output from the REMOVE+RENAME #2 block to an integer type. Reference [4] discusses a theoretical approach to implementing functions using relational algebra operations:

1. Create a constant-valued relation with attributes for all of the function's inputs and outputs. In our example, Table 2 defines the attribute *value_string* as the string-valued input, and *value_integer* as the integer-valued output, the latter being the equivalent integer representation of the former.

2. Perform an AND operation between the function inputs and the constant relation. The result is a relation with a new attribute, *value_integer*, that is the function result.

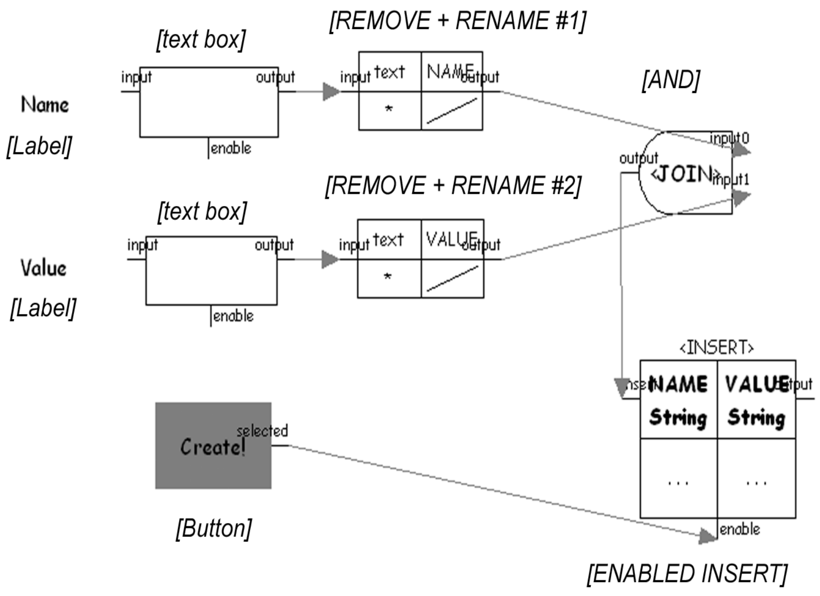


Figure 3: Example Model/View/Controller

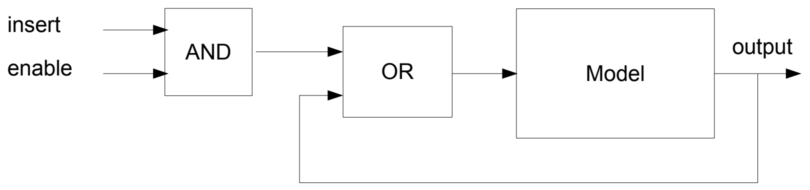


Figure 4: Enabled Insert Model

Table 2: Relation to Convert String to Integer

value_string	*value_integer*
"0"	0
"1"	1
"2"	2
.	.
.	.
.	.

To see how this works, suppose that the user has entered the string "42" into the Value text-box. The output of the text-box is the following relation:

text
"42"

The REMOVE+RENAME #2 block is modified to rename *text* to *value_string*, producing the following relation:

value_string
"42"

Performing an AND operation with the relation of Table 2 produces this result:

value_string	*value_integer*
string	integer
"42"	42

Finally, an added REMOVE + RENAME block removes the *value_string* attribute and renames *value_integer* to *value*, producing the following input relation to the ENABLED INSERT block:

value
42

The problem with this approach is that many useful functions require that the constant relation be of infinite size. In our example, there are an infinite number of strings which express legal integer values. *Relational Blocks* therefore implements the *equivalent* functionality by allowing the developer to specify functions in terms of expressions. A *Relational Blocks* FUNCTION BLOCK macro is specified by:

1. The name of the function output attribute. In our example, this is *value_integer*.
2. An expression that is applied to the function input attribute(s) to produce the function output attribute. In the example above, the expression is `integer_value(value_string)`.

For each input tuple, the function-evalution block produces a new tuple which is equal to the input tuple extended with the function-output attribute. Thus, the block output is the same as would be produced by an AND with a function expressed as a constant relation.

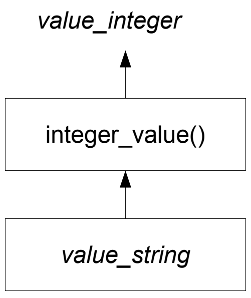


Figure 5: Expression Tree for *value_integer* = integer_value(*value_string*)

Internally, the expressions are parsed and represented as expression trees, as is typically done by compilers. The expression tree for the string-to-integer conversion example is shown in Figure 5. Expression-tree terminal nodes may be constants or input attributes. Expression-tree non-terminal nodes may be unary functions, such as integer_value(), unary minus, etc., or binary functions, such as arithmetic sum, concatenate, etc. *Relational Blocks* provides a library of node implementations to support commonly used functions and operators. If desired, additional node implementations can be implemented in an imperative language (Java, in the current implementation).

2.4 Exception Handling

Under any application design paradigm, errors can be divided into two classes: those that can be detected before runtime, and those which are detected only at runtime. The vast majority of *Relational Blocks* errors can be detected while designing the application, before running the application. All Model specification errors can be detected before runtime. Also, note that all block interconnections carry information about their Relation Headers: that is, the set of attribute names and types which flow on the connection. This enables the *Relational Blocks* design tools to validate, before runtime, that output connections are compatible with the inputs that they are connected to. For example, if the relation header of the `insert` input to the ENABLED INSERT block does not match the relation header of the Model, *Relational Blocks* detects an error and does not allow the application to be executed. Similarly, an attempt to RENAME or REMOVE an attribute that does not appear in the input relation header, can be detected at design time.

The only errors which cannot be detected before runtime manifest themselves during execution of FUNCTION BLOCKs. For example, the string "4t2" is not a legal string version of any integer, and therefore does not appear anywhere in the *value_string* column of Table 2. Thus a FUNCTION BLOCK which attempts to convert this to an integer will yield the empty relation. Intuitively, it is desirable for *Create Ex-*

ample to detect that the user has input an illegal value (a string that cannot be converted to an integer), and to recover gracefully.

Relational Blocks cannot use the exception model of imperative languages such as C++ and Java. Like other declarative languages, *Relational Blocks* does not have a "flow of control" that can be altered by an exception. More specifically, the Controller semantics is fixed by the relational-algebra blocks which make up the application design. Algebra blocks cannot "refuse" to produce a relation output, because the downstream blocks are depending on that output. Because *Relational Blocks* implements FUNCTION BLOCKs with expression trees, some allowance must be made for expression nodes which are provided with invalid inputs.

We considered an approach in which, given illegal input, the expression-tree evaluation yields no output, and thus does not appear in the FUNCTION BLOCK output. However, in order to make it easier to detect and handle such problems, *Relational Blocks* uses a "Replacement Model" approach [18]. In this approach, the output of an expression with invalid inputs is replaced by a fixed value, with the fixed value specified as a property of the FUNCTION BLOCK.

Also, the *Relational Blocks* FUNCTION BLOCK outputs contain an additional attribute called *invalid_domain*, which indicates whether an output tuple corresponds to an input tuple whose attribute value(s) were invalid inputs to the FUNCTION BLOCK expression. A distinguished value for the *invalid_domain* attribute indicates that the expression evaluated successfully. The FUNCTION BLOCK therefore **always** provides a result tuple for each input tuple. The result tuple(s) contain an attribute indicating an expression-input error, if one occurred; if an error did occur, the value of the function-output attribute is the statically-configured property value.

Downstream blocks may check the value of the *invalid_domain* attribute. They can also check whether the function's output is the value that signifies an error, and generate a popup or status-line message as desired. Such checking must be part of the application design itself; it is not provided by the *Relational Blocks* framework. We resisted the temptation to use NULL [3] as the output value that signals an exception. Partly because the semantics of the NULL value is overloaded, and for other reasons [4], we decided that *Relational Blocks* should not use the NULL-based approach for handling exceptions.

In *Relational Blocks*, only the framework or expression-node *implementation* can actually throw imperative exceptions (due to programming errors or database communication errors). If thrown, these exceptions are caught in the top-level processing loop (Section 2.2), an appropriate message is sent to the user, and the application's current transaction (Section 2.5) is rolled back.

2.5 Transactions

Relational Blocks supports the design and execution of *transactional* applications: i.e., applications that access and update shared state using the well-known *ACID* semantics [8]. A key issue, therefore, is how to transactionally scope an application's activities. Frameworks such as Enterprise JavaBeans [6] declaratively associate transaction semantics on a *per-method* basis. Developers can specify, for example, that the invocation of the *setAccountBalance()* method should start a transaction, unless a transaction is already active. This approach is unsuitable for *Relational Blocks* because the notion of a "method" does not exist. More fundamentally, we believe that transactions should not be scoped at method granularities, but rather should be controlled by user-initiated activities. Users expect that transactions are initiated (and soon committed) when they click the "submit" button after filling out a form (e.g., a funds-transfer screen). It seems more useful, therefore, to specify transactional boundaries based on user interactions. (In practice, transaction boundaries for imperative application development are also usually based on user interactions. With EJBs, each user interaction typically calls a top-level method which declaratively begins and commits the transaction.)

Since declarative programs have no explicit flow-of-control, developers cannot insert transaction begin, commit, and rollback statements at certain points in the flow. Instead, *Relational Blocks* implicitly begins a transaction with each user-interaction event, and commits the transaction immediately after processing the event. A special ROLLBACK block is available to control transaction rollback. If the input to the ROLLBACK block is TRUE, the transaction is aborted instead of being committed. This allows the application designer to specify conditions under which the database updates should not take place. In the example above, an entered value of "4t2" will be converted to some default value (e.g., zero) specified as a property of the Function Relation. However, the zero value should not be inserted into the database. A typical application design would convert the *invalid_domain* attribute indicator into a rollback condition. Alternatively, the application designer could detect error conditions and use them to disable all Model blocks. If done correctly, this is functionally equivalent to forcing a rollback, although it is probably more complex and error-prone.

2.6 Relationship to OO Application Construction

As with object-oriented design, *Relational Blocks* imposes a strict encapsulation on the basic blocks (Model, View, and Controller) that it provides. Developers could build their own "macro" blocks, composed from basic blocks provided by *Relational Blocks*, although this is not currently supported. The developer determines the granularity of a *Relational Blocks* macro component: should it be only a Model, View, or Controller component; or should it be a fusion, say, of Model and Controller function such as the ENABLED INSERT block discussed above? This would allow, for example, a generic "login page" block to be built and shared among multiple applications.

Regardless of the developer's decision, the *Relational Blocks* API ensures a complete separation of interface from implementation. Well-known benefits follow from this approach including: reducing complexity by hiding information; separation of concerns; and ensuring that changes to a component's implementation do not ripple-through the rest of the application. *Relational Blocks* is technology neutral: even though it is implemented in Java, the implementation could be replaced with another language without changing the semantics. In fact, *Relational Blocks* can be considered as the extension of the object-oriented approach to application assembly, such that current approaches to declarative application assembly are augmented with encapsulation.

In contrast, an approach that includes both declarative and imperative portions tends to discourage strict encapsulation. Looking at the XAML [1] example in Figure 2, the imperative Controller code must have a deep understanding of the details of the View, because the View API is accessed at the widget level. Thus small changes to the View may require rewriting the Controller code.

Relational Blocks, however, differs from the classic object-oriented approach in that the fundamental Model concept is a *relation* rather than an *object*. There are several reasons for this decision. First, we wish to leverage the huge existing base of relational data and applications. Relational database technology is mature, and provides persistence, transactions, and security. More fundamentally, we believe that relational algebra is the most natural way to express application logic in a declarative fashion. Finally, we are convinced that attempts to map *single* object instances to *entire* relational tuples are fatally flawed (see [4] chapter 2). This is not a problem that is specific to *Relational Blocks*, but affects all object-relational mappings. (See [10] for a brief review of previous efforts in this area.) However, using objects as attribute (column) values works well.

3. RELATIONAL BLOCKS PROTOTYPE

We have implemented a *Relational Blocks* prototype using the Graphical Editor Framework (GEF) [7], an Eclipse [5] tools project. GEF allowed us to easily create a rich graphical editor that maps the *Relational Blocks* application model to a graphical editing environment. GEF consists of two Eclipse plug-ins. The first, *draw2d*, is an SWT-based drawing plug-in that provides a layout and rendering toolkit for displaying graphics. The second, *gef*, provides a framework for common graphical editor operations based on a model-view-controller architecture. Developers provide the application model: by using GEF, they are able to apply changes made to the view (*via* the editor) to the model; conversely, changes made to the model can be immediately applied to the view. GEF is completely application neutral and we used it to build the *Relational Blocks* prototype fairly quickly. As shown in Figure 6, the pallette currently provides a small set of pre-fabricated widget, model, and algebra blocks. Developers drag blocks from the palette to some location on the application-design panel. By selecting the "connect" palette entry, developers can wire one block to another. The editor displays the "pin" names, and does not allow illegal wirings to be constructed. The property-sheet view is used to modify various properties: e.g., the database and table names of a ENABLED INSERT block.

In this section, we shall illustrate the prototype's features by walking through the steps used to build the *Create Example* application fragment. We discussed the semantics of *Create Example* in Section 1, as well as the *Relational Blocks* components used to build it. Here, we focus on how a developer uses the visual editor to assemble the application. The assembly steps do not have to occur in any particular order, and the steps can be interleaved.

3.1 Sketch the UI

The prototype is not focused on rendering a polished UI. Instead, *Relational Blocks* allows users to sketch the UI by laying out widgets such as labels, buttons, and text-entry fields on the screen. We plan to extend the set of available widgets to include more complicated types such as tables and lists. We also plan to improve the accuracy of the UI, perhaps by using the Eclipse *Visual Editor* [13] project.

Figure 6 shows how the developer sketched the UI. It consists of two text-entry fields (for the Name and Value), each associated with a corresponding label. The user clicks the "Create" button after supplying the Name and Value information. Label widgets currently have neither inputs nor outputs. Button widgets have a boolean valued output relation: it contains a TRUE value when the user has clicked the button, and is FALSE otherwise. This wire is an "enable" wire, as it determines whether the values in the text-entry fields will propagate to the model (Figure 7).

3.2 Design the Model

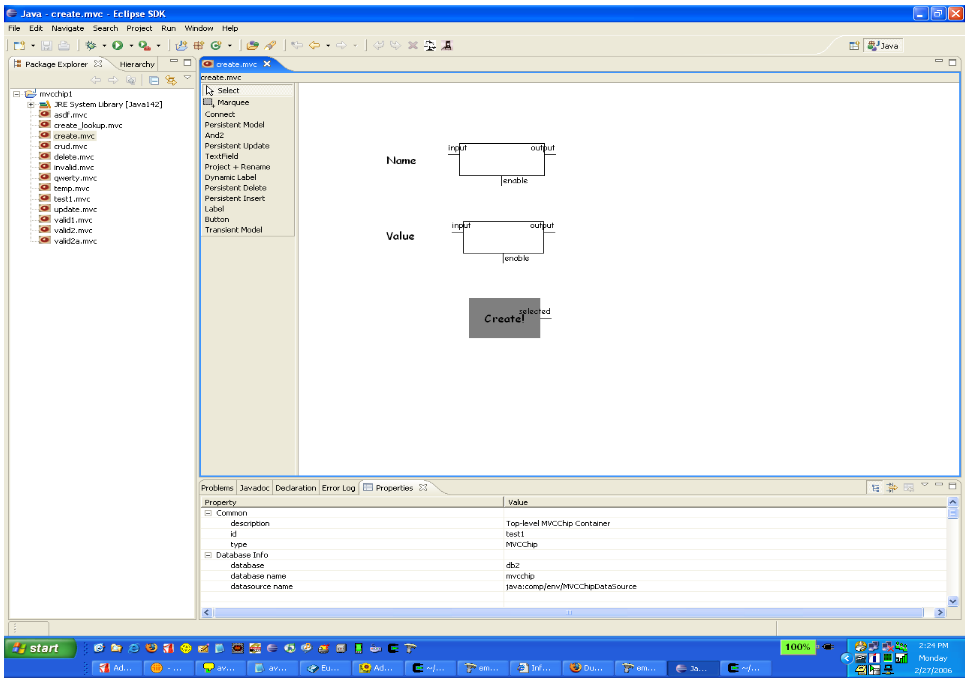
Figure 7 shows the developer designing the model in "bottom-up" fashion. She specifies the name of an existing relational database table for a ENABLED INSERT block, and the editor automatically extracts the Relation Header from the table's meta-data. The use of transient relations (e.g., an application "white-board") requires that the developer input the Relation Header through a dialog. As discussed above, ENABLED INSERT is composite block, consisting of a (persistent) model and enabled "insert" logic. A developer could explicitly assemble the required blocks, but *Relational Blocks* supplies the macro as a convenience. We currently provide corresponding ENABLED UPDATE and ENABLED DELETE composite blocks.

3.3 Specify Application Logic

Figure 8 shows the developer assembling the application by (1) inserting the application logic *via* algebra blocks and (2) wiring the blocks together, to form the *Create Example* application graph. The Figure shows that clicking the button determines whether the ENABLED INSERT block is enabled, and thus whether the text-entry fields update the Model. The tuple that is inserted into the model is formed by AND of the text-entry fields. Because these values have a Relation Header that differs from the model's, REMOVE + RENAME blocks are used to morph these Relation Headers to match the model's.

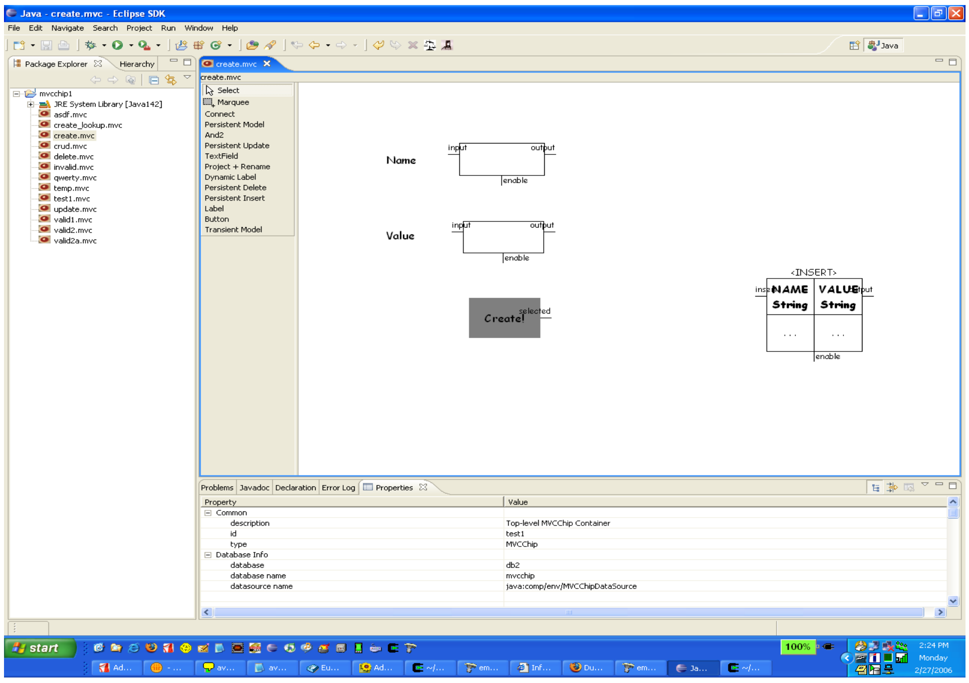
3.4 Run the Application

Developers can, at any time, invoke the "validate" or "execute" functions by clicking on the corresponding editor icon. Although the editor enforces some part of the *Relational Blocks* semantics through its property-sheet dialogs and forbidding illegal wiring, this is done only on a per-block basis. Validation looks at the "whole picture", and reports whether each of the blocks is valid, has warnings, or has errors. An error implies that the application cannot be executed in its current state. A warning indicates that the application will



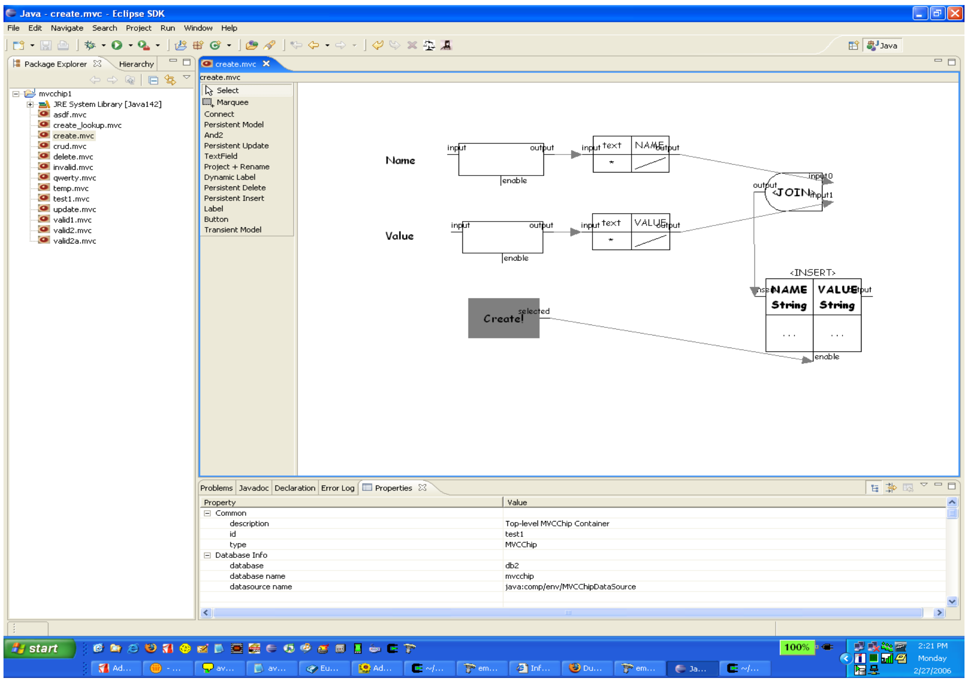
Property	Value
Common	
description	Top-level MVCChip Container
id	test1
type	MVCChip
Database Info	
database	db2
database name	mvcchip
datasource name	java:comp/env/MVCChipDataSource

Figure 6: Using the *Relational Blocks* Editor to Sketch an Application UI



Property	Value
Common	
description	Top-level MVCChip Container
id	test1
type	MVCChip
Database Info	
database	db2
database name	mvcchip
datasource name	java:comp/env/MVCChipDataSource

Figure 7: Using the *Relational Blocks* Editor to Specify the Model



Property	Value
Common	
description	Top-level MVCChip Container
id	test1
type	MVCChip
Database Info	
database	db2
database name	mvcchip
datasource name	java:comp/env/MVCChipDataSource

Figure 8: Adding Controller Logic to the Application

execute, but that the behavior might not be as desired (e.g., a text-box output left unconnected).

Executing the application is a superset of validation, since it constructs a graph comprised of runtime blocks, each of which corresponds to a design-time block displayed in the visual editor. Further semantic checks are performed, and if valid, a concrete version of the application is constructed. Several tasks are performed at this point. First, a runtime graph (Section 2.2) is constructed such that a directed edge exists from runtime block$_a$ to runtime block$_b$ *iff* an design-time wire connects an output terminal of design-time block$_a$ to an input terminal of design-time block$_b$. Second, view widgets are mapped to concrete SWT widgets in a layout that conforms to that displayed in the visual editor. An SWT event-handler is constructed for each SWT widget that delegates all event handling to the runtime graph's `processEvent()` method. `processEvent()` performs the state-machine clocking. Finally, database connections are established as necessary to the relational database tables that correspond to ENABLED INSERT blocks. At this point, the application is displayed to the user, and can be executed "as is". Figure 9 shows the result.

4. CHALLENGES

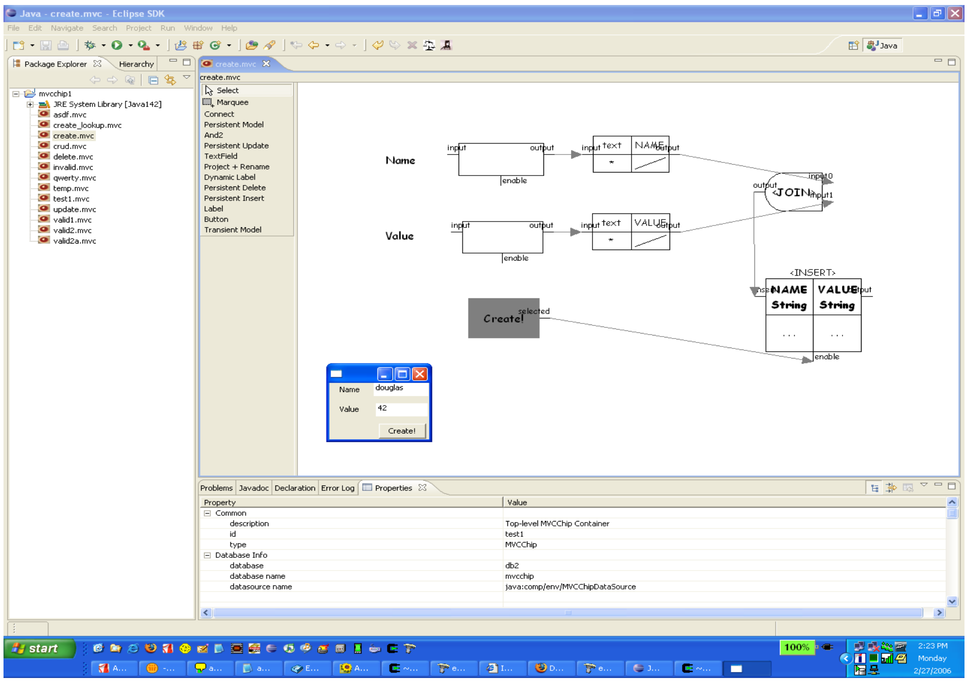
Our claim that *Relational Blocks* enables and supports fully declarative visual application assembly has been validated only with "demo-sized" applications. We face several challenges as we scale *Relational Blocks* to build real-world applications.

One issue is complexity: *Relational Blocks* designs will almost certainly grow too complicated to be displayed on a single screen. A complicated View which takes up all the screen real estate cannot be packaged on the design screen with its Controller and Model. The natural solution to this problem is to introduce hierarchy. The top-level design of the application should fit on a single screen. Designers can "drill down" to lower levels of the design to see more detail, or move up to see the entire application.

In order to support hierarchy, *Relational Blocks* needs to support designer-defined *composite* blocks, or macros. This allows the developer to encapsulate a set of Model, View and/or Controller blocks, and treat it as a single reusable block in higher-level designs. For example, the designer might design a composite block for each UI screen in the application. Connections between individual UI screens and the relevant Controller and Model can then be shown on a different screen. This also has the benefit of allowing the View design to be modified without affecting the Controller and Model, as long as the interface to the View remains unchanged. In addition to user-defined composite blocks, *Relational Blocks* should come with an existing library of composite blocks; for example, login Views, user database Models, and priority encoders (useful for radio button widgets).

Also, virtually all significant applications require multi-View navigation. Thus, *Relational Blocks* must have design-time view which shows a "navigational" view of all the Views of the application. Finally, designers must be able to integrate *Relational Blocks* applications (or portions of applications) with existing, non-relational-block, components. We must therefore devise a way to easily wrap a thin relational shell around such components. We already have some experience with this requirement, since many of the basic blocks are wrapped versions of SWT widgets and JDBC artifacts.

Despite these challenges, the approach of visual application



Name	douglas
Value	42

Property	Value
Common	
description	Top-level MVCChip Container
id	test1
type	MVCChip
Database Info	
database	db2
database name	mvcchip
datasource name	java:comp/env/MVCChipDataSource

Figure 9: Direct Execution of the Application from the *Relational Blocks* Editor

assembly used by *Relational Blocks* seems promising.

5. REFERENCES

[1] Avalon. `http://msdn.microsoft.com/msdnmag/issues/04/01/Avalon/default.aspx`, 2006.

[2] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 1970.

[3] C. J. Date and Hugh Darwen. *A Guide to SQL Standard*. Addison-Wesley, 4rth edition, 1996. ISBN: 0201964260.

[4] C.J. Date and H. Darwen. *Foundation for Object/Relational Databases: The Third Manifesto*. Addison-Wesley, Boston, MA, 1998.

[5] Eclipse Project. `http://www.eclipse.org/eclipse`, 2006.

[6] J2EE Enterprise Javabeans Technology. `http://java.sun.com/products/ejb/`, 2006.

[7] Graphical Editing Framework. `http://www.eclipse.org/gef`, 2006.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, USA, 1993.

[9] IBM Rational Application Developer for Websphere Software Version 6.0. `http://www-8.ibm.com/software/includes/pdf/rat_app_dev_LoRes.pdf`, 2006. Publication number GC34-2464-00.

[10] Comparing LINQ and Its Contemporaries. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/linqcomparisons.asp`, 2006. T. Neward.

[11] David McFarland. *Dreamweaver MX 2004: The Missing Manual*. O'Reilly Media, 2003. ISBN: 0596006314.

[12] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, Mass, 2004.

[13] Visual Editor Project. `http://www.eclipse.org/vep/WebContent/main.php`, 2006.

[14] XMLSchema. `http://www.w3.org/XML/Schema`, 2006. See also 'XML Schema' published by O'Reilly.

[15] XPath. `http://www.w3.org/TR/xpath`, 2006. See also 'Xpath and Xpointer', published by O'Reilly.

[16] XQuery. `http://www.w3.org/XML/Query/`, 2006.

[17] XSLT. `http://www.w3.org/TR/xslt`, 2006. See also 'Learning XSLT', published by O'Reilly.

[18] S. Yemini and D. Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1985.