WebRB: Evaluating a Visual Domain-Specific Language For Building Relational Web-Applications

Avraham Leff James T. Rayfield

IBM T.J. Watson Research Center

{avraham,jtray}@us.ibm.com

Abstract

Many web-applications can be characterized as "relational". In this paper we introduce and evaluate *WebRB*, a visual domain-specific language for building such applications. *WebRB* addresses the limitations of the conventional "imperative-embedding" approach typically used to build relational web-applications. We describe the *WebRB* language, present extended examples of its use, and discuss the *WebRB* visual editor, libraries, and runtime. We then evaluate *WebRB* by comparing it to alternative approaches, and demonstrate its effectiveness in building relational web-applications.

Categories and Subject Descriptors D [*1*]: 7—Visual Programming; D [*3*]: 2—Data-flow languages

General Terms Languages

Keywords webrb, web relational blocks, relational web-applications, web-application development, visual programming languages, data-flow languagesR

1. Introduction

Different techniques have been proposed for developing web applications. For example, some design tools exist (e.g., Dreamweaver [1] and IBM Rational Application Developer [2]), that provide a *largely visual* design paradigm. However, for web applications which create dynamic pages and update relational databases, the *imperative-embedding* approach seems to be by far the most popular. In this approach, developers create web pages in an imperative language such as Java or PHP. When the web page is accessed, the web server executes the imperative language code and sends the output to the web-browser client. The imperative code may be written by hand, or automatically generated and hand-tailored (e.g., Ruby on Rails [3]).

The popularity of such text-based approaches is puzzling, since an all visual technique for designing web-pages would seem to be more natural. This paper advocates the use of *WebRB* (Web Relational Blocks), which is unique in being precisely such an all-visual approach, in contrast to imperative-embedding or largely visual approaches.

Consider the *Product Catalog* web-page shown in Figure 1A. The web-page allows an eCommerce site to administer its product catalog through a browser interface. Although simple, it illustrates the following typical pieces of dynamic web-page function:

- A dynamic HTML table is displayed to the user in which each row shows information about a product in an eCommerce catalog. The table is populated by reading data from a relational database table.
- Buttons in each row of the HTML table allows users to modify the relational database table (e.g., to edit or delete a product in the catalog). Another button allows users to add products to the catalog.
- The web-page is linked navigationally to other web-pages in the application so that clicking the "add" button navigates to the "Add Product" page, and clicking the "edit" button navigates to the "Edit Product" page. Moreover, data required by the next page (such as a product's ITEMNO) flow from the current page into the next page.

The *Product Catalog* web-page illustrates properties of a broad class of dynamic web-applications which we characterize as "relational". By this we mean that they:

1. Read relational databases and present the data in a GUI;
2. Update relational databases based on a user's interaction with the GUI;
3. Perform transformations of the relational data which require only simple or moderately complex business logic.

Contrast the imperative-embedding approach for writing dynamic web-pages with the *WebRB* approach. Figure 2 is a fairly typical PHP specification of the *Product Catalog*

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

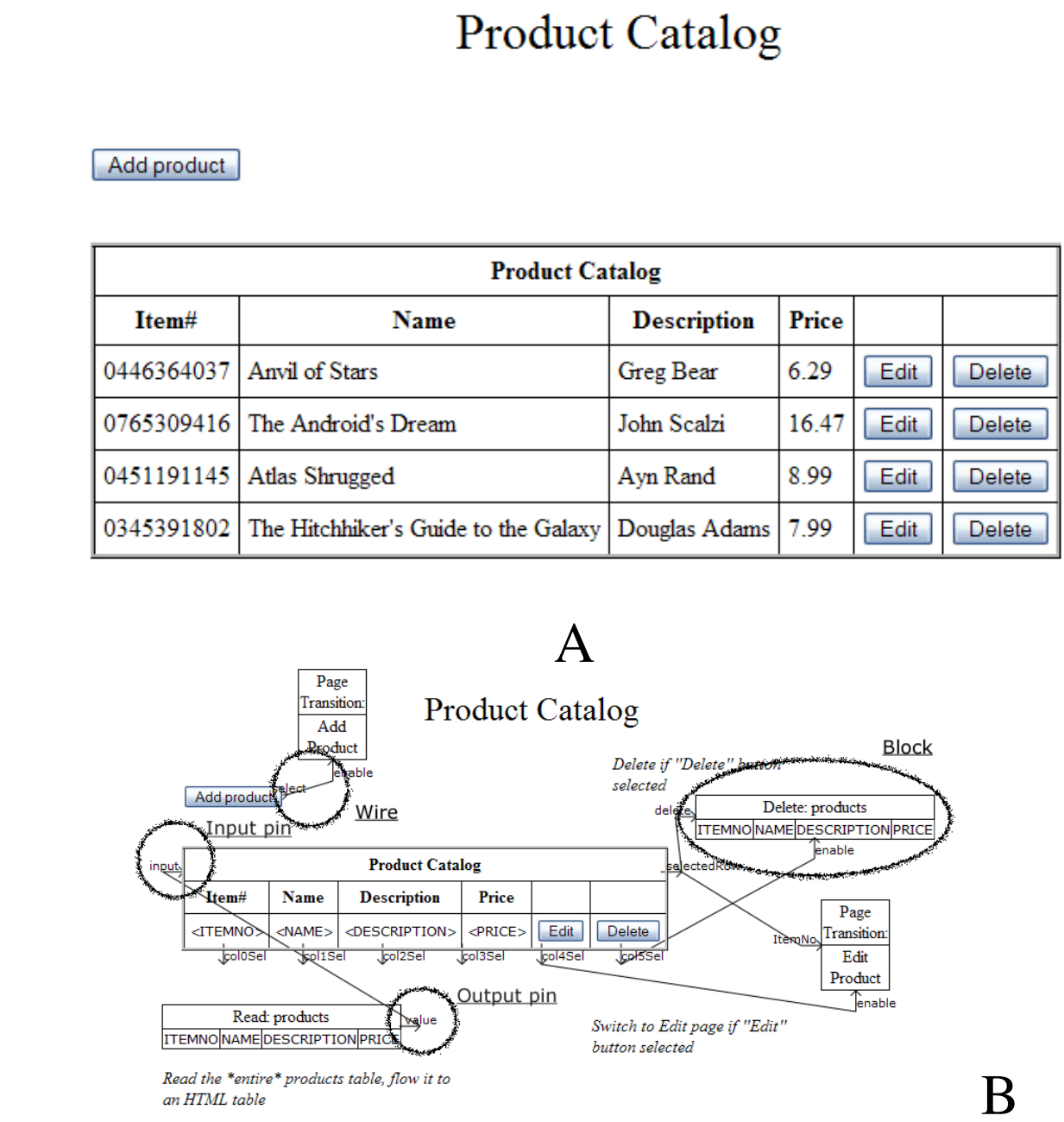


Figure 1. *Product Catalog* Web-Page and *WebRB* Page-Design

web-page; Figure 1B is a *WebRB* page-design of the same web-page. (Here, we are motivating the use of *WebRB*; a detailed set of examples and evaluation is provided later in the paper.) Since PHP pages start in unprocessed mode (typically HTML), the code in Figure 2 immediately switches to PHP mode. Using PHP imperative code, a connection to the database is established, and a SQL[4] query is executed. Next, the page switches back to HTML mode in order to generate the HTML table, then back to PHP to fetch the database data, then back to HTML to generate the contents of the table cells. After the loop, the page reverts back to HTML mode to generate the closing elements. This example illustrates a number of drawbacks with the imperative-embedding approach. First, six language transitions are required in this example, forcing a developer to switch between "model and business logic" mode and "GUI" mode. Second, the visual (HTML) portion of the page must be expressed in a non-visual fashion, requiring the developer to keep a mental image of how the page will look when rendered in a browser. Third, it mixes both imperative (PHP)

and declarative (HTML) styles in close proximity. Finally, the database query must be expressed in SQL, even though a visual query builder might be more appropriate.

WebRB is a visual domain-specific language designed to address the limitations of the imperative-embedding approach for relational web-applications. As shown in Figure 1B, *WebRB* includes the following specific features that make it easy to write such applications:

- Its relational API and dataflow approach strongly support the bi-directional movement of relational data between database tables and the GUI.
- By using relational algebra to transform relational data, developers are able to express business logic in a way that fits their data's semantics.
- As a visual programming language, it allows developers to *visually* construct the application's GUI. Moreover, even the non-GUI portions of the application are written using the same syntax and semantics as the GUI portion. Developers therefore do not have to repeatedly cross between visual and non-visual languages when developing the application.

These features also point to the types of web-applications that *WebRB* is *not* suited for: those with complex business logic (more difficult to express in relational algebra) or those that access non-relational data (need a different API).

In this paper, we substantiate the claim that, compared to the imperative-embedding approach, *WebRB* can significantly improve developer productivity and reduce maintenance costs for relational web-applications. Also, we show that *WebRB* provides sufficient functionality to implement any relational web-application with moderately complex business logic.

Section 2 defines the *WebRB* model of computation and visual language, and compares its approach to related work. We discuss the *WebRB* implementation in Section 3. In Section 4 we present a detailed set of examples to show how *WebRB* is used to construct non-trivial web-pages. The core of the paper is Section 5 in which we evaluate *WebRB* across a number of important language dimensions, and compare it to alternative approaches for developing web-applications. We summarize the contributions of *WebRB* in Section 6.

2. The Language

2.1 Model of Computation

The *WebRB* model of computation closely supports the language's application domain: relational web-applications. From the end-user's perspective, such applications consist of a set of pages, each of which has a GUI through which the user can interact with the application. A *WebRB* developer implements the application with a set of visual page-designs (such as Figure 1) that – in addition to specifying the GUI seen by the end-user – specify the business logic and database access that respond to user interactions. The *WebRB* model of computation is therefore "page-based", and differs from typical GUI event-handlers [5] in that response logic is specified by the page-design itself rather than through code that is attached to individual GUI widgets. Specifically, the event-loop for processing page P_j consists of the following steps:

0. Render P_j's GUI on the end-user's device (e.g., as a page displayed in a web-browser).
1. Wait for the user to interact with the GUI.
2. Process a user-interaction by executing P_j's declarative specification which describes (through the semantics of its blocks and the block data-flow):
   (a) What (and how) data should be transformed, and which persistent database tables should be accessed and updated.
   (b) How to select the application's next page (e.g., P_k). Note that the next page may be the same as the current page (i.e. $k = j$).
3. Repeat from step 0, using P_k.

We next explain how the behavior of an individual page-design is specified by *WebRB*'s visual syntax and by *WebRB*'s relational API.

2.2 Language Elements

A *WebRB* **page-design** is a named set of blocks that have been assembled by a developer, such as the one shown in Figure 1. Page-designs are constructed using the following visual elements:

- A **block** is a component that provides a well-defined relational function. Blocks may optionally receive input and/or produce output; such inputs or outputs may consist only of relational data.
- A block indicates that it receives relational input (or output) with an *input* (or *output*) **pin**. Pins are thus a named I/O port, and blocks can have zero or more input (or output) pins.
- Data flow between blocks is represented as a **wire** that connects a block's input pin to a block's output pin.

  Examining Figure 1, we see a wire transmitting an *n-tuple* relation (each containing four attributes) connecting the READ DB block's output pin to the HTML-TABLE block's input pin.

A *WebRB* implementation is responsible for providing developers with a set of useful blocks. The only requirement is that all blocks use a relational API. The number of input and output pins are arbitrary and may vary from block to block. A block's API can further constrain its input's relation header in terms of what attribute names or types are legal, or it can constrain the cardinality of a valid relation.

```
<?php // switch to PHP mode
try {
  $db =  db2_pconnect(RB_DATABASE, RB_USERID, RB_PASSWORD,
                      array("autocommit" => DB2_AUTOCOMMIT_OFF));
  $allQuery = db2_prepare($db, 'select * from ' . products);
  if ($allQuery === false) throw new Exception('db2_prepare failed: ' . db2_conn_errormsg($db));
  $success = db2_execute($allQuery);
  if (!$success) throw new Exception("db2_execute failed: " . db2_stmt_errormsg($allQuery));
  ?> <!-- switch to HTML mode -->
  <html>
    <head><link rel="StyleSheet" href="style.css" type="text/css"></head>
    <form method="get" action="addproduct.php">
      <div align="center" style="font-size: 24pt">Product Catalog</div>
      <button>Add product</button>
    </form>
    <form method="post" action="editdelete.php">
      <table>
        <thead>
          <tr><th colspan=10 align="center">Product Catalog</th></tr>
          <tr>
            <th>Item&#35;</th><th>Name</th><th>Description</th><th>Price</th><th></th><th></th>
          </tr>
        </thead>
        <tbody>
        <?php // switch to PHP mode to fetch data
          for (;;) {
            $row = db2_fetch_row($allQuery);
            if ($row === false) break;

            // Output HTML with embedded data
            $itemNo = htmlentities(db2_result($allQuery, 'ITEMNO'));
            print '<tr><td>' . htmlentities($itemNo). '</td>';
            print '    <td>' . htmlentities(db2_result($allQuery, 'NAME')) .  '</td>';
            print '    <td>' . htmlentities(db2_result($allQuery, 'DESCRIPTION')) . '</td>';
            print '    <td>' . htmlentities(db2_result($allQuery, 'PRICE')) . '</td>';
            print '<td><button name="editItemNo" value="' . $itemNo . '">Edit</button></td>';
            print '<td><button name="delItemNo" value="' .$itemNo. '">Delete</button></td></tr>';
          }
        ?> <!-- back to HTML mode for closing tags -->
    </tbody></table></form>
  </html>
  <?php // back to PHP mode for exception handling
}
catch (Exception $e) {
  if (isset($db)) db2_rollback($db);
  print 'EXCEPTION: ' . $e->__toString();
}
?> <!-- Close out PHP code -->
```

Figure 2. Product Catalog, PHP version

For example, by looking at Figure 1, we see that BUTTON blocks have no input pins, since they are used only for user input. In contrast, an HTML-TABLE block has one input pin which accepts a relation of any type, and whose data is used to populate the table. A BUTTON block has a single output pin which transmits a single boolean relation (see [6], TABLE_DEE) whose value indicates whether or not the button was clicked. In contrast, an HTML-TABLE block has multiple output pins: one that transmits a generic relation consisting of the tuple selected by the user's interaction, and a set of pins that transmit the individual attributes of the tuple selected by the user.

Once a developer assembles blocks as a page-design, that design becomes a "first-class" citizen of the language. From *WebRB*'s perspective, page-designs *are* blocks. The only difference is that built-in block types are pre-assembled by a *WebRB* implementation, whereas pages are assembled by a developer. Because they must be assembled, pages are the only construct that can be edited or validated using a *WebRB* visual editor. In the context of the visual editor, we refer to a page *design*. Once assembled, a page is represented using the "blocks, pins, and wires" metaphor, and behaves just like a pre-assembled block. In that context, we refer to a page *block*. The semantics of a page-design are self-contained since developers do not add any non-visual code. Pages can therefore be directly executed by the *WebRB* runtime, for example, as web-pages rendered in a browser and linked to data and control logic on the server. In the context of page execution, we refer to a page-design as an *application* page.

A page-design A can use a page block B in two ways:

- As a page that is directly embedded into A's page-design and application page. In this case, B is represented with an EMBEDDED PAGE block. EMBEDDED PAGE blocks allow developers to construct libraries of reusable page-designs such that a page can be nested in other pages as a custom block. Alternatively, a developer may re-factor

a complicated (visually "busy") page-design into a page containing EMBEDDED PAGE blocks.

- As a possible "next" page when A executes. In this case, B is represented with a PAGE TRANSITION block, to which a developer wires relational logic that specifies when the application will transition from A to B's application page (see Section 2.1).

The *Search* page-design of Figure 9 illustrates these two ways that page blocks are used. The "buy" button in the HTML table is wired to an embedded "buy" page block (Figure 12) that encapsulates the processing of purchasing a single line-item. Using PAGE TRANSITION blocks, the *Search* page-design specifies that a successful buy operation navigates to the *Shopping Cart* page (Figure 11). The *Search* page-design also specifies that a "search" operation refreshes the *Search* page, but populates the TEXT-ENTRY field with the user-supplied search string.

In order for pages (supplied by a developer) to behave exactly like pre-assembled blocks, there must be a mechanism to specify a page's input and output API. This is done with PAGE INPUT and PAGE OUTPUT blocks. A PAGE INPUT block passes data (unchanged) from an input pin on a embedded or PAGE TRANSITION block into the page-design in which the PAGE INPUT block appears. Likewise, a PAGE OUTPUT block passes data (unchanged) from the page design in which it appears to the output pin of an EMBEDDED PAGE block. Thus, PAGE INPUT blocks are analogous to a function's formal input parameters, and PAGE OUTPUT blocks are analogous to a function's formal output parameters. These blocks are represented as named input (output) pins on the page's corresponding EMBEDDED PAGE block or PAGE TRANSITION blocks.

PAGE INPUT and PAGE OUTPUT blocks are illustrated by the *Buy* page-design shown in Figure 12. There are two PAGE INPUT blocks: *ItemNoQty* supplies a two-attribute, single-tuple, relation consisting of the selected product's name and the quantity purchased; *buy* supplies a boolean relation indicating whether to initiate the buy processing. There are three PAGE OUTPUT blocks: *errorMsg* provides an error message string (as a single-tuple single-attribute relation); *success* is a boolean relation which is TRUE if the buy processing was successful; and *error* is a boolean relation which is TRUE if an error occurred.

2.3 Refined Model of Computation

The semantics of database and widget blocks require that we refine Step 2 of the *WebRB* model of computation described in Section 2.1. We do this characterizing pre-assembled blocks along two axes: (1) whether they are stateless or stateful; and (2) whether they appear in the GUI that is seen by the end-user.

Stateless, non-GUI, blocks are the simplest to understand. They are purely functional, in the sense that block output depends completely on the current inputs to the block. For example, the output of a JOIN block is always the natural join of the inputs. Such blocks have attractive properties: they are easily composed, and reasoning about their behavior is straightforward ([7], Chapter 3). Also, all their dependencies can be clearly seen in the visual representation.

Stateless GUI blocks are also fairly simple: they create a visible GUI artifact based on their current inputs. For example, a TEXT block generates a text label in the GUI, and the contents of that label are based on the block's input at the time that the GUI's widget is generated.

Stateful blocks are less straightforward. However, they are vitally necessary, because useful applications require the ability to enter, preserve, and modify persistent information. Even pure functional languages such as Haskell are forced to accommodate this requirement [8]. The *WebRB* model of computation incorporates stateful blocks by treating a page-design as a sequential synchronous circuit. As a result, step 2 above executes as a series of sub-steps that describe how block's inputs and its current state S_i interact to create the block's outputs and next state S_{i+1}.

Stateful GUI blocks (e.g., TEXT-ENTRY) are displayed in the GUI using the current value of their input(s). The user's interaction may alter the value of the block, e.g., by typing into the text-entry field. When the user interaction is processed, the output value(s) of the block are the last value(s) seen by the user: i.e., either the value that was initially displayed, or the value that was entered by the user. Thus, stateful GUI blocks can retain state during the user-interaction period, but may also be modified by the user to contain new state.

Stateful non-GUI blocks are used to access the database, and encapsulate the state of a database table. READ DB blocks have an output pin whose value is the current contents of the specified database table, and do not have any input pins. Mutator blocks (INSERT DB, UPDATE DB, and DELETE DB) make changes to the database. These blocks have two input pins: an enable pin, which enables or disables the block, and another pin which accepts the tuple(s) to be inserted, updated, or deleted. Mutator blocks have no output pins. During event processing, the database state transitions from the initial state S_i to the final state S_{i+1}.

WebRB allows multiple instances of READ DB and mutator blocks to reference the same database table. This is visual "syntactic sugar"; otherwise *WebRB* would have to define a CRUD block (one per table per page), combining the function of READ DB, INSERT DB, UPDATE DB, and DELETE DB. However, the *WebRB* runtime only allows one mutator block (per user) to be enabled at any given time. Otherwise the developer would need to specify ordering rules for updates.

Step 2 in Section 2.1 incorporates these various types of pre-assembled blocks in a refined event-processing algorithm that executes as follows:

1. All enabled mutator blocks evaluate their inputs (including "enable"). This recursively causes all the blocks

which feed those inputs to evaluate their inputs, and so on. The recursion terminates at stateful blocks which have a known output value (stateful GUI blocks and READ DB blocks). Stateful GUI block outputs correspond to the values that the user saw in the GUI, and READ DB block outputs correspond to the initial state of the database, S_i.

2. All mutator blocks simultaneously change the database state (according to the inputs saved in the previous step) to the new state S_{i+1}, by inserting, updating, or deleting table records as appropriate.

3. The next application page is selected. This is done by evaluating the inputs to the "enable" input pin of all PAGE TRANSITION blocks in the current page. Although "enable" is a reserved pin name, the semantics of this input pin evaluation are the same as other pin evaluations. A page is selected as the application's next page only if the enable pin's input is TRUE. In the *Shopping Cart* page-design (Figure 11), for example, the *Search* page is selected if the user initiates a search by clicking the "go" button. Although a page-design may contain any number of PAGE TRANSITION blocks, at runtime, only zero or one of them may be enabled simultaneously. If zero PAGE TRANSITION blocks are enabled, no page transition occurs (the next page is the same as the current page, i.e. $k = j$). It is a runtime error for two or more PAGE TRANSITION blocks to be simultaneously enabled.

4. The GUI for the next application page is generated. All GUI blocks on the page (stateful and stateless) evaluate their input pin(s). This ultimately evaluates the blocks which feed the GUI element inputs. As before, the recursion terminates at stateful blocks which have a known output value (stateful GUI blocks and READ DB blocks). Stateful GUI block outputs still correspond to the values that the user saw in the current GUI page, but READ DB block outputs now correspond to the new database state S_{i+1}.

5. The next GUI page is returned to the user for display.

2.4 Concurrency

Since relational applications are almost always multi-user, *WebRB* must support concurrent execution of its applications. The shared application-state in *WebRB* consists of the database tables which are accessed by the READ DB and mutator blocks. *WebRB* relies on the database manager to provide *ACID* [9] properties for the database tables. At the beginning of the refined event-processing algorithm (Section 2.3), a database transaction is started. Before the next GUI page is returned to the user for display (step 5), the database transaction is committed. If any runtime errors are detected by the *WebRB* runtime (e.g., for the same user, multiple mutator blocks are simultaneously enabled, or two or more PAGE TRANSITION blocks are simultaneously enabled), the database transaction is rolled-back. If they wish, developers can explicitly roll-back transactions by disabling all mutator blocks when an error is detected. We do this in Figure 12, in which a failed string-to-integer disables the INSERT DB block for the CART table.

The *WebRB* approach of coupling transaction scope to the event loop is a compromise between requiring an application to handle transaction-scoping explicitly, and providing middleware [10] which supports long-running transactions (those which span multiple user events). Future *WebRB* work will explore relaxing the "transaction-scope = event" constraint.

2.5 Security

With respect to access control to application page-designs or data, *WebRB* does not currently distinguish between the developer role (the person coding a page-design) and the end-user (the person executing the assembled application). *WebRB* implements access control by providing authenticated users of the system with a virtual database partition that stores page-designs and application data. However, a simple model is used in which one user (whether a developer of end-user) cannot access another user's database partition.

Obviously this design is unsuitable for a production environment. In such an environment, *WebRB* will have to distinguish the developer role from the end-user role and provide each role with different access rights. Developers would need new blocks to support access control. *WebRB* will also have to enable collaboration (both read and write access) between developers by providing function similar to CVS or SVN. We have not done any design work in this area with regard to *WebRB*.

With regard to an application's runtime security, *WebRB* is generally not susceptible to some of the attacks which are effective on contemporary systems. For example, SQL injection attacks are not effective because there is no way to specify a design which routes user input to the SQL parser. SQL cannot be directly entered into *WebRB* designs, and internal use of SQL always uses parameterized prepared statements. Similarly, *WebRB* is generally not vulnerable to cross-site scripting attacks, because there is no way to have a design which causes user-supplied or database-supplied data to be interpreted as HTML. TEXT-ENTRY, TEXT-LABEL, and HTML-TABLE widgets automatically escape all input data to prevent it from being interpreted by the browser.

2.6 Related Work

The task of easing the development of multi-page applications that integrate relational data, GUIs, and business logic has been approached from many directions. *WebRB* provides an integrated, visual, end-to-end approach for building relational applications, which eliminates the need for developers to cross between visual and non-visual languages. Although many mixed (visual and non-visual) development environments exist, the bulk of this paper focuses on a comparison

between *WebRB* and non-visual imperative-embedded approaches. We do so because imperative-embedding also provides a consistent language paradigm (all non-visual), and has enduring popularity among the developer community. In this section, however, we discuss the relationship of various components of the *WebRB* approach – such as visual programming, visual editors, and integration of relational data – to other work in this area.

2.6.1 Visual Programming Languages

The idea that visual programming languages (VPLs) should be used to improve developer productivity (relative to text-based languages) is well-known. Some VPLs, such as VIPR [11] and Prograph [12], have attempted to provide visual imperative programming languages. From a language perspective, *WebRB* differs from such VPLs because it is declarative (rather than imperative), and is relation-based (rather than object-oriented). More fundamentally, unlike *WebRB* which is a domain-specific language, these languages are general purpose and therefore do not provide built-in support for databases access or for constructing GUIs. Rather, such languages are intended to replace languages such as C. Other languages and systems do support visual GUI construction. Forms/3 [13] allows construction of screens based on function expressions. However, it does not support database access, and does not directly support relation types. Also, Forms/3 functions are expressed textually, not visually. Cocoa [14] supports simple GUI construction, but again does not support database access.

2.6.2 Visual Editors

Many visual editors exist which allow the design of static web pages, in WYSIWYG fashion. This is an attractive approach, because it provides a very close mapping between the design-time view and the run-time view. However, it is difficult to extend such editors to support dynamic pages, because of the lack of appealing visual languages which can handle the business logic ("controller") portions of the application.

Dreamweaver [1] is an example of a visual editor for static web pages that also provides some extensions for dynamic web pages. For example, it allows developers to attach non-visual imperative code (e.g., JavaScript) to web pages, to generate HTML tables from queries on database tables, and to generate HTML forms that update database tables. However, it allows only forms-based business logic to be specified visually, and forces developers to escape into non-visual languages when they need to express logic beyond this scope.

JavaServer Faces [15] (JSF) is an alternative technology for specifying web pages, and a variety of visual editors (e.g., [16]) allow developers to construct web pages using a mix of WYSIWYG and non-visual techniques. JSF provides (server-side) user-interface components, such as text-input boxes, tables, and buttons, which are rendered to the browser using pluggable rendering code. The web page is described by a set of JSF tags and XML configuration. Although the (initial) page layout can be done visually, and inter-page navigation can also be specified visually, existing JSF visual editors do not provide a visual mechanism for handling the dynamic portions of the page. The visual editors instead provide a linkage with (non-visual) Java code ("managed beans"), which reads and updates the JSF components.

2.6.3 Visual Access To Relational Data

Other languages, like *WebRB*, directly support visual access to relational databases, including Query-by-Example [17] which allows non-programmers to specify database queries and updates by filling out templates describing the desired operations. Visual query systems [18] provide read access and presentation services for relational database systems using a visual programming paradigm. Database modeling software such as DDS-Pro [19] support visual design of tables and queries. Unlike *WebRB*, however, none of these tools provides full support for the visual construction of relational applications. Like Cocoa [14], most do not support updates to the database – other than very simple property-editor GUIs.

2.6.4 Object-Relational Mapping

Object-Relational Mapping (ORM) technologies address the task of integrating relational data into applications by treating such data as objects that can be used in a general-purpose object-oriented language. Typically tooling (either visual or XML-based) is used to create a mapping between a row in a database table and an instance of an object. Examples of such technology include Hibernate [20] and EJB container-managed persistence [21]. In ORM approaches, objects are the first-class construct used to build applications.

WebRB, in contrast, is motivated by the observation that for relational applications – in which relational data is heavily accessed and manipulated – the most natural approach treats relations (tables) as the first-class construct used to build applications. The relational model is based on set theory, in which each relation represents a set, and each tuple (row) value indicates the presence of that value in the set. Attribute (column) values are drawn from a specific type, such as integer, string, date, etc. Row values indicate a relationship between multiple attribute values, and thus do not themselves naturally form a type system (see [6], chapter 2). In *WebRB*, objects are attribute values rather than the tuple values of an ORM system.

WebRB borrows many concepts from object-oriented programming, including encapsulation and separation of interface from implementation. Blocks have a defined interface (pins), and are only accessible via that interface. The implementation is unspecified and not available from outside the block. In addition, attribute values in *WebRB* are handled very much like object instances, and a natural extension of *WebRB* would be to support developer-defined

types (classes) for attributes. Developer-defined types could then be mapped onto relational columns, and FUNCTION blocks (Section 4.3) would enable operations to be invoked on developer-defined types.

2.6.5 Summary

To summarize, many tools, languages, and systems use visual techniques to improve developer productivity for constructing *portions* of an application. *WebRB*, however, uses visual techniques for *all* parts of an application. Its ability to do so stems, in large part, because *WebRB* is a domain-specific language for relational web-applications, and does not support programming of other types of applications. Thus, by limiting its scope, *WebRB* can use a data-flow approach which naturally expresses the movement of relational data between databases and the GUI with a small set of visual metaphors. By using relational algebra, transformations of relational data are done in the same semantic space – and with the same visual metaphors – as the data itself. Finally, the *WebRB* model of computation and runtime naturally supports the deployment of multiple page-designs as web-applications containing multiple web-pages that run in a standard web-browser.

3. Implementation

WebRB [22] also includes:

- a visual editor whose palette of pre-assembled blocks provides the functionality used to typically develop the simpler sort of web-applications. The visual editor allows developers to directly "code" *WebRB* page designs using only the language's visual syntax without additional constructs.
- a runtime which deploys *WebRB* page designs as an application's web-pages that execute in a standard browser. It also manages an application's inter-page navigation and associated data-flows. Finally, the runtime validates page designs and reports errors *via* the visual editor.

3.1 Visual Editor

The *WebRB* editor is written in JavaScript (about 10,000 lines of code), runs in a standard Mozilla Firefox browser, and is used to visually construct *WebRB* programs. Because *WebRB* is a visual language, there is a good fit between the editor and the programs that it constructs.

Figure 3 shows that the *WebRB* editor is comprised of several frames. A page-design (corresponding to a single web-page) is assembled in the *page editor* frame. As a named page-design, it will be listed in the set of EMBEDDED PAGE blocks and PAGE TRANSITION blocks that are available to the developer. The *palette* frame contains the set of pre-assembled blocks from which a developer *drags and drops* blocks when assembling a page design in the *page editor* frame. The palette can be thought of as a block "factory" that produces generic block instances. The set of pre-assembled blocks will be discussed as we walk-through the process of using *WebRB* to construct web-pages in Section 4. If a given block instance is selected, a *property editor* for that instance is displayed in the bottom frame of the visual editor. This allows developers to view and customize the block's behavior (e.g., rename input or output attributes). The "blocks, pins, and wires" visual metaphor is directly supported because the pin names of all block instances (including page-blocks) are displayed in the page-design. Wires are drawn by *left-clicking* near an given pin, dragging the mouse to the other pin while holding the left mouse-button down, and then releasing the mouse button. Wires can be deleted by selecting them and hitting the *delete* key. Blocks (and all of their attached wires) are deleted by selecting them and hitting the *delete* key.

The *WebRB* editor provides special support for constructing a web-application's GUI. Developers can move all of a page-design's blocks to specific "x, y" coordinates on the *page editor* frame. When a page-design is rendered as a web-page, its widget blocks (e.g., TEXT-ENTRY or HTML-TABLE) are rendered as corresponding GUI widgets at precisely those coordinates on the web-page. This approach is much easier than using imperative code to draw and position the widgets.

The *WebRB* editor reduces the length of the "code, test, and debug" development cycle because applications can be directly executed from the editor itself. 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. Any page may be immediately validated and executed.

3.2 Runtime

The *WebRB* runtime is written in PHP (approximately 15,000 lines of code), and is hosted on an Apache web-server, using IBM DB2 as the database storing the page designs and application data. The *WebRB* runtime is responsible for deploying *WebRB* page-designs as web-applications that execute in a standard browser. We now explain how our runtime can do this without requiring developers to add imperative code on either the client or the server.

The server does not have to directly manipulate the *WebRB* visual language because page-designs are stored in their serialized XML representation. The XML representation serves as a canonical form of a given block: it's instantiated as an PHP object on the server, and as a JavaScript object in the browser. The *WebRB* runtime renders a *WebRB* page by creating server-side versions of each *WebRB* "block". Model blocks are implemented as a wrapping of connections to database tables; algebra blocks are implemented to provide the required algebra function; and widget blocks are implemented so that their `getHTML()` method produces the corresponding HTML that can be rendered in a web-browser. Block inter-connection is implemented by driving the spec-

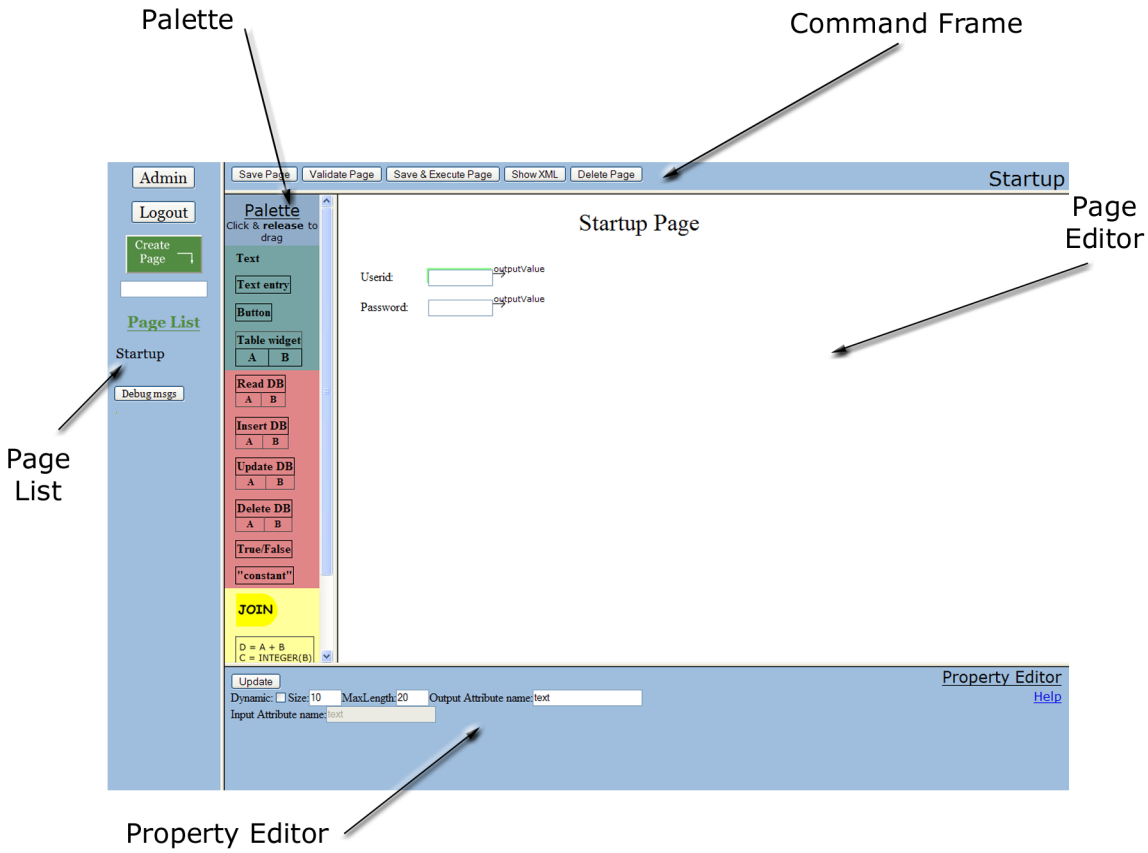


Figure 3. The *WebRB* Editor

ified data-flow from output pins to their input pins, and so on, using a recursive process.

The initial page of an application is displayed when the client's browser issues an HTTP GET request: the server's *WebRB* runtime calls `getHTML()` to create the initial page, and returns the web-page to the browser. During initial-page evaluation, widgets with output pins will return empty relations (since the user has not yet had an opportunity to enter any data). Once launched, the server "forgets" about the page until the user interacts with the web-page, causing an HTTP POST of the page and its data to the server. The *WebRB* runtime uses the POST data to instantiate the corresponding server-side version of that web-page, provides the user-supplied data (e.g., text-field input) to the widget blocks, and evaluates the resulting relational "circuit". The runtime must then determine which web-page should be instantiated next.

4. Examples

The *Product Catalog* example (Figure 1) and the discussion of the *WebRB* language (Section 2) show that *WebRB theoretically* addresses many of the scaling difficulties that have traditionally been associated with visual languages [23]. *WebRB* can be used to develop large-scale applications, since developers can scale an application in two ways: "hierarchical" scaling and "horizontal" scaling. Hierarchical scaling involves using EMBEDDED PAGE blocks to encapsulate function, both to reduce page complexity and enable re-use of function. Horizontal scaling involves partitioning an application into multiple pages, and using PAGE TRANSITION blocks to specify inter-page navigation and data flow.

In this section we'll use a set of examples to show that *WebRB* implementation (Section 3) is powerful enough to easily develop relational web-applications with significant function. The examples will incrementally introduce *WebRB* capabilities to enable us to highlight specific pieces of function. The page-designs are screen-shots of the *WebRB* visual editor [22], and the web-pages are screen-shots of the corresponding web-application executing in a web-browser.

4.1 "CRUD" Function

The examples in this sub-section allow an eCommerce site to administer a product-catalog: we use them to show how *WebRB* makes it easy to construct web-pages that perform the well-known Create, Retrieve, Update, and Delete (CRUD)

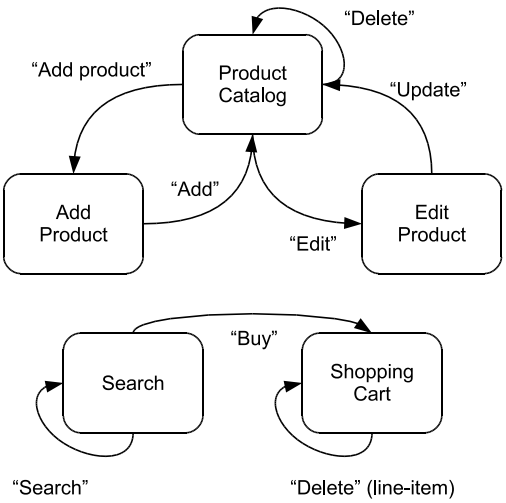


Figure 4. eCommerce Site Page-Transition Diagram

operations. The page-transition diagram of the site is shown in Figure 4. There are two groups of pages: product catalog administration, and shopping pages. We already introduced the *Product Catalog* page (Figure 1) in Section 1. It displays the current contents of the product catalog. "Add" and "Edit" buttons provide links to the *Add Product* (Figure 6) and *Edit Product* (Figure 7) pages, respectively. The *Search* page (Figure 9) is used to search the product catalog. If a product is purchased with the "Buy" button, the *Shopping Cart* page (Figure 11) is displayed.

The core of the *Product Catalog* page is a "read" operation in which a READ DB block flows the product catalog – the entire relation stored in the PRODUCTS database table – into an HTML-TABLE block. The developer uses the property editor (Figure 5) to specify that, in addition to displaying all columns of the database table, each row of the HTML table should include "edit" and "delete" buttons. These and the "Add product" button drive "write" operations from the GUI to the database table.

The Figure also shows how the *WebRB* developer specifies event-handling among a set of possible user-specified actions. The display of a new page after a user input occurs is controlled by PAGE TRANSITION blocks (Section 2) such as the "Add Product" and "Edit Product" blocks. A PAGE TRANSITION block has an "enable" pin which, if TRUE, causes a transition to the specified page. They also have any number of input pins, one for each PAGE INPUT block on the target page, that specify inter-page data-flow. Although the *Add Product* page-design does not have any PAGE INPUT blocks, the *Edit Product* page-design includes a PAGE INPUT block that flows the ITEMNO data into the page. Thus, clicking the "Edit" button brings up the *Edit Product* web-page, allowing the selected product to be updated. The page navigation is specified by wiring the boolean-relation output of the button to the boolean-relation "enable" input pin of the "Edit Product" PAGE TRANSITION block. (An unselected button emits the FALSE relation; a selected button emits the TRUE relation.) Data-flow to the *Edit Product* web-page is specified by wiring the HTML-TABLE's "selectedRow" output pin from the HTML-TABLE to the "ItemNo" input pin of the "Edit Product" PAGE TRANSITION block. The developer uses the property editor (Figure 5) to specify that the one-tuple relation that flows from the "selectedRow" pin contains only the ITEMNO attribute.

The *Add Product* and *Edit Product* page-designs (Figures 6–7) are conceptually similar to the *Product Catalog* page-design, except that they use UPDATE DB and INSERT DB blocks to represent the appropriate CRUD operation. The "delete product" operation is implemented by directly incorporating a DELETE DB block into the *Product Catalog* page-design. Since no PAGE TRANSITION block is enabled when a product is deleted, the *Product Catalog* page is redisplayed (no page-transition occurs).

4.2 Business Logic: Relational Algebra & Expressions

The CRUD examples of Section 4.1 show that *WebRB* has the right set of database and GUI blocks for web-pages that require no business logic. Now we show how *WebRB* allows developers to incorporate moderate amounts of business logic into their web-pages.

Using a visual programming language to specify business logic in a general-purpose way can be difficult [24]. *WebRB* is able to solve this problem because of the more constrained semantics of its relational application domain. The operators in the relational algebra ([6], Chapter 2) take at least one relation as input and produce a relation as output. Using operators such as JOIN, PROJECT, and UNION, allows *WebRB* business logic to benefit from the close fit between the relational algebra and the relational model used in the *WebRB* database and GUI blocks.

Figure 5. HTML-TABLE Property-Editor

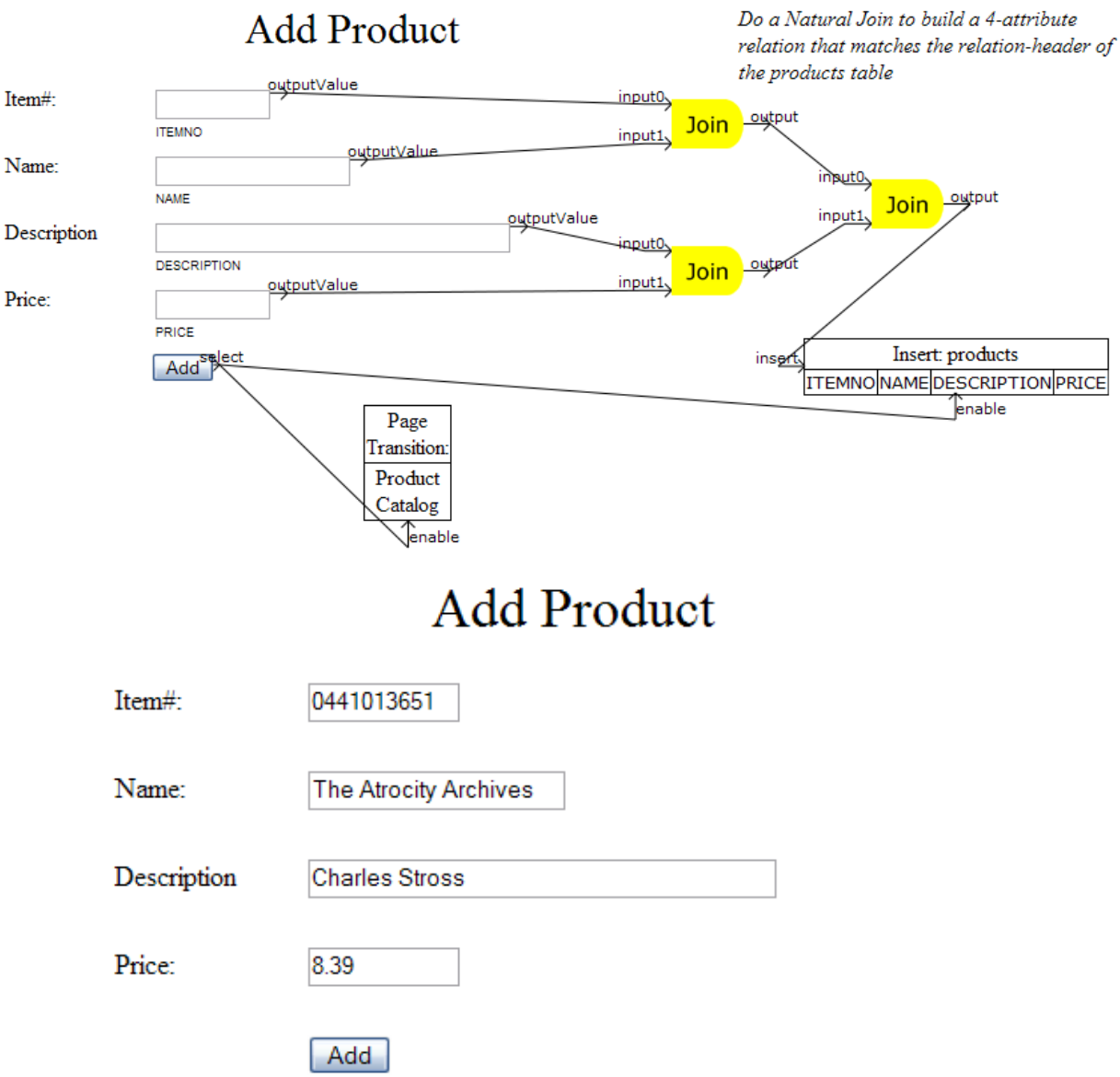


Figure 6. *Add Product* Page-Design and Web-Page

Consider Figure 8, a page-design for a *Search* web-page interface to the PRODUCTS database table. The web-page (Figure 9) allows users to search for a product "by name" and retrieve information about that product. Unlike the *Product Catalog* page, which retrieves the *entire* database table, here the developer wants to retrieve only a specified subset

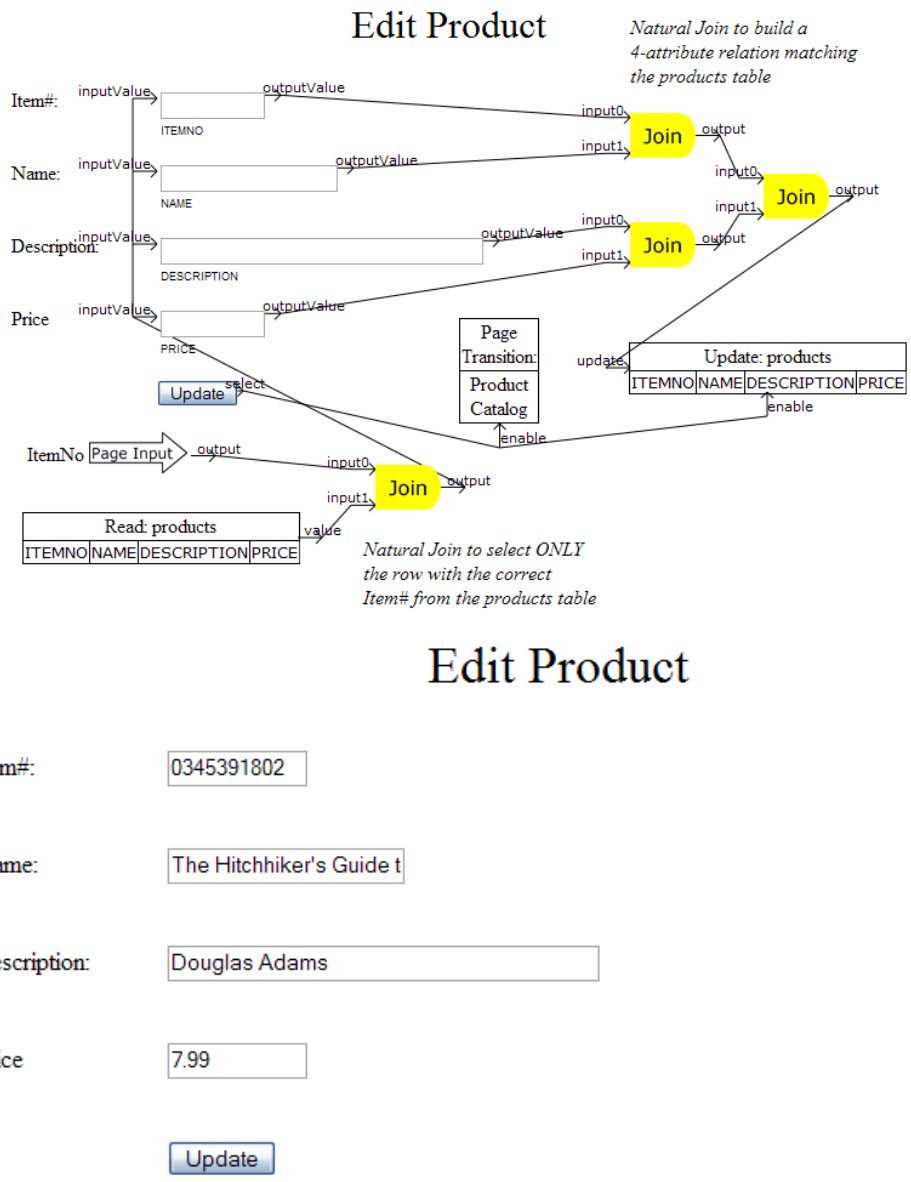


Figure 7. *Edit Product* Page-Design and Web-Page

of rows. There are two inputs to the page: "searchText" (the text to search for), and "msg" (a message to be displayed). When the page is first launched, these two inputs are empty relations.

The developer first uses a JOIN block to compute a Cartesian product between searchText and the PRODUCTS table. Since they have no common attributes, the searchText is essentially appended to each row of PRODUCTS as a new column. Then, the resulting relation is filtered by a WHERE block, so that only tuples where the product NAME matches the searchText are kept. Next, the result is JOINed with a constant string (QTY = "1") to set a default purchase quantity to be displayed. Finally, the results are displayed in an HTML-TABLE.

The WHERE block uses a *tuple expression* to specify that only tuples that "match" the searchText string should be selected from the PRODUCTS table. A tuple expression is an expression whose input is a single tuple and whose result is a single value. Tuple expressions can include constants, but differ from standard expressions in that their variables refer to attributes in the input tuple. Thus, the expression B+3 is interpreted as "the value of the attribute B plus the constant 3". More formally, constants and input attributes are the terminal nodes of a *WebRB* expression tree. The non-terminal nodes in a *WebRB* expression-tree may

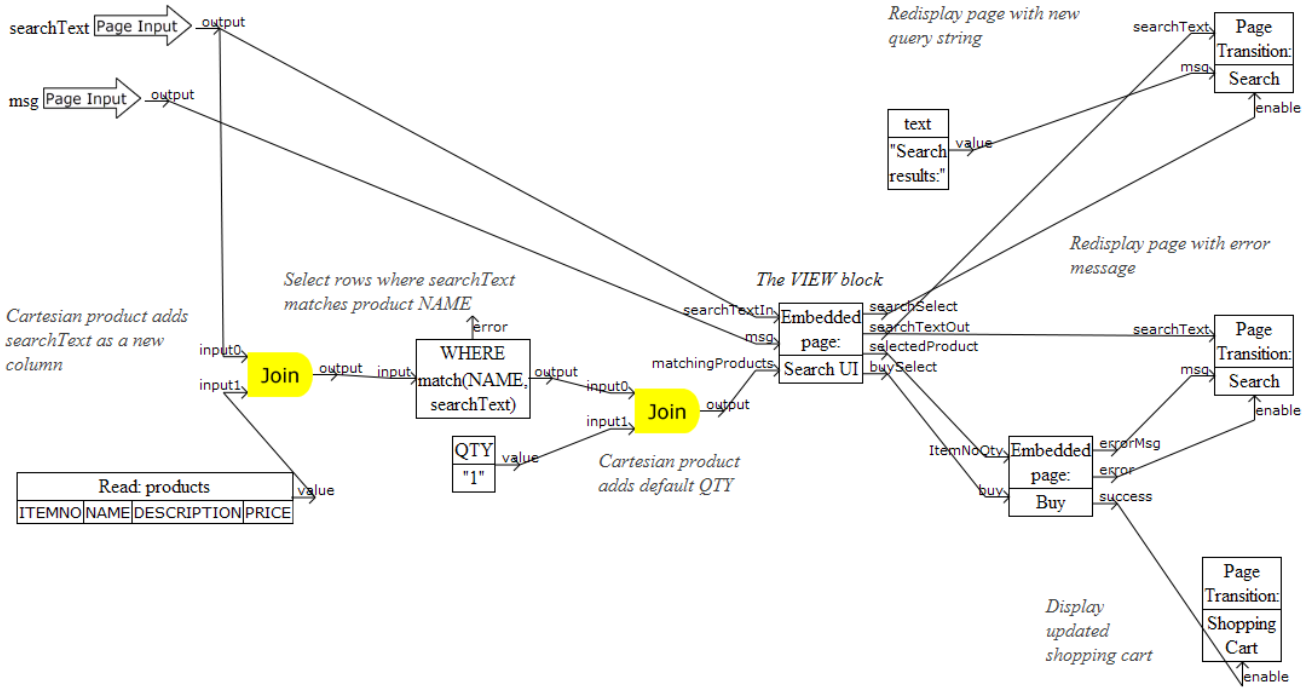


Figure 8. *Search* Page-Design

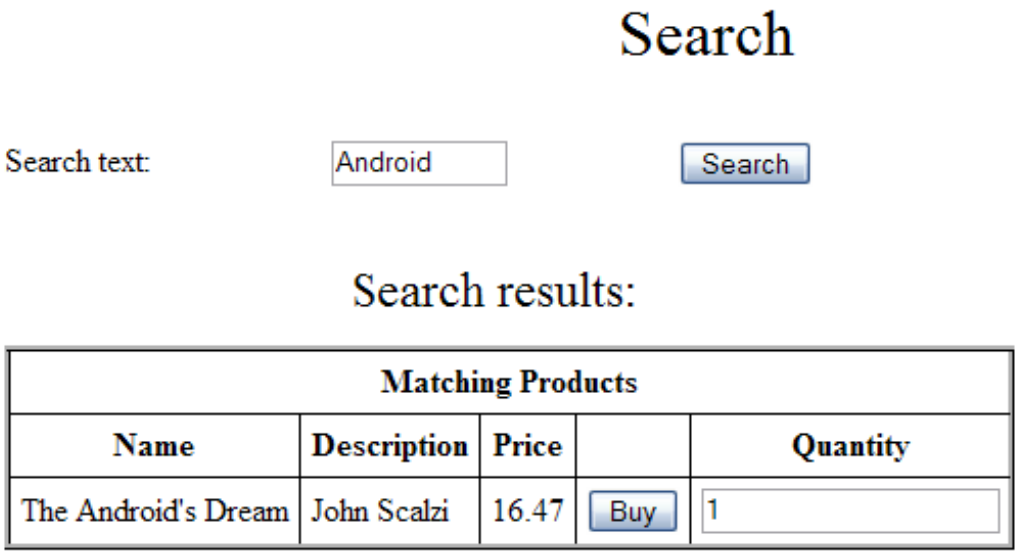


Figure 9. *Search* Web-Page

be unary functions (e.g., unary minus, type conversion) or binary functions (e.g., MATCH(), arithmetic sum, or CONCATENATE()). *WebRB* provides a library of node implementations to support commonly used functions and operators.

Figure 8 shows how *WebRB* uses EMBEDDED PAGE blocks to achieve hierarchical page-design scaling. "Buy product" is a chunk of "controller" function, for example, that was factored out from the original design into a separate "buy" EMBEDDED PAGE block. We discuss this EMBED-

DED PAGE in the context of how *WebRB* does exception handling in Section 4.4. Figure 8 also includes a "search UI" EMBEDDED PAGE block (Figure 10) which encapsulates the web-page's "view". An application's UI can thus be designed by one developer independently of the business logic or database access that is done by other developers. Working independently requires only that the developers agree on the EMBEDDED PAGE API: i.e., the PAGE INPUT and PAGE OUTPUT blocks.

4.3 Business Logic: Natural Join & Functions

The *Shopping Cart* page-design in Figure 11 introduces other constructs to specify web-page business logic. The developer wants to display the contents of a user's shopping cart which is persistently stored in a database table. Following standard normalization techniques, only a product's ITEMNO and the quantity ordered are stored in the CART table. In order to include product name and price information (stored in the PRODUCTS table) in the HTML-TABLE, the developer uses a JOIN block to perform the "natural join" of the two relations. Since the HTML-TABLE also includes a column showing the sub-totals for each line-item in the cart (the "extPrice" attribute), the developer uses a FUNCTION block to calculate this value and add it to the HTML-TABLE's input relation.

FUNCTION blocks are *WebRB*'s solution to the problem of how to implement functions using relational algebra. Consider the operation of converting the string representation of an integer (e.g., "42") to the corresponding integer (42). What is desired is a block with an input attribute of type string ("42"), and an output attribute of type integer (42), representing the converted value. In *WebRB*, this is accomplished by a FUNCTION block.

FUNCTION blocks define their output attributes in terms of tuple-expressions (Section 4.2) applied to their input attributes. In our example, the expression D = TOINT(STR) specifies that "the function's output attribute, D, is the result of applying the function TOINT (convert to integer) to the value of the attribute STR". More generally, a FUNCTION block allows developers to create new attributes, specifying that their value in terms of a tuple-expression that is applied to each tuple in the input relation. A FUNCTION block's output is thus a relation containing only attributes which are derived from the input relation *via* tuple expressions.

Developers can also use FUNCTION blocks to adapt relations to match another block's desired inputs. Examples include: removing unneeded attributes (projection), creating derived attributes, and type-conversion. As shown in Figure 11, input attributes may be propagated directly to the output by specifying the trivial expression NAME = NAME. This is done with the ITEMNO, NAME, and PRICE attributes. The FUNCTION block also converts the quantity to a string for display (QTYSTRING = VARCHAR(QTY)), and calculates the line-item total (EXTPRICE = VARCHAR(QTY * DOUBLE(PRICE))) as a string. (Although not shown here, the expression C = B renames attribute B as C; input attributes are removed from the output relation simply by not explicitly specifying them as output attributes.)

4.4 Business Logic: 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 *WebRB* errors can be detected while designing the application, before running it. *WebRB* uses database meta-data to validate that the names and types of relation attributes associated with persistent model blocks are correct. All block interconnections carry information about their relation headers – the set of attribute names and types which flow on the connection. This enables the *WebRB* 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 DB block does not match the relation header of the block's model, the error is reported to the developer and the page-design will not be executed. Similarly, attempts by a block to access attributes that do not actually appear in the input relation header, can be detected at design time.

Of course, user-provided input cannot be validated before the application actually executes. Some attribute type-mismatches occur only for specific data-values and cannot be detected at design-time. For example, in the example of Section 4.3, D = TOINT(STR), the string "4t2" is not a legal string version of any integer, and therefore cannot be converted to an integer. *WebRB* therefore needs an exception model through which developers can specify how they want the runtime to deal with such errors. Because *WebRB* is a declarative data-flow language, it cannot use the exception model of imperative languages such as C++ and Java. Like other declarative languages, *WebRB* does not have a "flow of control" that can be altered by an exception. Our solution is based on a *Replacement Model* approach [25] in which an expression that is supplied with invalid inputs returns a developer-supplied value as its output. This implies that an expression *always* has an output that it can supply to other blocks. *WebRB* inserts the developer-supplied "error value" into the output attribute whenever an error occurs. As a convenience to the developer, the output relation of a FUNCTION block optionally associates an *error attribute* with each developer-specified output attribute. Error attributes are string-typed, and contain the *WebRB* message generated when the error occurred, e.g. "Invalid integer value". As a further convenience, FUNCTION blocks have a boolean-relation "error" pin whose value is TRUE whenever the output relation contains one or more tuples with an error value.

For example, consider again Figure 9. After a search, the user is given the option to purchase a specified quantity of one of the products in the search results. The business logic for this operation is contained in an EMBEDDED PAGE block

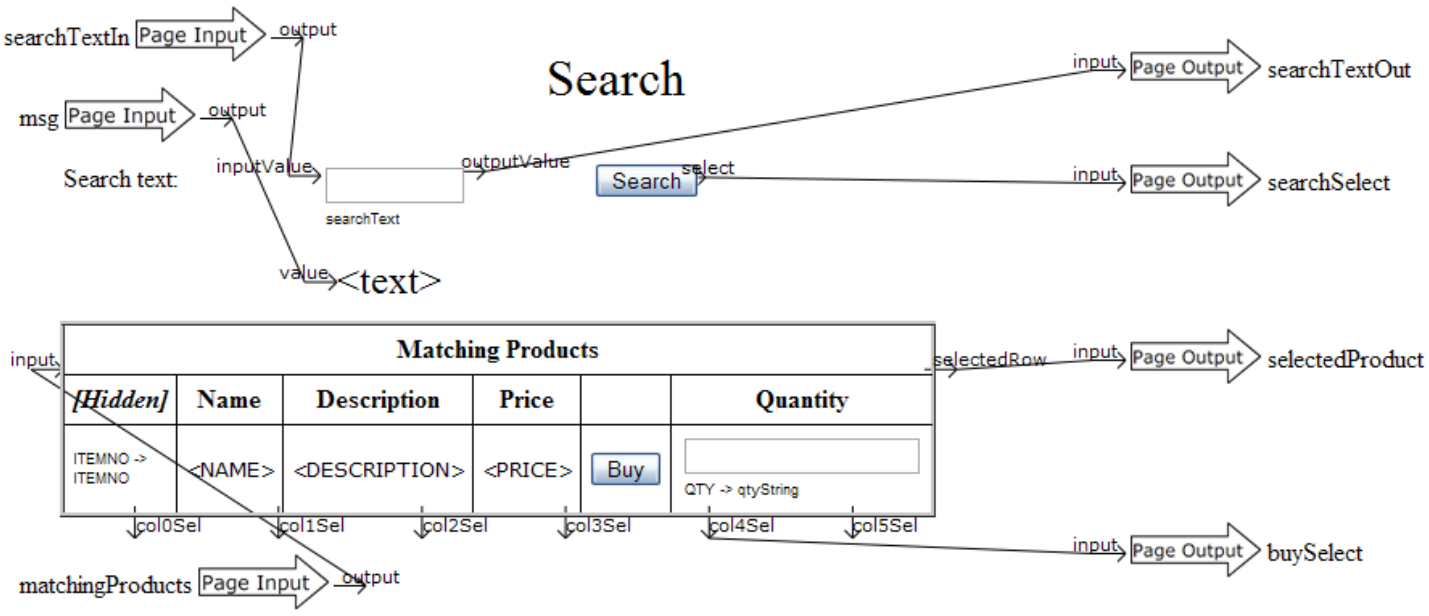


Figure 10. *Search* UI Page-Design

(*Buy*, Figure 12). The Buy block has two inputs: *ItemNoQty* (the item number and quantity from the HTML-TABLE on the *Search* page, and the *buy* enable, which is a boolean relation indicating that the "Buy" button was pressed.

Since the Quantity comes from a text-entry field, the value received by the Buy block is a string. This is converted to an integer by the FUNCTION block on the center left. If the Quantity that the user entered represents a valid integer, the "error" pin from the FUNCTION block will be false. The NOT block in the Buy block design is used to invert the sense of the error pin ("no error"), and fed to a JOIN block along with the "Buy" button signal. A JOIN between two boolean relations is the same as a logical AND of the two values. Thus, the item and quantity are inserted into the CART table if and only if there is no conversion error **and** the "Buy" button was pressed. This value is also fed to the "success" output pin from the Buy block.

If an error occurs during the conversion when the "Buy" button is pressed, a TRUE value is fed to the "error" output pin from the *Buy* page. Another FUNCTION block (top center in Figure 12) builds an error message string, using the function TEXT = CONCAT("INVALID QUANTITY: ", QTYSTRING). The message is fed to the "errorMsg" output from the *Buy* page.

4.5 Session Data

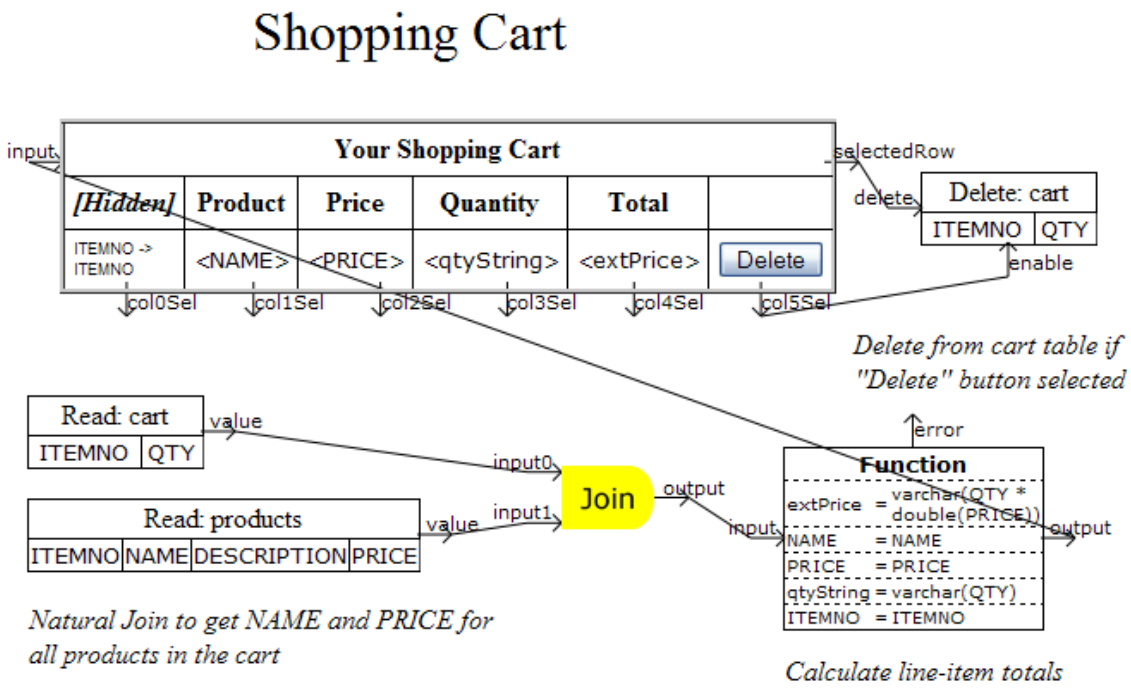
Web applications often maintain state between HTTP requests, usually called *session data*. *WebRB* does not have a built-in mechanism for handling session data. However, session data may be kept in a database table with the rest of the application data. In order to support this approach, *WebRB* will add a mechanism for accessing the session identifier. This could be done by adding a block whose output is the current session identifier.

While session data is not shared by concurrent users, and is therefore not usually considered a concurrency issue, in fact session data may be accessed by concurrent browser requests from the same user. By using database tables to implement session data, concurrent browser requests can be safely supported.

5. Evaluation

In this section we substantiate our claim that *WebRB* can significantly improve developer productivity for the subset of web-applications for which it's designed: multi-page interactive applications that primarily read and update relational databases and include moderate amounts of business logic.

We begin by noting how the *WebRB* data-flow language, and its consistent use of a relational API, allows developers to easily express typical web-application tasks. For example, consider a typical requirement to display the contents of a database table. The imperative-embedding approach (see Figure 2) requires that developers first issue the appropriate query; and then iterate through the result-set, displaying one tuple – and then one attribute – at a time. Figure 1B shows how a *WebRB* developer simply wires the database table (via a READ DB block) to the HTML-TABLE: the relation flows on the wire with the HTML-TABLE extracting the attributes it's interested in "on the fly". The task of acquiring and iterating through the result-set is left to the *WebRB* runtime. As shown by the use of JOIN and WHERE blocks in the lower-left of Figure 9, the simplicity of using relational data flow is maintained even when a developer must transform data between the database source and the HTML-TABLE sink.



Shopping Cart

Your Shopping Cart				
Product	**Price**	**Quantity**	**Total**	
The Android's Dream	16.47	2	32.94	Delete
The Hitchhiker's Guide to the Galaxy	7.99	1	7.99	Delete

Figure 11. *Shopping Cart* Page-Design and Web-Page

In Section 5.1, we compare *WebRB* to well-known alternatives for writing web-applications. We do this using the *Cognitive Dimensions* [26] (CD) approach, following the outline used in [24]. Cognitive Dimensions allows us to quickly evaluate the usability of "information-based artifacts" without doing a detailed analysis. In some cases we have identified deficiencies with *WebRB*. Unless otherwise noted, we plan to address these as time permits. In Section 5.2 we address specific issues of missing *WebRB* function.

5.1 Feature Analysis

For each feature, we contrast the *WebRB* approach with a standard "imperative-embedding" approach, used by PHP, Java, and other languages to generate dynamic web pages. We also mention other systems, where a specific contribution is noteworthy.

Abstraction Gradient: The *WebRB* language uses (Section 2) a relatively small number of abstractions, including *relations, attributes, blocks, pins,* and *wires*. The *WebRB* implementation (Section 3) adds a moderate number of built-in blocks, including *algebra blocks* (JOIN, NOT, FUNCTION, and WHERE); *constant blocks, database blocks* (READ DB,

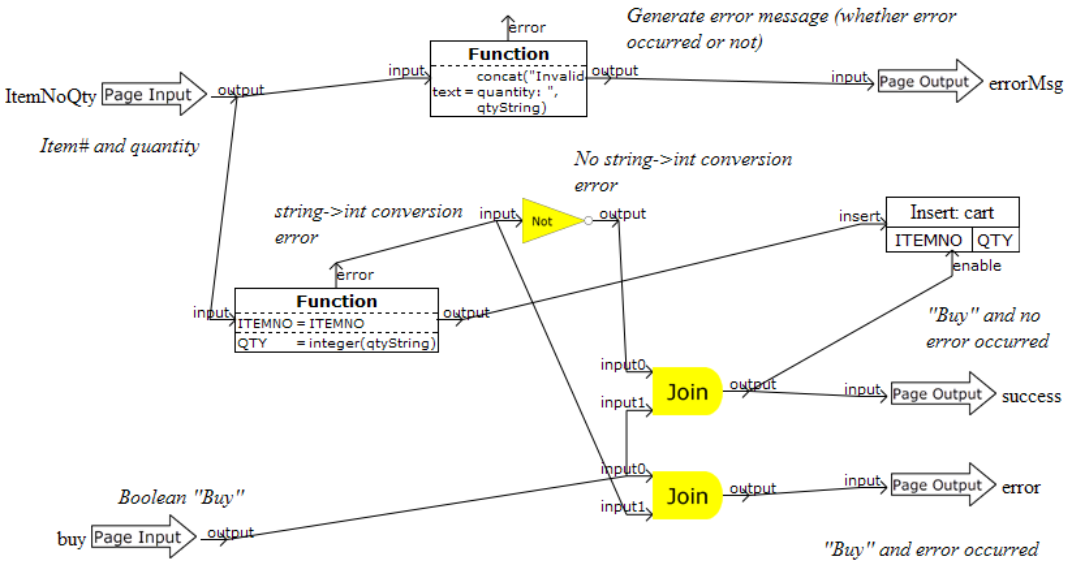


Figure 12. *Buy* Page-Design

INSERT DB, DELETE DB, and UPDATE DB); *widget blocks* (TEXT-ENTRY, TEXT-LABEL, HTML-TABLE, and BUTTON); and PAGE TRANSITION. *WebRB* allows developers to define their own new abstractions (user-defined blocks). CD characterizes such behavior as *abstraction tolerant*.

We note, however, that although many developers have already worked with relational databases, they have probably not worked with formal relational algebra – even though it is the basis for relational database systems. Thus, the use of relational algebra in *WebRB* represents a significant abstraction gradient for many developers.

Imperative embedding approaches include the abstractions of the basic language (e.g., variables, statements, expressions, control structures, arrays, exceptions, classes, methods), as well as abstractions introduced by the libraries used to build the application. For example, each language has a library which is used for database access (e.g., PDO for PHP, JDBC for Java). Other abstractions may be used for HTML page construction, such as Smarty [27] for PHP, and JSP for Java.

Closeness of Mapping: *WebRB* provides excellent closeness-of-mapping for the GUI portion of applications. In the design view, widgets are shown in the same location and with the same appearance they will have in the runtime view. Also, the *WebRB* wires provide a close visual mapping to the dataflow of the system.

The closeness-of-mapping for the database tables and algebra blocks is less clear. Although they provide a close mapping to relational algebra, relational algebra may not map well to the problem domain. For example, a "search" operation requires the use of a JOIN block, and optionally a WHERE block. Although this makes sense from the standpoint of relational algebra, it does not map closely to the desired "search" operation.

For imperative embedded approaches, closeness of mapping for web application GUIs is typically poor. The imperative code is generating HTML, which does not have a close mapping to the GUI that will be seen in the web browser. Visual design tools, such as Dreamweaver, help with this problem, but typically have trouble with generation of dynamic pages, and round-tripping between visual and imperative designs. Developers are often forced to do a "first-cut" using the visual design tool, and then manually modify the result as they add business logic with imperative code.

Imperative access to databases also has poor closeness of mapping. Virtually all database access is initiated via SQL, which does not map closely to the database design. Read operations return a result set, which is accessed a single cell at a time using function/method calls. Tools such as DDS-Pro let the developer work in a more closely mapped domain.

Consistency: *WebRB* attempts to be very consistent. All blocks, pins, and wires use a relation data-type, and any input may be connected to any output. Imperative languages have varying degrees of consistency, but typically a developer cannot infer anything about unknown parts of the language from the known parts.

Diffuseness/Terseness: *WebRB* has many of the diffuseness problems described in [24]. In particular, the visual representation of relational algebra and model components requires more design-screen real estate than comparable text-

based languages (e.g. SQL). However, *WebRB*'s EMBEDDED PAGE blocks mitigate this problem somewhat, since developers can use them to factor complexity of a page-design. Imperative languages such as PHP and Java are generally considered to be less diffuse than visual languages such as *WebRB*.

Error-Proneness: We have identified some common error scenarios with *WebRB* designs. For example, some of the widgets (e.g. TEXT-ENTRY and TEXT-LABEL) use a default attribute name, and developers forget to change it to what they want to use. However, such errors are quickly detected by the *WebRB* validation algorithms.

Imperative languages are well known for providing opportunities for "slips" (where the developer ends up doing something they didn't mean to do). Well-known examples include mistyped variable names and mismatched parentheses or braces [24].

Hard Mental Operations: *WebRB* has a lower demand on cognitive resources than non-visual systems. For example, visible wires are used to route data to and from the widgets. In contrast, many non-visual systems require the developer to remember the association between widgets and (separate) code.

On the other hand, many developers may find the use of relational algebra to be a hard mental operation. The popularity of visual query systems [18] indicates that the visual domain is popular for at least some relational operations. However, the use of relational algebra for **all** "controller" operations may be problematic.

Using imperative code to generate GUIs is certainly a hard mental operation. The developer must continually map between the HTML domain and the visual domain. Visual design tools are a considerable improvement.

Hidden Dependencies: *WebRB* contains hidden dependencies, which arise from the hierarchical composition style of the design environment. At a local level (intra-page), it is easy to see dependencies between blocks, because dependent blocks must be connected by wires. However, between pages, it is more difficult to see dependencies. For example, there is no automated way to find out which blocks reference a particular block. Also, *WebRB* allows multiple instances of database blocks to reference the same database table. Thus, it is not obvious that the input to an UPDATE DB block on Page A affects the next output of a READ DB block on Page B. These limitations can be addressed by enhancements to the development system.

Imperative languages such as PHP and Java certainly suffer from hidden dependencies. In the language itself, there is no way to determine the inbound linkages (e.g., there is no way to determine who calls a function). This is typically addressed by design tools such as Eclipse [28], which can search for dependencies and display them for the developer.

Premature Commitment: Like most visual programming languages, *WebRB* suffers from premature commitment with regard to layout [24]. We address this by making it very easy to move blocks (low viscosity). Premature Commitment also arises at table creation time, because all the attributes and their types must be specified, and cannot be changed without deleting and recreating the table.

Non-visual imperative languages also suffer from premature commitment. Typical workarounds are decoupling (interim versions, or "Plan to throw one away" [29]), and leaving place-holders in the code and going back later to fill them in later [24].

Progressive Evaluation: *WebRB* has excellent support for Progressive Evaluation. The current page displayed in the editor may be validated at any time. Any page without fatal validation errors may be immediately executed at any time.

With imperative languages, progressive evaluation is dependent on the development tools in use. High-end IDEs such as Eclipse support incremental compilation and quick launching of applications.

Secondary notation: *WebRB* provides some support for secondary notation. Developers may place and group the blocks on their designs to make the function more apparent. *WebRB* also supports *annotations*, which consist of text strings that are visible on the design page, but are not visible on the runtime page.

Imperative text languages provide some opportunities for secondary notation. The most obvious example is comments (escape from formalism) and indentation (redundant recoding) [26]. Opportunities for more expressiveness are limited by the one-dimensional nature of text coding.

Viscosity: *WebRB* has low viscosity for many types of changes. For example, blocks may easy by moved by dragging with the mouse, and any attached wires will "rubber band" to follow the block. Additional features would be helpful for reducing viscosity. For example, the ability to select a set of blocks and convert them to an embedded block (refactoring), and the ability to move multiple selected blocks in unison.

For imperative text languages, the "raw" viscosity is fairly high. For example, changing the name of a variable requires finding all the uses of that variable and changing them. Refactoring a piece of code into a separate function/method requires cutting, pasting, and a lot of cleanup work. For GUIs described by text (e.g. HTML), the viscosity is even higher, because editing takes place in the non-visual domain. Design tools (IDEs) are vital for reducing viscosity in textual languages.

Visibility: *WebRB* provides good visibility at the page level, since all the blocks and data flows (wires) are easily seen. Many of the blocks provide a visual indication of their property settings. For example, database blocks display their table name and attribute names, FUNCTION blocks display

a subset of their mapping function(s), and WHERE blocks display their condition expression. However, some blocks potentially have so many properties that displaying all of them would require too much screen real-estate (e.g. there is no limit on how many mapping functions a FUNCTION block may have).

Construction of GUIs with imperative code often leads to visibility problems. For example, GUI event-handling is often implemented by attaching small pieces of code to GUI widgets (e.g. the `onclick` attribute in HTML). Crucial pieces of an application's function are thus diffused, making it hard to modify the application. Visual design tools do not help much with this problem, because they do not have a good approach for managing this complexity.

5.2 Areas For Improvement

WebRB is both an *approach* for building web-applications – using a visual, data-flow, relational DSL – and an *implementation* [22] of this approach. Although our implementation is certainly missing certain features, the examples of Section 4 show that it already can be used to write non-trivial web-applications. We therefore distinguish between missing features that can easily be added to the implementation and features that require redesigning *WebRB* itself.

Modifying the visual presentation: Developers can statically position GUI widgets at the desired "x, y" locations. The property editor in *WebRB*'s visual editor gives developers control of some aspects of a web-page's GUI, such as a page's foreground and background colors. However, a developer cannot specify the color of non-text GUI widgets such as table elements, nor control style (fonts, or location) dynamically. We believe that, in a web-application environment, visual presentation issues should be addressed with HTML/CSS technologies. Thus, by allowing developers to supply their own style-sheets, visual presentation issues that are orthogonal to *WebRB* can be solved using a familiar technology.

More GUI blocks: *WebRB* currently includes HTML-TABLE, BUTTON, TEXT-ENTRY and TEXT-LABEL GUI blocks, but does not include "image" or "link" blocks. We plan to support these blocks as well using the same approach in which we wrap a relational API around a HTML widget's interface.

Integration with AJAX-based frameworks: *WebRB* application deployment is done through a server loading individual page-designs; managing inter-page navigation and data-flow; interrogating a page-design for its HTML; and sending the HTML to be rendered in a standard web-browser (Section 3.2). In this approach, the web-browser does not use "AJAX" techniques [30] such as caching application state or executing business logic. We note that AJAX techniques are primarily concerned with function placement. For example, many techniques move function into the browser to improve response time. *WebRB* is primarily concerned with application function, not placement, and thus is largely orthogonal to AJAX. Thus *WebRB* could incorporate AJAX concepts without requiring fundamental changes. For example, block execution could be done in the browser.

Interfacing with imperative code: A more fundamental weakness of *WebRB* is that it's "self-contained" and has no interface to arbitrary imperative code (e.g., business logic written in PHP and Java). Although *WebRB*'s use of relational algebra and tuple-expressions does support applications with small or moderate amounts of business logic, completely restricting access to other code is problematic. To address this restriction, we first observe that the difficulty does not lie in how the visual *WebRB* environment can interface to non-visual function. *WebRB* blocks such as the persistent database blocks very effectively wrap non-visual function. Instead, the issue is how can a functional or declarative programming language such as *WebRB* interface with an imperative programming language. Our (unimplemented) solution to this problem is to allow imperative code in a *WebRB* page-design so long as it is encapsulated in a functional and relational API. As shown by the persistent database blocks, this can be done – even for operations that change system state – as long as well-defined relational inputs and outputs can be abstracted from the imperative code. Although this approach does not allow arbitrary pieces of imperative code, it significantly relaxes the "all-or-nothing" *WebRB* constraint.

6. Summary

In this paper we identified a ubiquitous set of applications that we termed "relational web-applications". These are dynamic web-applications that:

1. Read relational databases and present the data in a GUI;
2. Update relational databases based on a user's interaction with the GUI;
3. Perform transformations of the relational data which require only simple or moderately complex business logic.

We identified a number of problems with imperative-embedding – the most popular approach for constructing relational web-applications, and introduced *WebRB* as a visual domain-specific language that solves these problems. Because *WebRB* is designed for, and limited to, construction of relational web-applications, it is able to successfully use visual programming techniques to improve productivity without suffering from the scaling issues that are often associated with visual programming languages. *WebRB* blocks have the right level of abstraction for common database read and write operations, and its use of relational algebra provides a good fit between moderately complex business logic and the application's data. Using a detailed set of web-page examples, we substantiated these claims by evaluating *WebRB* across a large number of language dimensions. Also, we showed that *WebRB* provides sufficient functionality to implement any

relational web-application with moderately complex business logic.

We have not yet done any analysis of the runtime performance of *WebRB*. However, we feel that 1) productivity improvements will offset the additional runtime costs; 2) it should be possible for *WebRB* designs to be compiled to efficient runtime constructs; and 3) much of the work can be pushed down to the database. In the current implementation [22], there are no noticeable response-time delays, even though the runtime implementation has not been optimized at all.

References

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

[2] 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.

[3] Ruby on rails. `http://www.rubyonrails.org/`, 2007.

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

[5] Wikipedia. Event loop. `http://en.wikipedia.org/w/index.php?title=Event_loop&oldid=89348024`, 2006.

[6] C.J. Date and H. Darwen. *Databases, Types and the Relational Model (3rd Edition)*. Addison-Wesley, Boston, MA, 2006.

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

[8] Antony J. T. Davie. *Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992.

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

[10] Brian T. Bennett, Bill Hahm, Avraham Leff, Thomas A. Mikalsen, Kevin Rasmus, James T. Rayfield, and Isabelle Rouvellou. A Distributed Object Oriented Framework to Offer Transactional Support for Long Running Business Processes. In *ACM Middleware*, pages 331–348, 2000.

[11] 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.

[12] 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.

[13] M.M. Burnett and A.L. Ambler. A declarative approach to event-handling in visual programming languages. *Proc. IEEE Workshop on Visual Languages*, pages 34–40, 1992.

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

[15] JavaServer Faces Technology. `http://java.sun.com/javaee/javaserverfaces/`, 2007.

[16] MyEclipseIDE. `http://www.myeclipseide.com/`, 2007.

[17] M. M. Zloof. Query-by-example: a data base language. *IBM Systems Journal*, 16(4), 1977.

[18] Tiziana Catarci, Maria F. Costabile, Stefano Levialdi, and Carlo Batini. Visual query systems for databases: A survey. *Journal of Visual Languages & Computing*, 8(2), April 1997.

[19] Database design studio. `http://www.dds-pro.com/products/main.html`, 2006.

[20] Relational Persistence for Java and .NET. `http://www.hibernate.org/`, 2007.

[21] Enterprise Javabeans Technology. `http://java.sun.com/products/ejb/`, 2007.

[22] IBM alphaWorks Services: Web Relational Blocks. `http://services.alphaworks.ibm.com/webrb/`, 2006.

[23] M.M 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.

[24] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7(2):131–174, 1996.

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

[26] Thomas Green and Alan Blackwell. Cognitive dimensions of information artefacts: a tutorial. `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf`, October 1998.

[27] Smarty : Template engine. `http://smarty.php.net/`, 2007.

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

[29] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, 1995.

[30] Justin Gehtland, Dion Almaer, and Ben Galbraith. *Pragmatic Ajax: A Web 2.0 Primer*. Pragmatic Bookshelf, 2006.