Hilda: A High-Level Language for Data-Driven Web Applications

Fan Yang Jayavel Shanmugasundaram Mirek Riedewald Johannes Gehrke
Alan Demers
Cornell University
Ithaca, NY 14850
{yangf,jai,mirek,johannes,ademers}@cs.cornell.edu

Abstract

We propose Hilda, a high-level language for developing data-driven web applications. The primary benefits of Hilda over existing development platforms are: (a) it uses a unified data model for all layers of the application, (b) it is declarative, (c) it models both application queries and updates, (d) it supports structured programming for web sites, and (e) it enables conflict detection for concurrent updates. We also describe the implementation of a simple proof-of-concept Hilda compiler, which translates a Hilda application program into Java Servlet code.

1. Introduction

An important class of applications are *data-driven web applications*, i.e., web applications that run on top of a back-end database system. Examples of such applications include online shopping sites, online auctions, and business-to-business portals. Data-driven web applications can conceptually be divided into four layers: (1) *Database*, which stores the persistent data, (2) *Application logic*, which contains the application logic for performing user actions, (3) *Web site structure*, which groups the application logic operations into web pages and provides the navigational (hyperlinked) structure of web pages, and (4) *Web site appearance*, which controls the appearance of web pages, such as the background color, font size and page layout.

While developing data-driven web applications is a complex and challenging task, the application development interface provided by existing platforms is often too low-level or does not provide a unified model across the different application layers. Specifically, while technologies such as J2EE, Java Servlets/JSPs, ASPs, PHP, WebML and Strudel simplify application development to some extent, they suffer from some of the following shortcomings (Section 5 discusses these approaches in more detail).

Impedance Mismatch: Most existing languages provide a different data model for each application layer (e.g., relational model for databases, Java objects for application logic, hyperlinks for website structure, and form variables for web pages). This "impedance mismatch" makes it hard to develop, maintain, and optimize applications.

Not Declarative: In contrast to declarative high-level database query languages such as SQL, web application development languages such as Java are low-level and procedural. This increases application development time and limits optimization opportunities.

No Unified Handling of Queries and Updates: While some tools such as AutoWeb [18] and Strudel [16] can declaratively specify the structure and content of web sites, they focus mostly on read-only applications. Consequently, they do not provide a uniform framework for handling applications that deal with both queries and updates.

No Structured Programming for Web Sites: Website specification tools such as WebML [7] and Strudel [16] represent a data-driven web site as a graph, where the nodes in the graph are web pages and the edges are links between the pages. Consequently, the "control flow" of the application can jump from one web page to another so long as there is a connecting edge. This is similar to programming with goto statements in the domain of web pages, and has similar disadvantages as compared to structured programming [15].

No Support for Application Conflict Detection: Multi-user, data-driven applications, by their very nature, have a potential for conflicts due to concurrently issued application updates. As we shall illustrate in Section 2.3, such *application-level conflicts* cannot always be handled by database transactions, and in complex applications, such conflicts can be very hard to detect. Existing systems do not provide support for conflict detection.

Mixing of Application Logic and Presentation: While there is broad agreement that application logic should be kept separate from presentation, many existing languages do not enforce this separation; this results in code that is hard to understand, modify, and extend.

To address the above issues, we propose Hilda, a high-level language for developing data-driven web applications. Hilda uses a single data model, is declarative, handles both queries and updates, enforces structured programming, and

supports conflict detection. The expected benefits of Hilda are reduced application development and maintenance cost, and increased optimization opportunities.

The design of Hilda embodies several key concepts. First, Hilda uses a single data model, the relational model [11], to represent all application state. Second, it captures application logic as a sequence of state transitions from one valid application state to another. The application query and update operations are declaratively specified using SQL. Third, Hilda provides an application building block called an *AUnit* (for Application Unit), analogous to a UML class [5]. AUnits support encapsulation like a regular UML class, but the creation and manipulation of AUnits is specified declaratively and provides natural support for conflict detection in the face of concurrent application updates. AUnits are single-entry and single-exit, which facilitates structured programming. Fourth, Hilda separates the application logic, which is represented as AUnits, from the presentation, which is represented as *PUnits* (for Presentation Units) with embedded HTML code. Finally, Hilda has a well-defined formal semantics that provides correctness guarantees for Hilda applications even in the face of application conflicts.

Besides the design of the Hilda language, another challenging aspect is building a Hilda *compiler*, which takes in a Hilda application program and generates executable code. We describe a simple proof-of-concept compiler that translates a Hilda program into Java Servlet code that can be run in a conventional application server. Both the compiler and an application generated by the compiler are available at http://www.cs.cornell.edu/database/hilda. While Hilda allows for many optimization opportunities, these details are beyond the scope of this paper and are part of future work.

The rest of this paper is organized as follows. In Section 2, we describe a course management application that we use as a running example. In Section 3, we describe the Hilda language and in Section 4, we describe our Hilda compiler. In Section 5, we discuss related work and we conclude in Section 6.

2. Case Study: A Course Management System

To illustrate some of the shortcomings of existing application development platforms, we use CMS [6] – a course management system application – as a case study. We developed CMS at Cornell to simplify the management of large courses. Figure 1 depicts the functions supported by CMS. CMS is currently being used by over 2000 students in 40 courses in computer science, physics, economics and engineering. CMS uses a standard three-tier architecture, with a back-end database server, middle-tier application servers and front-tier client browsers. The initial version of CMS was developed using PHP, while the current version was developed using J2EE.

We use four features of CMS to highlight some of the limitations of existing development platforms. Since the issues are similar for both versions of CMS, we focus on the J2EE version. As noted in Sections 1 and 5, other development platforms suffer from similar shortcomings.

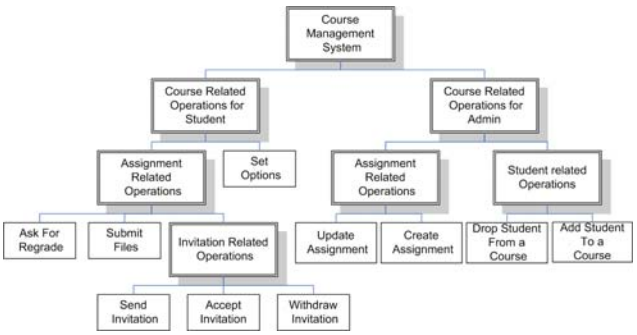

Figure 1. Course Management System

2.1. Assignment Creation

In CMS, the admin of a course can create an assignment for the course by specifying the name of the assignment, the release date, the due date and the set of problems etc.

Impedance mismatch: During assignment creation, the user input is obtained and temporarily stored using HTML forms in the web browser. When the user submits the assignment for creation, this input data is copied into the corresponding assignment Java Bean in the application server. While tools such as Struts [12] simply the mapping between forms and Java Beans to a certain extent, a lot of low-level code is still required to map between the different data models.

Mixing Application Logic and Presentation: When a user is in the process of creating an assignment, CMS performs some application-level sanity checks such as determining whether the due date of an assignment occurs after the release date. Normally, such checks are performed in the web browser (using, say, JavaScript) so that the user obtains an immediate response, without the overhead of contacting the server. However, this causes application logic to be mixed with presentation (in the web browser), which makes applications harder to understand and maintain. Note that design patterns such as Model-View-Controller [2, 12, 19] do not help here because they are server-side solutions.

2.2. Viewing Student Grades

CMS allows students and staff to view relevant grades.

Impedance Mismatch: The student, course, and grade data are stored in database as separate tables and are exposed to application developers as corresponding Java Beans. However, for performance reasons, application developers have to directly work with relational tables to produce a list of students and their grades. Specifically, since each course, student, and grade is represented as a separate Java Bean object, in order to compute the grade for each student in

the course, a join operation has to be performed in Java. It is far more efficient to issue a single SQL query to compute this information because the database can then optimize this query. Consequently, application developers must manually bridge the gap between the J2EE and relational data models and issue SQL queries.

2.3. Student Group Management

CMS allows students to form groups for a given assignment in a course. A student can initiate group creation by extending an invitation to another student. The other student can either accept the invitation (in which case a new group is formed) or decline the invitation. A student can also withdraw an outstanding invitation and groups can be disbanded at any time.

No support for conflict detection: When a student issues a request to accept or decline an invitation, CMS needs to guard against possible conflicting actions such as the inviting student withdrawing the invitation. In addition, there are a variety of other cases unrelated to group management where the action should not be performed, including if the student is dropped from the course (by the course administrator), if the inviting student is dropped from the course, if the assignment has been dropped, if the course itself has been dropped, and so on. Using current programming paradigms and tools, it is very difficult for application developers to correctly identify *all* conditions that need to be checked before performing a specific task such as accepting or declining an invitation. Database constraints cannot be used to solve this problem, because application-level constraints often do not translate directly into database constraints. For instance, dropping an assignment in CMS does not delete the assignment but only sets a "hidden" flag for that assignment (so that it can be resurrected if necessary). Thus, the database will not identify the invitation accepts and declines for a dropped assignment as a violation of database constraints.

Not declarative: Even if the application developer were to correctly identify the correct precondition for performing an action, he or she would have to make an *a priori* decision about how to enforce the condition. For example, the application developer could decide to hold transaction locks for the entire duration of the user input and action, or alternatively, check the precondition just before performing the user action. However, since this precondition cannot be specified declaratively, the system cannot dynamically optimize for the preferred strategy given the current workload, nor can it explore other possibly more efficient strategies such as using triggers to invalidate actions.

2.4. Web Site Structure

No structured programming for websites: CMS supports a rich navigational interface whereby various pages (such as the course overview page) can be reached through multiple paths. While this interface is intuitive for the user, it is very difficult for the application developer to understand the "control flow" between different pieces of application logic spread over interconnected pages. Programming the structure of the web site is reminiscent of programming with goto statements, which make programs difficult to understand and maintain. What is missing is a more "structured programming" paradigm for websites, which nevertheless provides the same rich navigational interface for end-users.

3. The Hilda Language

Hilda is designed to address the above problems. We begin by motivating some of our design decisions.

3.1. Design Decisions

First, Hilda is based on UML [5], a well-accepted modeling framework. Hilda's main construct are **AUnits**, which correspond to UML classes. The local state of an AUnit corresponds to UML class attributes. As classes can have operations, AUnits can have $Activators$. With data and associated operations, the Hilda programming model is state-based in that a Hilda programmer specifies what operations are allowable in a given state of the program. The main difference from the traditional use of UML is that the object creation and operations are specified declaratively[1], which enables the Hilda compiler to automatically perform various optimizations without burdening the user with performance issues.

Second, Hilda uses a single data model - the relational model - to represent the state of all parts of the application, including the database, application logic and the client. This eliminates the impedance mismatch problem and also enables the application logic to be specified declaratively using SQL. The choice of the relational model also allows for a practical and efficient implementation since most existing database systems are relational.

Third, Hilda *logically* separates server and client state to enable highly concurrent execution. The server maintains the current state of the application, and each client sees a (possibly out-of-date) version of it locally. Whenever a client wants to perform an update operation, it checks with the server to see if this operation is still valid in the current system state (to avoid application conflicts). Notice that this separation between client and server state is only *conceptual*. The real separation can be different and should be done by the Hilda compiler or runtime environment based on certain optimization criterion, e.g., sanity checks can be pushed to client side to save bandwidth and round trip time.

Fourth, Hilda models the application logic and associated control flow as a hierarchy. This decision is based on our experience in developing data-driven web applications: since navigation can be very complex, and since the oper-

1 This is also the main reason we use different names for otherwise standard object-oriented concepts, so that declarative and non-declarative constructs are easily distinguished.

ations that a user can perform at any time depend on complex conditions that have to be satisfied by the current state of the user's session, we need a way to cleanly specify these preconditions. Hilda specifies preconditions hierarchically; this helps the programmer to think in high-level abstractions which are then further broken down into smaller steps further down in the hierarchy. Hilda's hierarchical structure also enables encapsulation as the hierarchy naturally limits the scope of the data access of an object. Hilda's control flow goes along the same hierarchy. It is like structured programming, with a tree-like execution structure. It is powerful enough to capture complex graph control flows, but makes the specification of operations more structured and confined to small parts of the code.

Fifth, Hilda uses inheritance to separate application logic from web site structure. Specifically, application developers can derive a web site AUnit by inheriting from the corresponding application logic AUnit. The use of inheritance for this purpose has two advantages: (1) the same structured programming model can be used for both application logic and web site structure, and (2) the same application logic can be reused for multiple web site structures.

Finally, Hilda provides a HTML-based presentation construct called a PUnit (Presentation Unit), which is associated with an AUnit and describes how the content of the AUnit is to be presented. PUnits ensure a clear separation of application logic from presentation because they deal only with presentation issues like page layout, font size and background color, while AUnits deal only with application logic and web site structure.

In the remainder of this section we describe the Hilda language in detail. We start by overviewing Hilda's core construct, the AUnit, in Section 3.2, and we describe AUnits in detail in Section 3.3. We then discuss inheritance in Section 3.4. The semantics of Hilda are described in Section 3.5, and PUnits are described in Section 3.6. We use MiniCMS, a small application inspired by the CMS system discussed above, as our running example.

3.2. AUnits Overview

An AUnit is a single-entry single-exit programming construct that is associated with an (optional) *input schema* and an (optional) *output schema*. The input and output schemas are both relational schemas. Given an AUnit, one or more *instances* of the AUnit can be created. Each instance of an AUnit takes in an input conforming to the input schema of the AUnit and returns an output conforming to its output schema. The act of creating an instance of an AUnit is called *activation*, and the act of destroying an instance of an AUnit is called *deactivation*.

There are three types of AUnits: *Basic AUnits*, *User-Defined AUnits* and *External AUnits*. Basic AUnits are predefined by the system and provide functionality to interact with end users. For example, an instance of the *ShowRow* AUnit shows the attribute values of the input(a single row) to the user and returns no output. Similarly, an instance of the *GetRow* Basic AUnit returns a row of values entered by a user; it takes in no input and returns a single row as an output. Other Basic AUnits for other common interaction tasks are defined similarly.

A User-Defined AUnit corresponds to a functional component in the system. Just as components can have subcomponents, each instance of a User-Defined AUnit also contains zero or more instances of child (User-Defined or Basic) AUnits, which are called *child AUnit instances*. AUnits (like sub-components) can be reused in more than one place. The definition of a User-Defined AUnit contains the application logic of activating and deactivating child AUnit instances, preparing input for child AUnit instances, updating local state and processing output of child AUnit instances and it's own input and output schemas.

External AUnits are used to express small parts of the application logic that do not lend themselves to declarative specification. For example, if an application requires the use of a max-flow min-cut algorithm, it will be awkward to program this using SQL (even though it can theoretically be done with order-based functions and recursion in SQL'99). External AUnits support the same API as other AUnits, but are specified in an imperative language such as Java. Since most data manipulation can be specified declaratively, we expect only a small part of the code to be written using External AUnits; in fact, applications such as CMS do not need External AUnits at all.

One AUnit in the hilda program is designated as the *root AUnit*, which intuitively corresponds to the "main" function in a program. A new instance of the root AUnit is activated each time a new user connects to the Hilda application, and this instance is deactivated when the user disconnects.

Our MiniCMS application consists of the User-Defined AUnits pictorially depicted in Figure 2, which map directly to the functional components of CMS (Figure 1). The root AUnit is the CMSRoot AUnit.

3.3. User-Defined AUnits

Figure 3 shows the BNF grammar for a User-Defined AUnit. As shown, each AUnit has a name (line 2) and a number of other components which we discuss in the next few sections. We will use the code for some of the MiniCMS AUnits (Figures 4, 5, and 9) to illustrate these components.

3.3.1. Schemas A User-Defined AUnit has optional input and output schemas (Figure 3, lines 3-4). Here `Schema` is a non-terminal that describes a relational schema (the production rules for `Schema` are not shown). As a convenient shorthand, a AUnit can have an inout schema (line 5) when the same schema is used for both input and output. We use the notation in.X and out.X to refer to the input and output versions, respectively, of a table X in an inout schema.

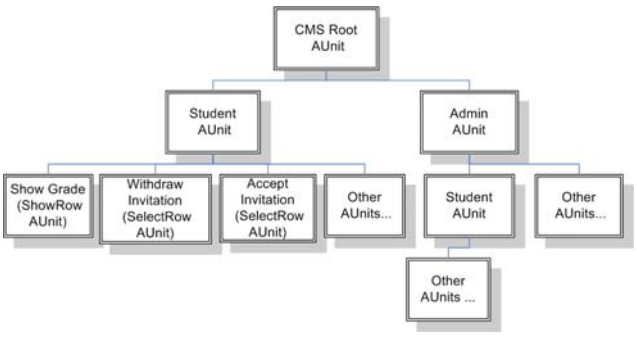

Figure 2. AUnits in CMS

```
01) AUnit ->
02)     AUnitName:STRING '{'
03)     ['input' 'schema' '{' Schema '}']
04)     ['output' 'schema' '{' Schema '}']
05)     ['inout' 'schema' '{' Schema '}']
06)
07)     ['persist' 'schema' '{' Schema '}']
08)     ['persist' 'query' '{' Assignment* '}']
09)
10)     ['local' 'schema' '{' Schema '}']
11)     ['local' 'query' '{' Assignment* '}']
12)
13)     Activator*
14)     '}'
15) Activator ->
16)     'activator' ActName:STRING : AUnitName:STRING '{'
17)     ['activation' 'schema' '{' Schema '}'
18)     'activation' 'query' '{' Query '}']
19)     ['input' 'query' '{' Assignment* '}']
20)     Handler*
21)     '}'
22) Handler ->
23)     [return] 'handler' HandlerName:STRING '{'
24)     ['condition' '{' Query '}' ]
25)     'action' '{' Assignment* '}'
26)     '}'
27) Assignment -> TableName:STRING ':-' Query
```

Figure 3. BNF grammar for a User-Defined AUnit. Constants are denoted as strings in quotes (e.g., `'input'`). Terminals are denoted as name-type pairs, where the name is the symbolic name for the terminal and the type is the type of the terminal (e.g., `ActName:STRING`). Non-terminals are denoted as regular strings (e.g., `AUnit`).

An AUnit can also have a persistent schema (line 7). The data stored in a persistent schema has two important properties: (1) it is *persistent* across AUnit instance activations and deactivations, and (2) it is *shared* between different instances of the same AUnit. The data in the persistent schema is initialized by evaluating the persistent query the very first time the Hilda program is run (line 8). A persistent query is a set of `Assignments`, each of which assigns the result of an SQL query to a table in the persistent schema (line 7). Here `Query` is a non-terminal that describes a SQL query (the production rules for `Query` are not shown). In addition to a persistent schema, an AUnit can have a local schema (line 10). The data stored in the local schema is initialized when a new instance of an AUnit is activated, by evaluating a local query (line 11). The data stored in the local schema is private to a specific instance and is not shared between instances. When an AUnit instance is deactivated, the data in its local schema is destroyed. Another way to view them is that local schema captures session states that are private to each instance in a single user session and persistent schema represents the shared states that can be accessed and updated by multiple instances in difference sessions.

MiniCMS Example: Consider the CMSRoot AUnit in Figure 4. CMSRoot has an input schema (line 2) that specifies the name of a user logged in to the system. CMSRoot does not have an output schema since it is the root AUnit. CMSRoot also has a persistent schema (lines 5-14) that describes the data that the course management system works with – courses, students, assignments, etc. The data stored in the persistent schema is shared among all CMSRoot instances, hence different users can access that data. Since CMSRoot does not have a persistent query, all the tables in the persistent schema are initially empty. CMSRoot does not have a local schema.

As another example, consider the CourseStudent AUnit in Figure 5. CourseStudent captures the application logic for a student in a course, who can view grades and manage his or her groups. CourseStudent takes in the student id, the set of course assignments (lines 2-5), and course group information (lines 6-10) as input, and returns the new course group information (lines 6-10) as output.

3.3.2. Activators: Overview Continuing our discussion of the grammar in Figure 3, AUnits can have zero or more activators (line 13). Activators are used to control (1) how child AUnit instances are activated, (2) how a return of a child AUnit instance is processed, and (3) how child AUnits are reactivated after a child AUnit return has been processed. These three tasks correspond to the *activation phase*, the *return phase*, and the *reactivation phase*, respectively. The activation and reactivation of child AUnit instances is specified declaratively using an "activation query" (to be described soon), which enables Hilda to automatically detect application level conflicts. The return of child AUnit instances are also processed declaratively using SQL-based "handlers". We now describe the three phases.

3.3.3. Activators: Activation Phase As shown in Figure 3, each activator contained in an AUnit has a name, `ActName`, which is unique within the scope of the containing AUnit. Each activator also specifies the name of the child AUnit, `AUnitName`, whose instances it activates (line 16). Each Activator also has an activation schema (line 17) and an activation query (line 18). An activation schema is a relational schema that contains exactly one table (the table can contain any number of columns). The activation

```
1:  AUnit CMSRoot
// Obtain the name of the user as input
02 input schema { user(name:string) }

03// Store information about admins, courses, students, etc.
04// Initially, all tables are empty.
05 persist schema {
06   course(cid:int, cname:string)
07   staff(sid:int, cid:int, sname:string, role:string)
08   student(sid:int, cid:int, sname:string)
09   assign(aid:int, cid:int, name:string, rel:date, due:date)
10   problem(pid:int, aid:int,name:string,weight:float)
11   group(gid:int, aid:int)
12   groupmember(gmid:int, gid:int, sid:int, grade:float)
13   invitation(iid:int, gid:int, invitersid:int, inviteesid:int)
14 }

15 // Activator to activate a student AUnit for each course.
16 // Each student can place, withdraw, accept invitations
17 // from other students to form a group
18 activator ActCourseStudent : CourseStudent {
19   activation schema { acourse(cid:integer) }
20   activation query {
21     SELECT C.cid
22     FROM course C, student S, user U
23     WHERE C.cid = S.cid AND S.sname = U.name
24   }
25   // Prepare the assignments corresponding to the course
26   input query {
27     Student.invitation :-
28     SELECT G.*
29     FROM assign A, group G, invitation I,
30          Student S, user U
31     WHERE A.cid = activationTuple.cid
32       AND A.aid=G.aid
33       AND G.gid=I.gid
34       AND S.sname=U.name
35       AND (I.invitersid=S.sid OR I.inviteesid=S.sid)
36    ...
37   }
38   handler UpdateInv {
39     action{
40       //update assignment
41       invitation :-
42          SELECT *
43          FROM invitation I
44          WHERE I.iid not in
45          (SELECT * FROM Student.in.invitation)
46          UNION
47          SELECT *
48          FROM Student.out.invitation
49     }
50   }
51 }

52 ... (similarly for course staff, system admin, etc.)
```

Figure 4. The CMSRoot AUnit

query produces a set of tuples that conform to the activation schema; the activation query can refer to the tables in the containing AUnit's input schema, local schema and persistent schema. Whenever an instance of an AUnit is activated, each activator contained in the AUnit is processed as follows: for *each* tuple produced by the activation query, a child AUnit instance is activated. This enables an activator to activate multiple child AUnit instances.

Each activator also has an (optional) input query (line 19), which is used to compute the input for each activated

```
1:  AUnit CourseStudent
02 input schema {
03    curstudent(sid:int)
04    assign(aid:int, name:string, release:date, due:date)
05 }

06 inout schema {
07   group(gid:int, aid:int)
08   groupmember(gmid:int,gid:int,sid:int,grade:float)
09   invitation(iid:int,gid:int,invitersid:int,inviteesid:int)
10 }

11 // Show the student's grades for each assignment
12 activator ActShowGrades : ShowRow(string,float) {
13   activation schema {
14     agrade(aid:int,assignname:string,grade:int) }
15   activation query {
16     SELECT A.aid, A.name, GM.grade
17     FROM groupmember GM, student S,
18          assign A LEFT OUTER JOIN
19          Group G ON A.aid = G.aid
20     WHERE G.gid = GM.gid and GM.sid = S.sid
21   }
22   input query{
23     ShowTable.input :-
24       SELECT activationTuple.assignname,
25       activationTuple.grade
26   }
27 }

28 // Withdraw an invitation
29 activator ActWithdrawInv : SelectRow(int,int) {
30   activation schema {
31     aassign(iid:int,inviteesid:int) }
32   activation query {
33          SELECT I.iid, I.inviteesid
34          FROM invitation I, curstudent S
35          WHERE I.invitersid = S.sid
36   }
37   input query {
38     SelectRow.input :-
39       SELECT activationTuple.iid,
40                activationTuple.inviteesid
41   }
42   return handler {
43     //delete the invitation we withdrew
44     invitation :-
45          SELECT *
46          FROM invitation I, SelectRow.output O
47          WHERE I.iid <> O.iid
48   }
49 }

50 // Accept an invitation
51 activator ActAcceptInv : SelectRow(int,int) {
52   activation schema {
53     aassign(iid:int,invitersid:int) }
54   activation query {
55          SELECT I.iid, I.invitersid
56          FROM invitation I, curstudent S
57          WHERE I.inviteesid = S.sid
58   }
59   input query {
60     SelectRow.input :-
61       SELECT activationTuple.iid,
62                activationTuple.invitersid
63   }
64   return handler {
65     //delete the invitation accepted
66     ...
67     //update group, groupmember tables ...
68     ...
69   }
70 }

71 ... (place, decline invitations, etc.)
```

Figure 5. Student AUnit.

child AUnit instance. The input query can refer to the tables in its containing AUnit's input schema, local schema, and persistent schema. In addition, the input query can refer to a special table called `activationTuple`. The activationTuple table has the same schema as the activation schema. Consider a child AUnit instance X that is activated since there exists a tuple x in the activation schema. The activationTuple table for that child AUnit contains exactly x. Thus, the contents of the activationTuple table are different for each child AUnit, so activationTuple can be used to tailor the input for a given child AUnit instance based on its associated tuple in the activation query.

Note that the above process, whereby an AUnit instance recursively activates child AUnit instances, creates a tree of active AUnit instances, with the root of the tree being an instance of the root AUnit. We refer to this tree as an *activation tree* and use the term *parent AUnit instance* to denote the parent of an AUnit instance in the activation tree. We refer to the set of activation trees corresponding to all active root AUnit instances as the *activation forest*.

MiniCMS Example: When a user first connects to MiniCMS, a new user session is created by activating a new instance of CMSRoot, the root AUnit. Figure 6 Session 1, shows the activation tree of a new instance of CMSRoot. When a new instance of CMSRoot is activated, CMSRoot uses its activators – ActCourseStudent (lines 18-51) and other activators (not shown) – to activate child AUnit instances. The ActCourseStudent activator is used to activate instances of the CourseStudent AUnit (line 18) for each course for which the current user is a student. This activation is controlled by the activation schema (line 19), which contains the ids of the relevant courses, and the activation query (lines 20-24), which produces the ids of all courses for which the current user is an administrator. The input query (lines 26-37) produces the input (i.e., information about the student groups) for each activated CourseStudent instance.

Each activated CourseStudent AUnit (Figure 5) instance recursively activates child AUnit instances using its activators: ActShowGrades (lines 12-27), which shows the student grades, ActWithdrawInv (lines 29-49), which allows the student to withdraw outstanding invitations, and ActAcceptInv (lines 51-70), which allows the student to accept invitations. Again, the activation query associated with these handlers declaratively specifies the condition under which the child AUnit instances are to be activated.

Figure 6 shows the activation forest that results when two students connect to CMS in separate sessions. In Session 1, the set of course ids for which the current user is a student is $\{10, 11\}$ (this information is computed from the data in the persistent tables, part of which are shown in the figure). For *each* of these course ids, a new instance of the `CourseStudent` AUnit is activated. Each `CourseStudent` AUnit instance recursively activates child AUnit instances for displaying grades, accepting invitations and withdrawing invitations, as shown in the figure. The activation phase for Session 2 is similar. Note that the different instances of CMSRoot share the same persistent schema (by definition of the scope of persistent schemas).

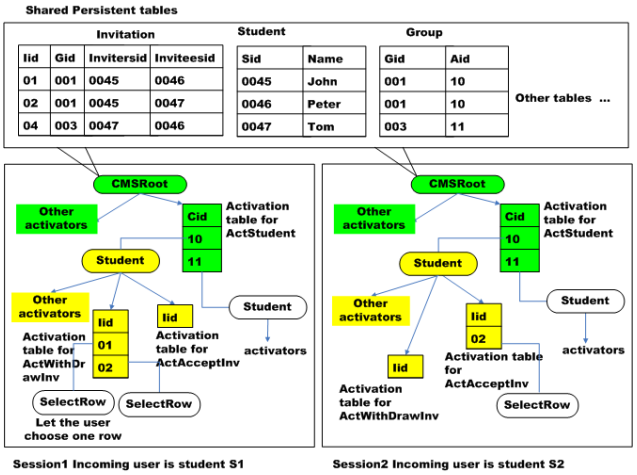

Figure 6. Activation Phase

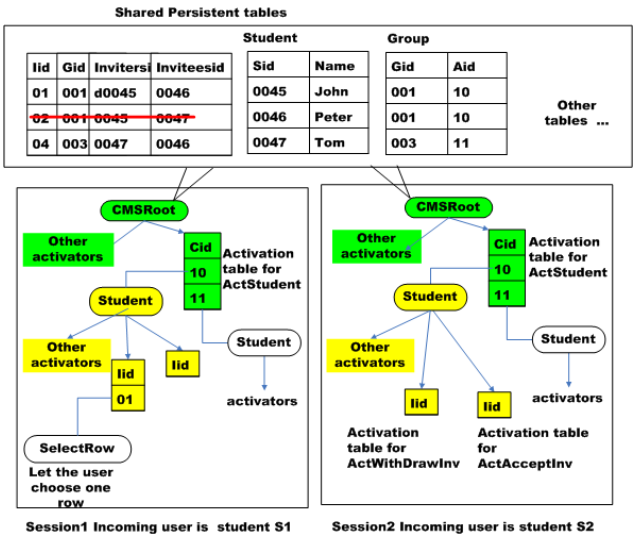

Figure 7. Reactivation Phase

3.3.4. Activators: Return Phase The return phase is initiated when a Basic AUnit instance returns. Since Basic AUnits deal with Input/Output functions, the return phase is typically initiated by a user action such as selecting a row. When a Basic AUnit instance returns, its output is processed by an activator *handler*. The handler can perform certain actions and can (optionally) cause the parent AUnit instance

(of the returning AUnit instance) to return, recursively. After all returns have been processed, the return phase ends and the system transitions to the re-activation phase (discussed in the next section).

Returning to Figure 3, the return of a child AUnit instance is processed by zero or more `Handlers` in the activator that activated the child AUnit instance (Figure 3, line 20). Each handler has a name, `HandlerName` (line 24), which is unique within the scope of the containing activator. Each handler also has an (optional) condition (line 25) and an action (line 26). Whenever a child AUnit instance returns, the conditions of all the handlers contained in the activator are checked. One of the handlers whose condition evaluates to true is non-deterministically chosen and its action is performed. Then, if the handler has the keyword `return` (line 24), the enclosing AUnit also returns and its return is recursively processed. If the handler does not have the keyword `return`, then the system enters the re-activation phase. The action of a handler is specified as an `Assignment`. The queries in the `Assignment` can refer to the same tables as the query in the condition. The action of a return handler can modify only the tables in the persistent schema and output schema of the containing AUnit. The action of a non-return handler can modify only the tables in the local schema and persistent schema of the containing AUnit.

MiniCMS Example: Consider the activation forest in Figure 6 and assume that the user in Session 1 wishes to withdraw an invitation and causes the Basic AUnit instance with ID 20 to return. This return will be processed by the appropriate handlers (Figure 5, lines 42-48, Figure 4, lines 38-50), which will cause the invitation to be removed from the persistent schema.

3.3.5. Activators: Reactivation Phase As described above, during the return phase, the activator handlers along the branch of the activation tree that returns can change the contents of the local and persistent schemas. Consequently, the activation forest has to be "reactivated" so that it is consistent with the new schema contents. The reactivation phase is similar to the activation phase, with special semantics to deal with local schema values and with concurrent user actions. Specifically, suppose AUnit instance a did not return and its activation tuple (in the parent AUnit instance) remains present during reactivation, then instance a is said to be *preserved* across the reactivation and its local schema remains unchanged. The intuitive reason is that an AUnit instance should not lose its temporary state as long as its activation is not affected by the return of a different AUnit instance.

For concurrent user actions, Hilda guarantees a correctness notion analogous to database serializability: the resulting activation forest and user output are *as though* the actions were performed in some serial order. A subtle issue arises here: although two (or more) user actions may be valid in a given activation forest, only one of them may be valid in any serial processing of these actions. For example, consider a student A who has invited a student B to join his group. Two actions are possible in this activation forest: A can withdraw the invitation to B, or B can accept A's invitation. However, if both these actions are submitted concurrently, only one of them can complete successfully (since either of them invalidates the other). A similar situation occurs if A withdraws the invitation to B, but B has still not refreshed her page and tries to accept the invitation.

One of the advantages of Hilda is that it can *automatically* detect such application-level conflicts. The key to detecting such conflicts lies in using the activator conditions of AUnit instances. If an AUnit instance is deactivated (due to an update that causes its activator condition to be false), then pending actions on the AUnit instance cannot be performed since the AUnit is not preserved during reactivation. For instance, after Student 1 withdraws the invitation to Student 2 in Figure 6, the activation forest is updated appropriately (Figure 7) so that Student 2 can no longer accept the invitation from Student 1.

We note that the above semantics of Hilda relaxes the traditional notion of serializability. Two Basic AUnit instance returns (transactions) are said to conflict iff one Basic AUnit instance return violates the activator condition of the other Basic AUnit instance (or any of its ancestors). Hilda's notion of correctness thus specifies conflicts in terms of application-level conditions (which are automatically inferred from activator conditions) and can be viewed as a specific extended transaction model [10, 20, 23, 25] that is tailored to data-driven web applications. Note that the processing of the activation-return-reactivation phases of user actions are still fully serializable since the actions are (logically) performed one after the other; the application-level conditions are used only to check whether a user action is still valid after updates to the activation forest.

3.4. AUnits: Inheritance

Like conventional object-oriented languages, Hilda supports a notion of *inheritance* for extending the functionality of AUnits. Hilda inheritance can be used to add new application logic and also (as we shall see) to specify the structure of an application web site. We use the term *extended AUnit* to refer to an AUnit that uses inheritance, and we use the term *base AUnit* to refer to the AUnit from which an extended AUnit inherits. An extended AUnit inherits all the schemas from its base AUnit. An extended AUnit also inherits all the activators from the base AUnit. In addition, an extended AUnit can add new activators and extend existing activators in the base AUnit. An activator in a base AUnit can be extended in two ways: (1) by adding new handlers, and (2) by filtering the set of activation tuples so that only a subset of the child AUnit instances are activated. The filter-

```
01) ExtendedAUnit ->
02)   'AUnit' AUnitName:STRING
03)      'extends' BaseAUnitName:STRING '{'
04)
05)   ['input' 'schema' '{' Schema '}']
06)   ['output' 'schema' '{' Schema '}']
07)   ['inout' 'schema' '{' Schema '}']
08)
09)   ['persist' 'schema' '{' Schema '}']
10)   ['persist' 'query' '{' Assignment* '}']
11)
12)   ['local' 'schema' '{' Schema '}']
13)   ['local' 'query' '{' Assignment* '}']
14)
15)   (Activator | ExtendedActivator)* '}'
16)
17) ExtendedActivator ->
18)   'extend' 'activator' BaseActName:STRING '{'
19)   ['filter' 'activation' '{' Query '}']
20)   Handler* '}'
```

Figure 8. Grammar for AUnit Inheritance.

ing of the set of activation tuples is specified as a query that returns a non-empty set iff the current activation tuples corresponds to a child AUnit instance that should be activated. Such filtering is usually used to structure the web site by selecting the child AUnit instance that should be presented to a user at a given time.

MiniCMS Example: Consider the NavCMS AUnit in Figure 9. NavCMS inherits from CMSRoot and structures it as a web site that only shows the currently active course selected by the user (recall that CMSRoot activates *all* relevant courses). NavCMS adds this new functionality by defining its own local schema to store information about the currently active course (line 2). It also defines a new activation handler to get user input on the current active course (lines 4-11). In addition, it extends the ActCourseStudent activator (Figure 4, line 18) in CMSRoot so that the CourseStudent child AUnit is only activated for the currently active course; this condition is specified in the activation filter query (Figure 9, lines 13-17), which returns a non-empty result only for the current active course.

3.5. The Hilda Semantics

We have developed a formal semantics [28] for the Hilda language and we present the main aspects here. An important requirement for the semantics is to deal formally with issues of application consistency and concurrency control, thus providing correctness criteria for some of the optimizations discussed below. The semantics is based on execution *histories* [4], where the set of acceptable histories yields a correctness criterion analogous to serializability.

The semantics is based on a formalization of the notions of AUnit instances, activation trees, activation and reactivation. Let a denote an *instance* of some AUnit A, as described informally in Section 3.2. A Hilda application *state* S is just a forest of AUnit instances. We write $a \in S$ when a is an instance contained in S. Recall from the description of reactivation (Section 3.3.5) that AUnit instance a

```
1:  Aunit NavCMS extends CMSRoot

// Keeps track of the currently active course
02 local schema { currcourse(cid:integer) }

03 //Allows user to select from list of courses
04 activator ActSelectCourse : SelectRow(integer,string){
05    input query {
06       SelectRow.input :- SELECT * FROM course
07    }
08    handler {
09       currcourse :- SELECT O.1 FROM SelectRow.output
10    }
11 }

12 activator extending ActCourseStudent {
13    filter activation {
14       SELECT *
15       FROM currcourse CC
16       WHERE activationTuple.cid = CC.cid
17    }
18 }

19 ... (similarly for showing admin courses, etc.)
```

Figure 9. NavCMS inherits from CMS

```
punit ShowNavCMS for NavCMS {
    <body bgcolor="yellow">
        <hr>
        <punit activator=''ActSelectRow''
               name=''ShowSelectRow''>
        <hr>
        <punit activator=''ActCourseAdmin''
               name=''ShowCourseAdmin''>
        <hr>
        ...
    </body>
}
```

Figure 10. PUnit example

may be preserved across multiple reactivation phases. Thus, in a state sequence $S_1, S_2, \ldots, S_n$ we may have $a \in S_i$, $a \in S_{i+1}$, ..., $a \in S_j$.

We define an *operation* to be a pair $op = (a, S)$, where a is an instance of some *basic* AUnit A and $a \in S$. Intuitively, the operation (a, S) represents an attempted return from a in state S. We next define a state transition function $apply : State \times Op \rightarrow State$, representing the effect of a particular operation performed on the application state; $apply$ captures the effect of an instance return on all local, output, input and persistent tables. The relation $allowable : State \times Op$ holds on $(S, (a, S'))$ whenever (a, S') is an operation that could be performed (returned) in state S. Intuitively, $allowable(S, (a, S'))$ reflects a decision to perform a return of AUnit instance a in state S even though the user requesting this operation believed the state to be S'. Clearly, we must require that $a \in S'$ is a basic AUnit instance and that $a \in S$.

With this mechanism, we can define an *execution history*

to be a pair $H = (SE, \preceq)$, where

- $SE = [(S_i, SO_i) \mid 1 \leq i \leq n]$ is a sequence of (state, operation set) pairs.
- $\preceq$ is a partial order on operations consistent with SE, satisfying
$$\forall i, j, op_1, op_2(((i < j) \wedge (op_1 \in S_1) \wedge (op_2 \in S_2)) \Rightarrow \neg(op_2 \preceq op_1))$$

Intuitively, $\preceq$ does not violate the ordering of states.

An execution history is said to be *correct* if there is a sequential ordering of requested operations that is consistent with the history. Formally, there must exist $op_1, \ldots, op_n$ such that

- $\forall j : 1 \leq j \leq n : op_j \in ((\cup_{i=0}^{j-1}(SO_i) - \cup_{i=1}^{j-1}(\{op_i\})) \cap allowable(S_i))$
- $\forall j : 1 \leq j \leq n : \forall op \in ((\cup_{i=0}^{j-1}(SO_i) - \cup_{i=1}^{j-1}(\{op_i\})) \cap allowable(S_i)) \ \neg(op \preceq op_j)$
- $\forall j : 1 \leq j \leq n : S_{j+1} = apply(S_j, op_j)$

The above definition yields a property analogous to serializability for concurrent execution of Hilda programs. This definition will enable us to prove the correctness of the cross-layer caching optimizations discussed in Section 4.

3.6. PUnits

AUnits use a unified model to describe the application logic and the structure of the application website. However, AUnits do not specify presentation details, such as background colors and the page layout in the web browser. In Hilda, such details are specified using PUnits (for presentation units). This enforces the separation of application logic from presentation.

Hilda associates one or more Basic PUnits with each Basic AUnit. Each Basic PUnit describes how a Basic AUnit is to be displayed. For example, the SelectRow Basic AUnit can have one or more Basic PUnits that specify how SelectRow is to be presented to the user (e.g., as form entries or pull-down menus).

Hilda allows users to develop User-Defined PUnits. Each User-Defined PUnit is associated with a User-Defined AUnit and has embedded HTML code that generates part of the HTML page corresponding to that AUnit. Since each web page is composed of a hierarchical tree structure of nested AUnit instances, the User-Defined PUnits associated with User-Defined AUnits can render their part of the page, and recursively invoke the PUnits associated with the child AUnits to build the remaining part of the HTML page. This idea of recursively building up presentation units is similar to the technique proposed in [22].

MiniCMS Example: An example User-Defined PUnit specification for MiniCMS is given in Figure 10. The ShowNavCMS PUnit is associated with the NavCMS AUnit. The PUnit has embedded HTML code to set the page background and draw horizontal lines (<hr>) on the page. In addition, it uses the <punit> tag to invoke other PUnits – ShowSelectRow, ShowCourseStudent – to build up the HTML page. In this example, ShowSelectRow is the PUnit associated with the SelectRow AUnit, which is invoked by the ActSelectRow activator of NavCMS. ShowCourseStudent is similarly associated with the CourseStudent AUnit, which is invoked by the ActCourseStudent activator.

4. Hilda Compiler

We have developed a simple proof-of-concept Hilda compiler (without optimizations), which translates a Hilda program into executable code. The compiler takes in a Hilda program and generates two outputs: the first output is a set of scripts to create tables in a relational database, and the second output is Java Servlet code that can be run in an application server. The generated application runs in a standard three-tier architecture with a client browser, an application server, and a relational database. Using this compiler, we have developed a simple CMS application available at http://www.cs.cornell.edu/database/hilda.

Although our Hilda compiler uses the standard three-tier architecture to implement an application program, the Hilda program itself uses a unified model for all layers of the application. This opens up an opportunity for the compiler to perform *cross layer optimizations*, i.e., automatically choose the layer at which certain pieces of application logic should reside and optimize the interactions between layers. We now list some of the specific cross-layer optimizations we are currently building into our compiler.

Client-Server Code Partitioning. For many applications, it may be more efficient to do certain tasks at the client (browser) instead of at the server. For example, for assignment creation in CMS, a Hilda compiler can decide to cache user input and perform data validation (such as the due date occurring after the release date) at the client, thereby minimizing network traffic. This strategy is possible because the data associated with a newly created assignment does not conflict with any other data. Since the Hilda program is declaratively specified, the Hilda compiler can automatically detect this absence of data conflicts and partition the program accordingly, without burdening the application developer with such details.

Data Caching. There are various opportunities for precomputing data. In CMS, read-mostly data such as student grades can be cached and incrementally maintained at the client side to avoid frequent round-trips and reduce the load on the back-end database server. As another example, entire HTML pages or fragments of pages that contain read-mostly data, such as course overview pages, can be cached to avoid the overhead of building the HTML pages for every access [27]. Since Hilda uses a unified model for all layers of the application, a Hilda compiler can transparently decide to cache data so as to improve performance.

Application Concurrency Control. To avoid inconsistent ap-

plication states, certain conditions need to be checked at the application server before an update operation is performed on the database. For instance, in our CMS group example (Section 2), various conditions such as dropped assignments, withdrawn invitations, etc. need to be checked before a student can accept or decline an invitation. Since such conditions are specified declaratively in Hilda using activation queries, the application server logic can optimize how this condition is to be checked based on query and update workloads, e.g., using locks, optimistic concurrency control, or triggers.

5. Related Approaches

Many tools have been developed to simplify the development of the four conceptual layers (see Section 1) of data-driven web applications.

Commercial tools. Sun's Java 2 Platform, Enterprise Edition (J2EE) with Enterprise JavaBeans (EJBs), JSP and Java Servlets, Microsoft's .NET including ASP.NET, and scripting languages like PHP are representative examples of powerful commercial tools for building Web applications. Other tools for designing web sites and HTML pages are surveyed by Fraternali [17]. Such tools typically use a relational database for the database layer, use objects (such as J2EE) and dispatchers to object methods (such as the Model View Controller [2, 12, 19]) for the application logic layer, use a scripting language (such as JavaScript) and HTML links for specifying the web site structure, and use style sheets such as CSS for specifying web site appearance. The main drawback of these approaches is that they do not provide a unified model for all layers of applications, are not declarative, do not use structured programming for web sites and do not provide systematic methods to deal with application-level conflicts.

Declarative approaches. A variety of research prototype systems has been proposed with the common goal of supporting web application development at a higher level of abstraction. Strudel [16] defines the content of web pages in StruQL, a declarative language which can access and integrate semi-structured data sources and generate web site graphs. However, Strudel only supports read-only operations, but no database updates.

WebML [7, 9] is a powerful and declarative web application development language which shares many common goals with Hilda. WebML provides sophisticated tools for specifying the organization of persistent data, navigational structure, and query/update operations. At its core, WebML extends UML with the concept of "links", which mirror the structure of a web site. The web site is then declaratively specified in this model using a GUI. Hilda differs from WebML in three aspects, which we believe are important especially for large application programs.

First, WebML does not fully separate web site structure from application logic; application logic is embedded as special boxes in the web site graph [7]. Consequently, the control flow of an application is similar to programming with goto statements (links), which makes it difficult to create and maintain large programs. Further, since the application logic is tightly coupled with the web site structure, it is difficult to develop multiple web site structures for the same application logic. In contrast, Hilda supports a more structured programming model, whereby each AUnit instance only communicates with its parent and child AUnit instances. Further, Hilda uses AUnit inheritance to separate application logic from web site structure.

Second, WebML only provides a limited form of code reuse and code abstraction. Specifically, WebML does not provide a declarative way to create complex "functions", which capture complex parts of the application logic and can possibly be reused in multiple places. Instead, all complex application logic is directly embedded in the web site graph as a sequence of simple operation units, and this sequence has to be replicated if it is used in multiple places (unless the sequence is specified non-declaratively as a simple operation unit, in which case the declarative benefits are lost). Hilda, on the other hand, supports encapsulation and code reuse using AUnits. Specifically, each AUnit is declarative, fully encapsulates its functionality, and can be reused in multiple places.

Finally, WebML does not provide support for declaratively specifying and detecting application-level conflicts. In contrast, Hilda uses the activation tree to capture the allowable operations in the current state, and uses this set of allowable operations to detect application-level conflicts.

Abstract State Machines and relational transducers are powerful approaches for describing and validating computing systems [1, 21] and there has been related work on formally specifying workflows and verifying their properties [13, 26]. Recently proposed new standards for describing various aspects related to Semantic Web Services, including a Web Services Modeling Language (WSML) [3], fit into this context as well. This work is related to Hilda in that it models application execution as a sequence of states, and declaratively specifies actions that are possible in each state. Hilda takes this work a step further by providing a complete programming language (Hilda programs are compiled into executable web applications), which is tailored to building data intensive web applications by providing features such as persistence, AUnits with sophisticated support for application conflict detection and PUnits.

Industrial standards. There is growing interest in the industry to separate business and application logic from the underlying platform technology. A major emerging standard is Model Driven Architecture (MDA) [24]. MDA defines different levels of abstraction and well-defined transformations between them. A number of major database vendors like IBM and Oracle support

MDA and data-driven application development ([8, 14], http://www.oracle.com/technology/products/designer).
MDA is a programming methodology rather than an actual programming language. Hilda, on the other hand, is a programming language that can use the MDA programming methodology.

6. Conclusion and Future Work

We have presented Hilda, a high-level declarative language for developing data-driven web applications. Hilda offers many benefits for application developers, including providing a unified model for all layers of the application, providing a structured programming paradigm for developing websites, and providing support for application conflict detection. We have also developed a proof-of-concept Hilda compiler, which can translate Hilda programs into executable code, and have used it to generate code for a simple course management application.

We note that Hilda is not a general-purpose programming language for developing arbitrary applications; rather, it is specifically designed for developing data-driven web applications. Hilda achieves this goal by tightly integrating with the data model and declarative query language of the underlying database system. While we have used the relational model and SQL in this paper, Hilda could be extended to use other data models and associated query languages e.g. XML and XQuery.

As with all new programming languages, we expect that programmers learning to program in Hilda will have a learning curve. However, the fact that Hilda has only a few simple constructs and offers many potential benefits might help ease this transition. We are also working on a GUI development interface based on the Hilda constructs to enable rapid program development as in WebML [7, 9]. It is an open question at this point as to whether the potential benefits of Hilda offset the overhead of switching to a new language.

Our current focus is on developing an *optimizing* Hilda compiler, which can exploit the declarative nature of Hilda language specifications to generate high-performance application code. We are extending the language to provide better ways of specifying work flow and are also using Hilda to develop more applications to illustrate its usability and expressiveness.

References

[1] S. Abiteboul et al. Relational transducers for electronic commerce. In *Proc. PODS*, pages 179–187, 1998.

[2] D Alur et al. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

[3] S. Arroyo et al. *Web Service Modeling Ontology Primer*, June 2005. W3C submission.

[4] P. Bernstein and others. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] G. Booch et al. *The Unified Modeling Language User Guide,The Addison-Wesley Object Technology Series*. Addison Wesley, 1998.

[6] C. Botev et al. Supporting workflow in a course management system. In *Proc. SIGCSE*, 2005.

[7] M. Brambilla and others. Declarative specification of web applications exploiting web services and workflows. In *Proc. SIGMOD*, pages 909–910, 2004.

[8] A. W. Brown. An introduction to model driven architecture. *The Rational Edge*, February 2004. e-zine for the Rational community.

[9] S. Ceri et al. Architectural issues and solutions in the development of data-intensive web applications. In *Proc. CIDR*, 2003.

[10] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of extended transaction models using acta. *ACM TODS*, 19(3):450–491, 1994.

[11] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[12] M. Davis. Struts, an open-source mvc implementation. February 2001, http://www-106.ibm.com/developerworks/library/j-struts/?n-j- 2151.

[13] H. Davulcu et al. Logic based modeling and analysis of workflows. In *Proc. PODS*, pages 25–33, 1998.

[14] P. M. Deshpande et al. Model driven development of content management applications. In *Proc. COMAD*, pages 112–121, 2005.

[15] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *CACM*, 11(3):147–148, 1968.

[16] M. F. Fernandez et al. Declarative specification of web sites with strudel. *The VLDB Journal*, 9(1):38–55, 2000.

[17] P. Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Computing Surveys*, 31(3):227–263, 1999.

[18] P. Fraternali and P. Paolini. Model-driven development of web applications: The AutoWeb system. *ACM TOIS*, 18(4):323–382, 2000.

[19] E. Gamma et al. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[20] H. Garcia-Molina and K. Salem. Sagas. In *Proc. SIGMOD*, 1997.

[21] Y. Gurevich. Abstract state machines: An overview of the project. In *Proc. FoIKS*, pages 6–13, 2004.

[22] D. R. Karger et al. Haystack: A general-purpose information management tool for end users based on semistructured data. In *Proc. CIDR*, pages 13–26, 2005.

[23] J. Moss. Log-based recovery for nested transactions. In *Proc. VLDB*, 1987.

[24] Object Management Group. *MDA Guide Version 1.0.1*, 2003. Available at http://www.omg.org/docs/omg/03-06-01.pdf.

[25] H. Wachter and A. Reuter. The contract model. *Database Transaction Models for Advanced Applications*, 1992.

[26] D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In *Proc. ICDT*, pages 230–246, 1997.

[27] K. Yagoub et al. Caching strategies for data-intensive web sites. In *Proc. VLDB*, pages 188–199, 2000.

[28] F. Yang et al. Hilda: A high-level language for data-driven web applications. Technical report, Cornell University, 2005. http://www.cs.cornell.edu/database/hilda/.