RC24014 (W0608-005) August 2, 2006
Computer Science

IBM Research Report

Relational Blocks: Declarative Visual Assembly of Enterprise Applications

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

Relational Blocks: Declarative Visual Assembly of Enterprise Applications

Avraham Leff James T. Rayfield

Abstract

Relational Blocks is a visual programming language, editor, and runtime that enables developers to create enterprise applications without adding any non-visual or imperative code. This is possible because the model, controller, and view language primitives use a compatible relational API and have the same visual representation. Unlike many other visual programming languages, *Relational Blocks* is general-purpose and oriented towards building transactional applications that have significant amounts of business logic which interacts with the application's view and model.

We believe that using *Relational Blocks* can greatly enhance developer productivity because of the higher levels of abstraction used to develop the application and because the GUI is developed visually. To evaluate this claim, we compare a *Relational Blocks* version of a non-trivial application to a Java version, and show that the unified approach of *Relational Blocks* can considerably simplify application development.

Index Terms

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

I. Introduction

A. Motivation for Relational Blocks

Relational Blocks is a visual programming language, editor, and runtime that enables developers to create enterprise applications without adding any imperative code. *Relational Blocks* has the following characteristics:

IBM T.J. Watson Research Center email: avraham@ibm.com
IBM T.J. Watson Research Center email: jtray@ibm.com

- *Relational Blocks* uses the same relational API and visual representation for all of the language primitives: model, algebra (controller), and widget (view) blocks.
- Composite blocks use the same relational API and visual representation as primitive blocks.
- Applications are assembled *visually* with a visual editor with no imperative code required from the developer.
- *Relational Blocks* is a general-purpose (rather than domain-specific) language, and is intended for use in developing transactional enterprise applications.
- *Relational Blocks* applications may be directly executed from the visual editor.

Our motivation for building *Relational Blocks* is to increase developer productivity. Developer productivity is increased because:

- Visual portions of an applications are developed visually, rather than by using imperative code or text-based declarative languages like HTML.
- A visual editor is used to assemble the application, and all of the application's components ("blocks") use the same API and have the same visual representation. Because blocks have a common interface, they are easily combined to produced the desired effect. Developers are not forced to use different languages to develop different parts of the same application.
- Because business logic is expressed in a two-dimensional canvas, developers have more freedom to express application interconnections than with the one dimension allowed by the standard text editor. Interconnections between blocks need not be labeled, so no effort is expended on naming data flows whose semantics are apparent from the visual layout.
- Understanding the language semantics requires only understanding the well-known relational algebra concepts. The fact that the relational algebra has a precise mathematical definition (set theory) is another advantage.
- The "code, test, and debug" development cycle is reduced because applications are directly

executed from the visual editor. Incremental construction is encouraged because only a small set of blocks is required to bootstrap a working application. Blocks may be added, removed, or rewired at any time, and the application can be immediately validated and re-executed.

B. Background & Related Work

Because it uses visual representations (blocks and wires) to accomplish all of what is done textually in conventional languages, *Relational Blocks* is visual programming language (*VPL*). In addition, *Relational Blocks* is a strongly “declarative” (in contrast to “imperative”) language as can be seen from the lack of explict flow-control mechanisms, its use of a high-level relational algebra, and in the stateless nature of its algebra blocks [19]. The contribution of *Relational Blocks* can therefore best be understood by contrasting it to both non-visual, imperative approaches, and with other VPLs.

In general, the use of programming abstractions increases developer productivity. Well known examples of such abstractions include procedural (relative to unstructured) programming, and object-oriented (relative to procedural) programming. Similarly, increasing an application’s ratio of declarative (specifying *what*) to imperative (specifying *how*) programming [19] increases productivity. In fact, declarative programming techniques have been successfully applied in *parts* of enterprise application assembly. For example, in the domain of GUI construction, HTML is successfully used to build Web pages: 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. In the domain of defining an application’s Model, relational database schema can be described by the DDL subset of SQL [8], and XML document structure can be specified by schema [21]. Finally, application logic 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.,

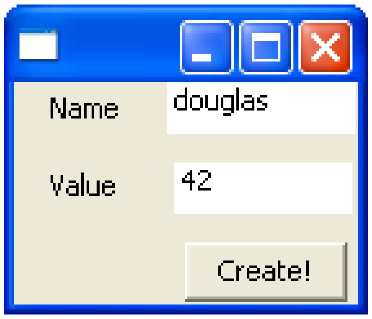


Fig. 1. Requirement to Integrate Different Model/View/Controller Domains

Oz). XML document instances can be manipulated in declarative fashion using languages such as XSLT [24], and navigated using query languages such as XPath [22] and XQuery [23].

Although successful in limited application domains, most current approaches require varying degrees of imperative code when assembling enterprise applications. The task of integrating non-trivial view, model, and controller portions of an enterprise application is so complex that the fine-grained control afforded by imperative techniques is assumed to be necessary.

Consider Figure 1 which shows the front-end of an application that allows a user to create $\{Name, Value\}$ records (rows) in a database system. 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. Even this trivial bit of function requires that the following tasks be integrated in a single application:

1. view construction
2. event handling (to read the state of the name and value text boxes and thus bridge the view and the controller domains)
3. model definition (to access the database table and insert the new record)
4. business logic (to handle the case where a record with that key already exists).

At one extreme, developers code the *entire* enterprise application using a single language such as Java. Widget-technologies such as Swing or SWT are used to define the UI panels, text-entry fields, and button. Event-handler code is attached to the widgets to bridge the view and model domains and to specify the business logic. The advantage of this "all imperative" approach is developers can use a single language to code the entire application. The most obvious problem with this 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. Developers often resort to drawing the view by hand on graph paper in order to determine the correct parameters for the imperative code!

This disadvantage is sufficiently great to motivate the use of hybrid approaches in which declarative or visual techniques are used to define the view portion of an application while imperative code is used to bridge the view to the rest of the application. For example, tools such as Dreamweaver [17] and IBM Rational Application Developer [15] allow users to define views visually. Some tools ([15]) even enable the visual design of model components using the relational model: relational tables are represented as "tables", with a column displayed for each attribute. Microsoft's XAML [1] similarly allows programmers to declaratively define a view layout of text, images, and controls, using a visual editor or XML. 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 have a significant dependency on the current state of the model. For example, most visual HTML tools do not allow the displayed table size to be based on the current contents of the model at runtime. More fundamentally, only view construction and model definition (via DDL[8]) is done declaratively: event handling and business logic is done through imperative code.

Cocoa Bindings ([10]) provide a means for keeping Model and View values synchronized. It provides a "binding" such that a change in one is reflected in the other. Cocoa Bindings support straightforward synchronization between View widgets and Model properties (e.g. a preferences

```
<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>
```

Fig. 2. Non-Visual, Imperative Code Accessing Declarative Widgets

editor), but do not support more complex interactions.

In the case of XAML, for example, 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. Widgets must call imperative event-handlers when interesting events occur; the event handler code must access widget state 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 XAML.

Such hybrid approaches resemble *Relational Blocks* in the way that visual or declarative techniques are used to construct the application’s view. Unlike *Relational Blocks,* they suffer from an impedance mismatch caused by the need to mix declarative and imperative programming in the same application. Even worse, the hybrid approaches tends to discourage strict encapsulation. Looking at the XAML 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.

Thus, *Relational Blocks* can be explained as an approach that not only constructs an application's view visually, but that also uses the same visual representation and API to code the model and controller portions of the application.

Relative to other VPLs, it is important to first note that *Relational Blocks* is language that supports *general purpose* programming, in contrast to VPLs intended to support domain-specific tasks such as simulation [2] or programming-by-example [18]. In this sense, *Relational Blocks* resembles VPLs such as VIPR [5] and Prograph [7]. *Relational Blocks*, however, differs from such VPLs in two important ways. First, it is a declarative, rather than an imperative, language; we have discussed this aspect previously in the context of non-visual languages. Second, *Relational Blocks* is not an object-oriented language, although it incorporates ideas from the latter with its emphasis on "encapsulation".

Relational Blocks imposes a strict encapsulation on block primitives (Section II) and on the composite blocks that developers assemble from the primitives. 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 can be replaced with another language without changing the semantics. In fact, *Relational Blocks* can be considered the extension of the object-oriented approach to application assembly, such that current approaches to declarative application assembly are augmented with encapsulation. *Relational Blocks*, however, differs from the classic object-oriented approach in that the base construct is a block whose input and output are *relations* rather than an *object*. There are several reasons for our approach. 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 [9], chapter 2). This is not a problem that is specific to *Relational Blocks*, but affects most object-relational mappings. (See [16] for a brief review of previous efforts in this area.) Note, however, that *Relational Blocks* easily accommodates objects when they are used as attribute (column) values.

In summary, then, *Relational Blocks* is a VPL for the assembly of general-purpose applications, specifically transactional applications with a requirement to integrate non-trivial model, view, and controller portions. It is a strongly declarative language which emphasises encapsulated application design. The most novel aspect of *Relational Blocks* relative to other VPLs is its use of a relational API to integrate application components.

C. Evaluating Relational Blocks

Although our motivation for building *Relational Blocks* is to increase developer productivity, it is very difficult to directly evaluate the effectiveness of this work. Many, many alternatives exist for building enterprise applications: doing controlled experiments to measure the "productivity improvement" metric therefore requires tremendous outlay of resources. The problem is exacerbated because developers already know many of the alternatives to *Relational Blocks*, so that *Relational Blocks* development incurs a learning curve penalty that is difficult to control for properly.

In this paper we therefore use application complexity as a proxy to the productivity metric, with the assumption that less complex applications (of equivalent functionality!) increase productivity by reducing development and maintenance costs. Furthermore, we use application "size" to approximate application complexity, and measure the size of a visual program by the size of its serialized representation. We use CRUD++ (below) as an example of a non-trivial enterprise application that is still small enough to be described in detail within the scope of this paper. As we

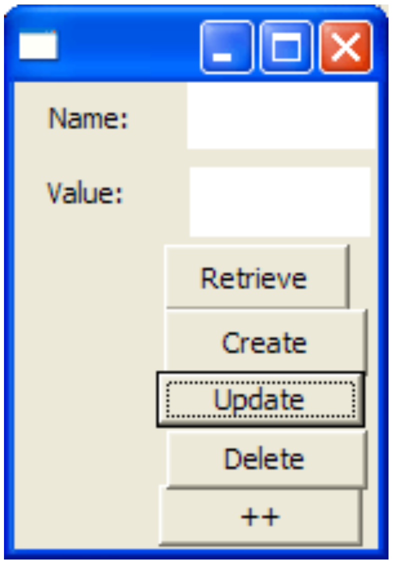


Fig. 3. The CRUD++ Example

show how the visual editor is used to assemble CRUD++ (Section V) we will also compare it to the equivalent Java (used because it is a well-known, non-visual, imperative programming language) rendering.

We use CRUD++ as a sample application to make our discussion more concrete. This application allows users to perform any of the classic "CRUD" (C(reate), R(etrieve), U(pdate), D(elete)) operations applied to a set of $\{Name, Value\}$ associations maintained in a persistent database table. The ability to perform such CRUD operations is a minimal requirement for any approach to building enterprise applications. In addition, CRUD++ allows users to increment a named value when the value is represented as an integer; an attempt to increment non-integer values is invalid, and causes an exception to be displayed to the user. CRUD++ includes this somewhat contrived operation because it requires more controller logic than the other operations. Figure 3 is the GUI displayed to the user. For example, a new $\{Name, Value\}$ association is created by entering a *name* in the Name text box, a *value* in the Value text box, and then pressing the Create button.

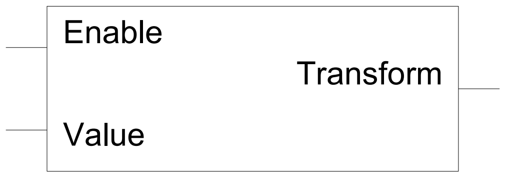


Fig. 4. A Single “Block”

D. Paper Structure

The remainder of this paper explains the *Relational Blocks* language, runtime, and visual editor in detail. Section II introduces the basic language constructs, Section III discusses the runtime, and Section IV explains the more advanced features needed to assemble enterprise applications. With this background in place, Section V is a detailed walk-through of the process of assembling the CRUD++ sample using *Relational Blocks*. Section VI discusses on-going challenges that we are currently addressing to make *Relational Blocks* scale to support larger applications.

II. Basic Language Constructs

Before explaining how an *Relational Blocks* application is assembled, we discuss the (visual) language constructs and their semantics.

A. Block Primitive

Figure 4 shows a single block with three pins: two input pins, named “value” and “enable”; and an output pin, named “transform”. An input pin implies that the block can receive relation-valued input from other blocks. An output pin implies that the block can transmit relational-valued output to other blocks. Model-update blocks have an “enable” pin; these are boolean-valued relations, and Model-update blocks only update if the enable input pin is true.

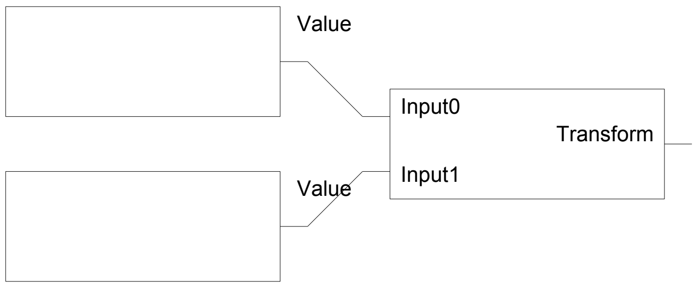


Fig. 5. Blocks and Wires

B. Wires

Figure 5 shows how wires are used to specify relational data-flow between two blocks. Wires connect input pins to output pins: in this Figure, the two blocks on the left – each with an output pin named “value” – transmit data to the rightmost block’s input pins. For more complicated connections, wires may be used like wires in an electrical circuit. That is, any set of pins may be connected by wires, as long as exactly one pin from the set is an output pin. All input pins in the set will receive the data from this output pin. No other restrictions are imposed on the circuit topology. A block’s semantics specify a well-defined transformation from its inputs to its outputs. For example, JOIN is an algebra block that computes the relational AND of two input relations and places the result on its output pin.

C. Primitive Block Types

Relational Blocks contains a set of pre-defined block prototypes, which can be in one of three flavors: *model*, *widgets*, and *algebra*. These correspond, respectively, to the well-known Model/View/Controller paradigm. A salient point of *Relational Blocks* is that *all* block types have a compatible relational API and visual representation. This enables a visual representation of a complete application to correspond to an integrated relational algebra application.

- Model: Visually, a Model block is represented as a mathematical table, as in existing visual tools for relational database design. Model blocks may be backed by persistent or transient state: this is an application design issue; the *Relational Blocks* paradigm makes no distinction between persistent and transient Models. Model blocks have an output, which is the current state of the relation, and an input, which is the desired next state of the relation.

One problem faced by *Relational Blocks* is that the relational algebra by itself does not provide a means to *update* a model block, 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 ([9], Chapter 5, *Tutorial D*). *Relational Blocks* takes a different approach: it treats model blocks as the state of a Relational State Machine. Thus, the execution of the State Machine causes model (and widget) blocks to be updated. The State Machine is typically "clocked" (makes transitions) on GUI events.

- *Widgets*: GUI widgets are 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. A slider might have a single tuple with a single attribute *value* in its output. Note that read/write widgets (e.g., text boxes) contain portions of an application's model since they act as a tiny (transient) database. More complicated widgets such as tables and lists are multi-tuple relations. Widget blocks are thus directly compatible with model and algebra blocks.
- *Algebra*: an application's controller logic is described declaratively using relational algebra. We have chosen a formulation of the relational algebra that is based on *Relational A* ([9], 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 [6]. 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).

Input to an algebra block can include the current state of model blocks or the current values of the readable widgets. An algebra block's output can be the next state of a model block or the next state of a writeable widget block.

D. Composite Blocks

Developers can assemble *composite* blocks, which are arbitrary graphs of the primitive block types discussed above. A composite block can be itself embedded in another composite in a nested paradigm. Composite blocks have exactly the same relational API and visual representation as primitive blocks. The assembled blocks form a directed graph which is maintained by the *Relational Blocks* runtime (Section III). Although the functional (algebra) portions of the graph must be acyclic, cycles are allowed to pass through model and widget blocks, because the cycles are "broken" by the clocked nature of these blocks. Finally, note that an application is simply a self-contained composite block with no input or output pins.

III. RUNTIME

These basic language constructs suffice to explain almost all of the *Relational Blocks* runtime behavior. The static structure of a *Relational Blocks* application graph's state changes only when a “clock tick” (typically a GUI event) occurs. Execution of an *Relational Blocks* application is event-driven, typically by a user interacting with a GUI widget (e.g., clicking the “Create” button). (We have not closely examined alternative event sources such as database triggers.) Every 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, widget blocks, and model block outputs.
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 block outputs will equal this new state.
3. All writeable widget blocks update their state based on their current inputs. Note that the GUI is thus updated synchronously by the event-handler, and asynchronously by the user.

The approach *Relational Blocks* uses for event handling is consistent with its overall declarative approach. In general, most declarative approaches offer weak support for programming actions in response to events. With *Relational Blocks*, event handling is simply a (widget) block output indicating, for example, that a button was clicked or an entry was made to a text field. (Forms/3 [4] takes a similar approach of treating “events as values”.) Events are given special treatment only in the sense that they are recognized to be asynchronous input to the current application state which therefore represent a “clock tick” to the Relational State Machine.

The *Relational Blocks* runtime therefore has a different model of computation from the concept of a *program counter* used in imperative languages. A 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.

A. 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 [14]. A key issue, therefore, is how to transactionally scope an application's activities. Frameworks such as Enterprise JavaBeans [12] 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 applications 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 CRUD++, the developer will not want an illegal value (e.g. "4t2") to be incremented. A typical application design would detect the illegal value and enable the ROLLBACK block. 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.

IV. ADVANCED LANGUAGE FEATURES

We discuss some of the advanced features of the *Relational Blocks* language in this section.

A. Flexibility

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. (This is required by the "increment" function in CRUD++.) Reference [9] (Appendix A) 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. With respect to our example, Table I 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*, which is the desired function result.

The problem with the theoretical approach is that many useful functions require that the constant relation be of infinite size. Thus, in our example, there are an infinite number of strings which express legal integer values. *Relational Blocks* therefore implements the *equivalent* functionality

TABLE I
RELATION TO CONVERT STRING TO INTEGER

value_string	*value_integer*
“0”	0
“1”	1
“2”	2
.	.
.	.
.	.

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.

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 6. 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).

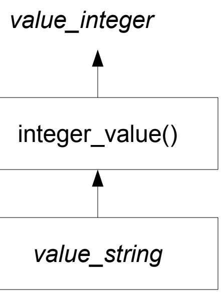


Fig. 6. Expression Tree for *value_integer* = integer_value(*value_string*)

B. 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 INSERT block does not match the relation header of the block's 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 I. Thus a FUNCTION BLOCK which attempts to convert this to an integer will yield the empty relation. Intuitively, it is desirable for CRUD++ 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 [25]. 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 [8] as the output value that signals an exception. Partly because the semantics of the NULL value is overloaded, and for other reasons [9] (Chapter 7), 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 III), an appropriate message is sent to the user, and the application's current transaction (Section A) is rolled back.

V. BUILDING CRUD++ WITH *Relational Blocks*

We have built a prototype implementation of *Relational Blocks* in Java, specifically building the visual editor with the Graphical Editor Framework (GEF) [13], an Eclipse [11] 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.

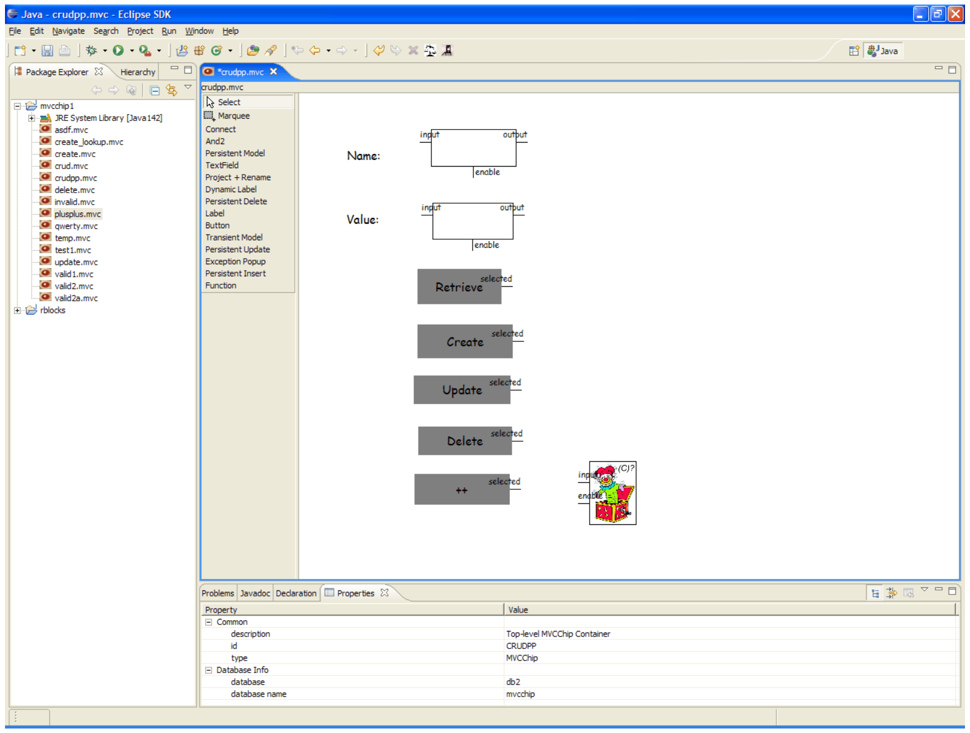
As shown in Figure 7, the editor is structured with the set of available applications on the far left; a palette providing a small set of widget, model, and algebra prototype blocks in the center-left; a property-sheet view on the bottom; and the application-design canvas in the center. The property-view is used to modify properties such as the database name and the table names of a INSERT block. Properties such as function expressions are edited textually, others such as relation headers are edited with a UI dialog. Developers assemble applications by dragging blocks from the palette to 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.

In this section, we show how the CRUD++ sample (Section I.C) is assembled with the prototype. We also compare the *Relational Blocks* version of CRUD++ with one done in Java, a well-known, non-visual, imperative language. Note that the serialized versions of the *Relational Blocks* application are *never* directly manipulated by the developer: they are used only to provide persistence, and we show them only to give a sense of the application size.

Comparing the actual CRUD++ GUI produced by the *Relational Blocks* runtime (Figure 3) to the version sketched in the *Relational Blocks* editor (Figure 7), we see that the prototype is not currently focused on rendering a polished (WYSIWYG) UI. The editor allows users to sketch the GUI 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. One approach, which would also make the sketched version look more realistic is to use the Eclipse *Visual Editor* [20] technology.)

Two text-entry fields are used to input and output the desired *name* and *value*, and are labeled with corresponding labels. The user clicks on one of five buttons to invoke the desired CRUD++ operation. Finally, the developer drags a “popup” widget to the application-design so that the user can be informed when an exception occurs.

Code sample 1 is an abbreviated Java (Swing) program that builds *part* of the CRUD++ GUI. It specifies a “create” button, a “name” label, and a “name” text field, and embeds the widgets in a top-level frame. We contend that (1) the visual approach to building the GUI is far more intuitive than the non-visual approach and (2) the complexity and size of the imperative Java code is bigger than the *Relational Blocks* version (see code sample 2 which is the serialized XML representation).



Property	Value
Common	
description	Top-level MVCChip Container
id	CRUDPP
type	MVCChip
Database Info	
database	db2
database name	mvcchip

Fig. 7. Using the *Relational Blocks* Editor to Sketch an Application GUI

These advantages are actually larger since the actual CRUD++ GUI contains four more buttons, and an additional label and text field.

Note that Label widgets have neither inputs nor outputs. In contrast, Button widgets have a boolean valued output relation which will transmit a TRUE value when the user has clicked the button, and will otherwise be FALSE. In Figure 8 the developer wires these “selected” pins to the “enable” input pin of the corresponding model blocks (e.g., the “create” button enables the “insert” model operation, and if not clicked, will implicitly disable that part of the circuit). Also, because the “increment” button can cause an exception, its output pin is wired to the “popup” widget. The property editor is used to specify the database table that provides persistence to the model blocks.

Code sample 3 shows how a $\{Name, Value\}$ tuple is deleted using the Java APIs. The actual code flows are even more complicated than this, because the database access code must be added to the event-handling code associated with a widget (as in code sample 1). In contrast, the *Relational*

Code Sample 1 Creating Some of the CRUD++ Widgets in Java

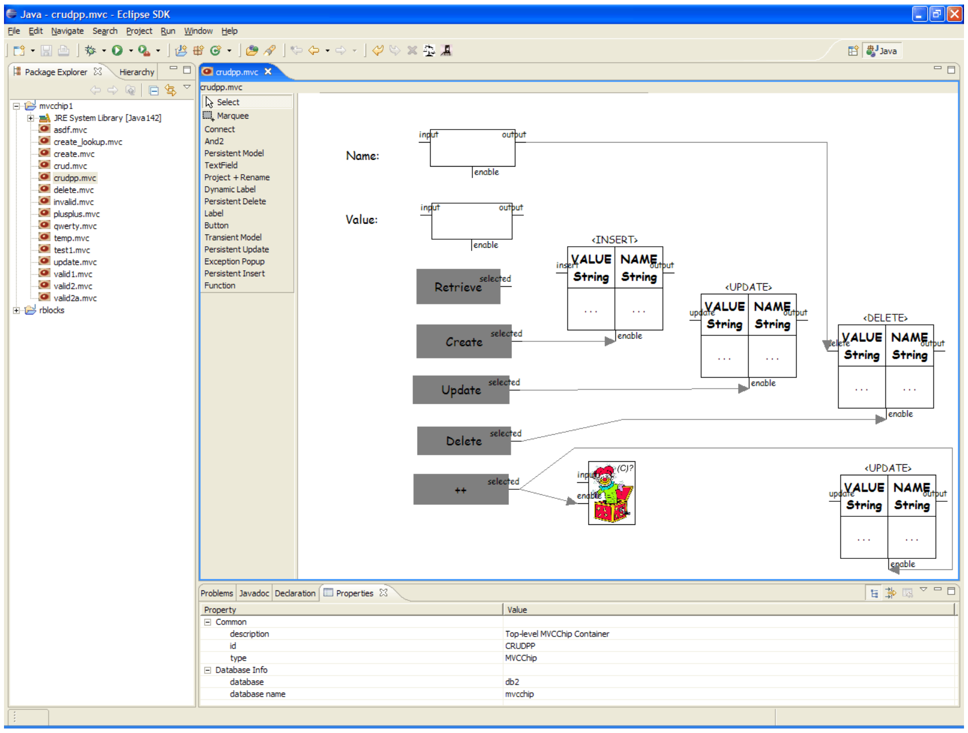
```
JLabel name = new JLabel("Name");
JButton createButton = new JButton("Create");
JTextField nameTextField = new JTextField(20);

// Similar code attached to each of the operation buttons
createButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // Access TextField State: e.g., the nameTextField
      // Do a JDBC "insert" of the "name, value" to the database table
    }
  });

JPanel pane = new JPanel(new GridLayout(0, 1));
pane.add(createButton);
pane.add(name);
pane.setBorder(BorderFactory.createEmptyBorder(30,30,10, 30));
JFrame frame = new JFrame("SwingApplication");

SwingApplication app = new SwingApplication();
Component contents = app.createComponents();
frame.getContentPane().add(contents, BorderLayout.CENTER);

frame.pack(); // Display the window.
frame.setVisible(true);
```

Code Sample 2 Creating the *Relational Blocks* Widgets (Serialized Version)

```
<H id="create_button" type="Button" text="Create">
  <L h="75" w="157" x="143" y="293" />
</H>
<H id="name label" type="Label" text="Name:" i="text" >
  <L h="54" w="109" x="32" y="56" />
</H>
<H id="name textfield" type="TextField" i="text" o="NAME" >
  <L h="80" w="144" x="161" y="33" />
</H>
```

Blocks version (see code sample 4) requires only that the text fields output be wired to the JOIN block that feeds into the INSERT block and that the "delete" button enable the DELETE model block.

Figure 9 is the fully-assembled application. Consider what happens when the user interacts with the GUI. For example, when the "create" button is clicked, the contents of the *name* and *value* text boxes are merged into a single tuple by being fed to a JOIN block. The new tuple is wired to the "insert" input pin of the INSERT block which has also been enabled by the user's action. As a result, a new $\{$*Name, Value*$\}$ association is inserted into the database. Similar flows occur when the



Property	Value
Common	
description	Top-level MVCChip Container
id	CRUDPP
type	MVCChip
Database Info	
database	db2
database name	mvcchip

Fig. 8. Using the *Relational Blocks* Editor to Connect the GUI to the Model

Code Sample 3 Using Java APIs to Delete a $\{Name, Value\}$ Tuple From a Database Table

```
String url = "jdbc:db2:databasename";
Connection connection = DriverManager.getConnection(url, "", "");
Statement stmt = connection.createStatement();
ResultSet rs;

rs = stmt.executeQuery("SELECT VALUE FROM NAME_VALUE WHERE NAME = ’xxxx’");
boolean isRow = rs.next();
rs.deleteRow();
if (rs.next())
  throw new Exception ("More than one value associated with name");

conn.close();
```

Code Sample 4 Deleting a Database Row in *Relational Blocks* (Serialized Version

```
<DBInfo db="db2" name="mvcchip" />
<H id="delete_pmodel" type="Persistent Delete" schema="mvcchip"
   table="crudpp" >
  <L h="-1" w="-1" x="705" y="252" />
  <I inpin="enable" outpin="selected">delete_button</I>
  <I inpin="delete" outpin="output">name textfield</I>
</H>
```

other CRUD buttons are clicked. The most complicated part of the circuit involves the "increment" operation. In the far left, a JOIN block first does a lookup of the *value* that is currently associated with the *name* specified by the user in the text entry widget. The resulting {*Name, Value*} tuple is fed to a FUNCTION BLOCK on the bottom of the circuit which:

1. Attempts to convert the string representation of the *name* attribute into the corresponding integer *via* the *integer* library function.
2. Increments the integer.
3. Converts the integer into the corresponding string *via* the *varchar* library function.
4. Transmits an error message to the "popup" widget if the increment logic generates an exception.

If the "increment" operation succeeds, it outputs a tuple which contains an integer-valued *valuePlus1* attribute in addition to the original {*Name*, *Value*} attributes. By feeding the new tuple to the PROJECT+RENAME block, a tuple with the right {*Name, Value*} relation header can be input to the "update" model block. The original *value* attribute is "projected" out; the original *name* attribute is left "as is"; and the *valuePlus1* is renamed *value*.

In summary, our comparison of *Relational Blocks* to a non-visual, imperative, programming language shows that *Relational Blocks* applications are less complex because:

- they are smaller for GUI portions of the application, and about the same size for other portions
- operations such as database access operations are expressed at a higher level of abstraction
- code that performs one function (e.g., database operations) does not require detailed knowledge of other code (e.g., the widgets event handlers).

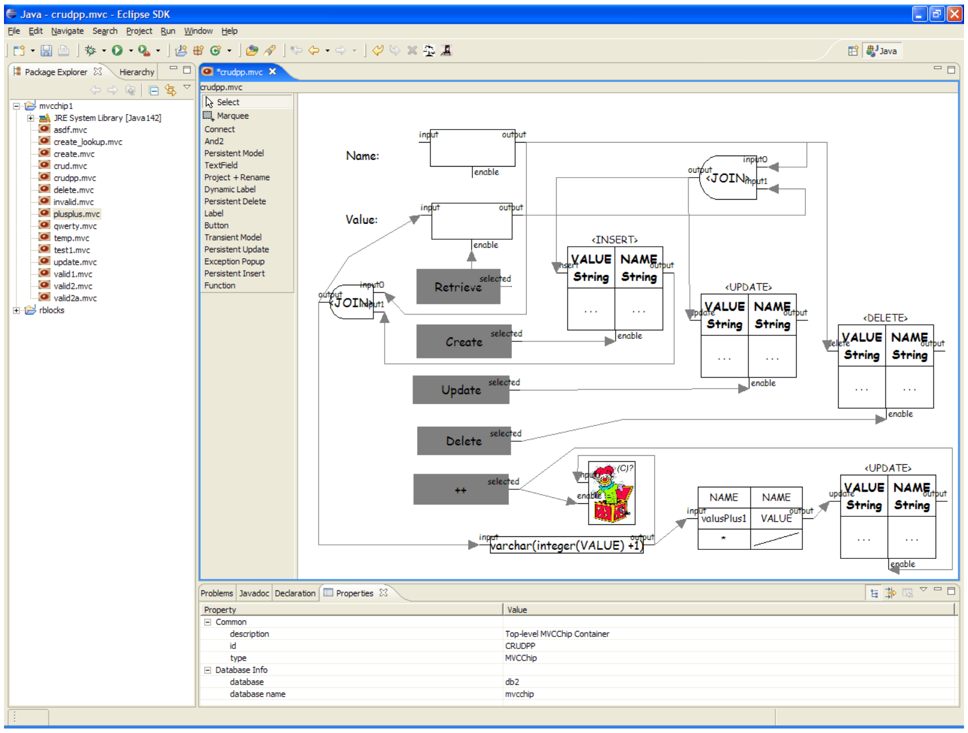
Developers can, at any time, invoke the "validate" or "execute" functions by clicking on the corresponding editor icon. Although the editor enforces some part of the *Relational Blocks* semantics through its property-sheet dialogs and forbidding illegal wiring, this is done only on a

Code Sample 5 Incrementing a Value using Java APIs

```
String url = "jdbc:db2:databasename";
Connection connection = DriverManager.getConnection(url, "", "");
Statement stmt = connection.createStatement();
ResultSet rs;

rs = stmt.executeQuery("SELECT VALUE FROM NAME_VALUE WHERE NAME = ’xxxx’");
boolean isRow = rs.next();
String value = rs.getString("VALUE");
if (rs.next())
  throw Exception ("More than one value associated with name");
try {
  int valueAsInt = Integer.parseInt(value, 10).intValue();
  valueAsInt++;
  String incrementedValue = new String(valueAsInt);
  stmt.executeUpdate("INSERT INTO NAME_VALUE " +
     "VALUES ("’+xxxx+"’, incrementedValue+")");
}
catch (Exception e) {
  throw new Exception ("Problem converting value to int: "+e);
}

conn.close();
```

Code Sample 6 Incrementing a Value Using *Relational Blocks* (Serialized Version)

```
<H id="inc_exception" type="Exception Popup"
   message="Cannot increment a non-numeric value">
  <L h="-1" w="-1" x="372" y="475" />
  <I inpin="enable" outpin="selected">inc_button</I>
  <I inpin="input" outpin="output">inc_value_function</I>
</H>
<H id="inc_value_function" type="Function"
   expression="varchar(integer(VALUE) +1)"
   o="valusPlus1" invalidDomainValue="deadbeef">
  <L h="-1" w="-1" x="241" y="575" />
  <I inpin="input" outpin="output">model_name_join</I>
</H>
<H id="project_rename" type="Project + Rename">
  <L h="-1" w="-1" x="518" y="509" />
  <I inpin="input" outpin="output">inc_value_function</I>
  <ProjectRename>
    <P_R old_name="NAME" new_name="NAME" />
    <P_R old_name="valusPlus1" new_name="VALUE" />
  </ProjectRename>
</H>
<H id="update_valueinc" type="Persistent Update" schema="mvcchip"
   table="crudpp">
  <L h="-1" w="-1" x="708" y="452" />
  <I inpin="update" outpin="output">project_rename</I>
  <I inpin="enable" outpin="selected">inc_button</I>
  <WhereClauseAttributes><Attribute name="NAME" /></WhereClauseAttributes>
</H>
```

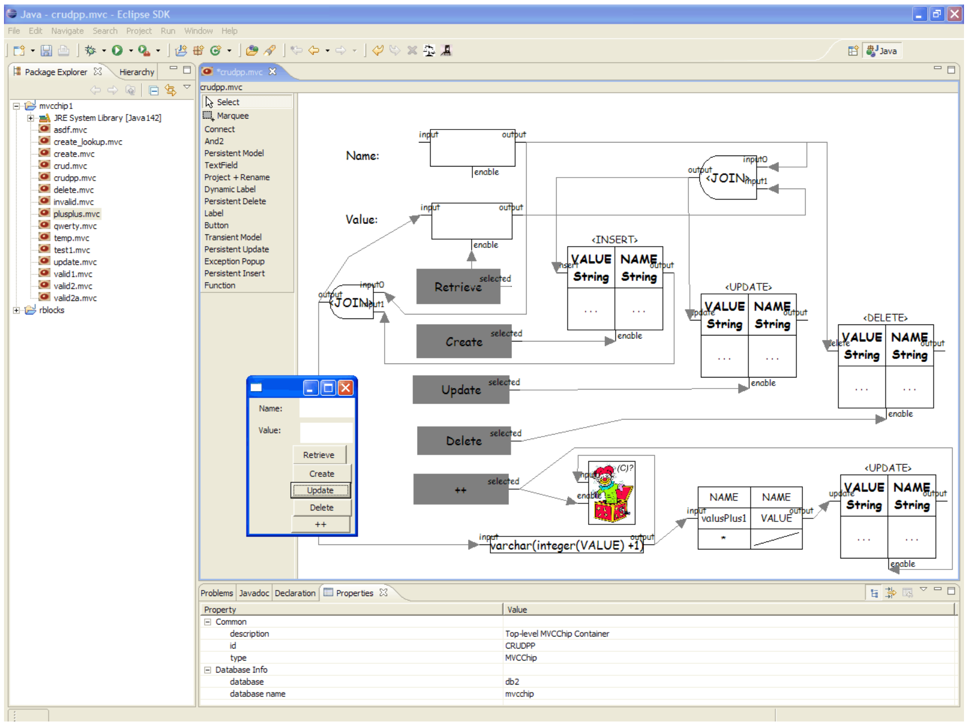


Property	Value
Common	
description	Top-level MVCChip Container
id	CRUDPP
type	MVCChip
Database Info	
database	db2
database name	mvcchip

Fig. 9. Fully Assembled CRUD++ Application in the *Relational Blocks* Editor

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 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 III) 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 dele-

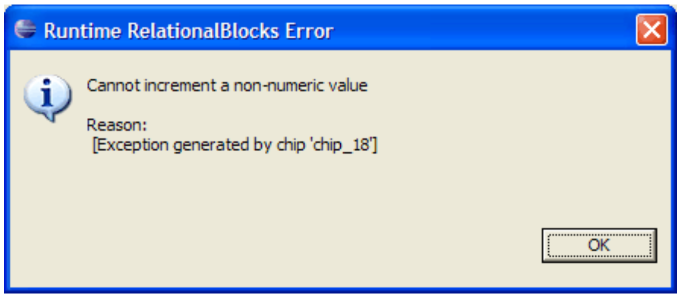


Property	Value
Common	
description	Top-level MVCChip Container
id	CRUDPP
type	MVCChip
Database Info	
database	db2
database name	mvcchip

Fig. 10. Direct Execution of the Application from the *Relational Blocks* Editor

gates 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 the model blocks. At this point, the application is displayed to the user, and can be executed "as is". Figure 10 shows the result (the small overlay window at center-left).

Figure 11 shows what happens if a user attempts to increment a non-integer value. The exception logic (Section B) in the lower portion of Figure 9 detects the error, and enables a popup widget. The runtime then instantiates a popup that displays an appropriate error message to the user.


Runtime RelationalBlocks Error

Cannot increment a non-numeric value

Reason:
[Exception generated by chip 'chip_18']

Fig. 11. Processing a User-Generated Exception

VI. The Scaling Challenge

The *scaling-up* problem for visual languages is defined in [3] as the fact that "making visual languages suitable for solving large programming problems seems to require the very complexities VPLs try to remove or simplify". *Relational Blocks* successfully addresses some of the issues discussed in [3], and is beginning to address other scaling issues.

Relational Blocks provides (in the visual editor) a static representation of the application which facilitates reviewing the application, analyzing it, and explaining it to others. The notion of data and procedural abstraction is also built-in to *Relational Blocks* in the "block and pins" constructs. As discussed above, *Relational Blocks* provides declarative event handling to support interaction with the GUI, and provides static type checking (of relations) with explict (attribute) types of data flowing from an output pin to an input pin. Finally, by (de)serializing to (from) XML, *Relational Blocks* provides data persistence so that an application persists beyond a single editor session.

The single most important scaling issue that *Relational Blocks* must solve relates to the effective use of screen real estate: i.e., how can the visual editor be an effective front-end to a large application? This issue manifests itself in two ways. First, application complexity implies that *Relational Blocks* designs will grow too large to be effectively displayed on a single screen. A complicated GUI which takes up all the screen real estate cannot be presented in the visual editor together with its algebra and model blocks. Second, virtually all significant applications require multi-page navigation., *Relational Blocks* must therefore be extended with language and editor constructs that express the concept of navigation between an application's multiple GUI pages.

We are considering addressing this issue through the composite block construct discussed earlier. Composites introduce the notion of hierarchy into an *Relational Blocks* application: the top-level design of the application can then fit on a single screen, while designers can "drill down" as necessary to lower levels of the design to see more detail. *Relational Blocks* must supply its own set of composite prototypes, and more importantly, enable developers to introduce their own composites as "first-class citizens" of the visual editor.

Finally, we are beginning to consider how designers may integrate *Relational Blocks* applications (or portions of applications) with existing, non-relational-block, components. We prefer an approach in which a thin relational shell is wrapped around these components. This approach seems plausible as many of the visual editor's prototype blocks are similarly wrapped versions of SWT widgets and JDBC artifacts.

The current power and flexibility of the *Relational Blocks* language, editor, and runtime are sufficiently promising to encourage us to solve these challenges.

REFERENCES

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

[2] Alan Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):353 – 387, 1981.

[3] I. Burnett, M.J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. Scaling up visual programming languages. *Computer*, 28:45 – 54, March 1995.

[4] M.M. Burnett and A.L. Ambler. A declarative approach to event-handling in visual program-

ming languages. *Proc. IEEE Workshop on Visual Languages*, pages 34–40, 1992.

[5] Wayne Citrin, Michael Doherty, and Benjamin Zorn. Formal semantics of control in a completely visual programming language. *Proc. Symposium on Visual Languages*, pages 208–215, 1994.

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

[7] P.T. Cox, F.R. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from textual conditioning. *IEEE Workshop on Visual Languages*, pages 150 – 156, 1989.

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

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

[10] James Duncan Davidson. *Learning Cocoa with Objective-C, Second Edition*. O'Reilly, Sebastopol, CA, USA, 2002.

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

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

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

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

[15] 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.

[16] Comparing LINQ and Its Contemporaries. `http://msdn.microsoft.`

com / library / default . asp ? url = /library / en-us / dndotnet / html / linqcomparisons . asp, 2006. T. Neward.

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

[18] B. A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. *Special issue: CHI ’86 Conference Proceedings*, 17(4):59 – 66, 1986.

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

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

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

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

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

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

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