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, (e) it enables conflict detection for concurrent updates, and (f) it separates application logic from presentation. 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 are run on top of a back-end database system. Examples of such applications include online shopping sites, online auctions, and business-to-business portals. 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 for the whole application stack. Specifically, technologies such as J2EE-based application servers, Java Servlets/JSPs, ASPs, PHP, WebML, Strudel, and relational transducers, suffer from some of the following shortcomings (Section 7 discusses these approaches in more detail).

- **Impedance Mismatch:** Data-driven web based applications span three conceptual layers: database, application logic, and presentation. Most existing languages provide a different data model for each layer (e.g., relational model for databases, Java objects for application logic, 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 [21] and Strudel [15] can declaratively specify the structure and content of websites, 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 [14] and Strudel [15] represent a data-driven website 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 [10].
- **No Support for Conflict Detection:** Multi-user, data-driven applications, by their very nature, have a potential for conflicts due to concurrently issued application updates. In a complex application, 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, 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, enforces structured programming, and clearly separates application logic from presentation. The expected benefits of Hilda are reduced application development time, reduced application maintenance cost, and increased optimization opportunities.

The design of Hilda embodies several key concepts. First, Hilda uses a single data model, the relational model [8], 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), which is a single-entry single-exit programming construct. AUnits provide the basic means for structured programming and encapsulation in Hilda, and they are used to specify the structure of the application and its associated web site. AUnits are also structured to aid conflict detection in the face of conflicting application updates. Finally, 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.

Besides the design of the Hilda language, another challenging aspect is building a Hilda *compiler*, which takes in the Hilda program for an application 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 a demo of an example Web site 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 an application, a course management system, that we use as a running example. In Section 3, we describe the Hilda language. We revisit the course management system in Hilda in Section 4. In Section 6, we describe the implementation of the Hilda compiler. In Section 7, we discuss related work, and we conclude in Section 8.

2 Case Study: A Course Management System

To illustrate some of the shortcomings of existing application development platforms, we use CMS [11] – a course management system application – as a case study. We developed CMS at Cornell to simplify the management of large courses. CMS provides the software infrastructure for managing assignments, grading, student groups, etc., and is currently being used by over 1900 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 core 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-based implementation. As noted in Sections 1 and 7, other development platforms also suffer from similar shortcomings.

2.1 Assignment Creation

One of the basic features supported by CMS is the ability to create an assignment for a course. The high-level description of this feature is quite simple: an administrator of a course can specify the name of an assignment, the release date, the due date, the set of problems, and so on. However, the actual code to implement this functionality is surprisingly complex for the following reasons:

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. HTML forms and Java Beans use different data formats, and a good deal of fragile, low-level code is required to map between them.

Mixing Application Logic and Presentation: Before 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. Currently, such checks are performed using JavaScript in the web browser because the user input is obtained and temporarily stored in the web browser (using HTML forms). Such checks are not performed in the application server (where the rest of the application logic resides) because, if the check fails, the temporary state in the web browser (e.g., the user's answers in form fields) would have to be restored so that the user does not have to type in all the information again. Since restoring the state in the web browser requires a lot of low-level code that maps data from the application server data model to the web browser data model, application developers often avoid this mapping by directly performing the check in the web browser. Consequently, application logic is mixed with presentation, making the application harder to understand, modify, and maintain.

2.2 Viewing Student Grades

CMS allows students and staff to view relevant grades.

Impedance Mismatch: The student, course, and grade data is stored in relational tables, while this data is exposed to application developers as Java Beans. However, for performance reasons, application developers have to directly work with the relational tables to produce the list of students and their grades. Specifically, since each course, student, and grade is represented as a separate Java Bean object, it is very inefficient to write nested "for loops" in the application to compute the grade for each assignment for each student enrolled in a course because this translates to performing nested loop joins in the application server. 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 have to 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 development tools, it is practically impossible for application developers to correctly identify *all* the conditions that need to be checked before performing a specific task such as accepting or declining an invitation. Further, these application-level conflicts do not necessarily translate to database conflicts. 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/declines for a dropped assignment as a conflict. Consequently, there is a risk that an action may be performed in an inconsistent application state.

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

Impedance mismatch: The CMS website structure is specified using HTML links with embedded parameters (such as the current course id and so on). Since this data model is completely different from the data models used for the application logic (J2EE and relational databases), it is difficult for the application developer to understand and maintain the interactions between the website and the application logic.

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 website 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

We now introduce the Hilda language. Hilda is based on three key concepts, which help address the shortcomings discussed in the previous sections.

1. *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 design decision has several benefits. First, the use of a single data model **eliminates the impedance mismatch** between the different layers of the application. Second, the use of the relational model enables the application logic to be specified **declaratively** using SQL queries and updates. Finally, the choice of the relational model allows for a practical and efficient implementation since most existing database systems are relational.

2. *Hilda provides a powerful single-entry single-exit programming construct called an AUnit (Application Unit), which models application logic and*

web site navigation as a sequence of state transitions from one valid application state to another. AUnits offer several benefits. First, AUnits handle **both queries and updates in a unified model** since application state transitions are specified like queries using SQL. Second, AUnits enable **structured programming for web sites** since they are single-entry single-exit constructs (like function calls) with a nested tree-like structure, which can nevertheless capture rich graph-like user navigation of a web site. Finally, AUnits can automatically **detect application conflicts** since they explicitly model state transitions and can monitor changes in the state that can invalidate an application action.

3. *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 core Hilda construct – the AUnit – in detail. We also briefly discuss PUnits. We use MiniCMS, a small application inspired by the CMS system discussed above, as our running example. In Section 4, we also use MiniCMS to concretely illustrate how Hilda addresses the limitations described in Section 2.

3.1 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*.

Informally, an AUnit is analogous to a class in conventional programming languages. The input and output schemas of the AUnit correspond to the input and output signatures, respectively, of a method associated with the class. An instance of an AUnit is analogous to an object instance of a class, activation is analogous to object creation, and deactivation is analogous to object deletion.

There are two types of AUnits: *Basic AUnits* and *User-Defined AUnits*. Basic AUnits provide simple Input/Output functionality. For example, an instance of the *ShowRow* AUnit shows a row to a user; it takes in a single row as input (whose attribute values are shown to the user) and returns no output. At a first glance it might seem counterintuitive that ShowRow has input, but no output (it is supposed to *show* something). However, recall that input and output refer to the information flow in the application. ShowRow's data will ultimately be displayed in a browser, but this is part of the presentation, not the application logic. 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 Input/Output tasks are defined similarly.

User-Defined AUnits are used to construct complex AUnits from simpler AUnits. Each instance of a User-Defined AUnit contains zero or more instances of other (User-Defined or Basic) AUnits, which are called *child AUnit instances*. A User-Defined AUnit also contains the application logic to activate and deactivate child AUnit instances, to prepare input for child AUnit instances, and to process output from child AUnit instances. Like all AUnits, a User-Defined AUnit also has its own input and output schemas.

A Hilda program consists of a set of User-Defined AUnits. One of these AUnits is designated as the *root AUnit*, which intuitively corresponds to the "main" function in a program. The root AUnit can have an input schema (to access application environment variables) but cannot have an output schema, i.e., it cannot produce any output. A new instance of the root AUnit (or session) 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 shown in Figures 2, 3, 4, 8, and 13 (as well as some other AUnits that are not shown). For a life demo of the application visit http://www.cs.cornell.edu/database/hilda. The root AUnit is the CMSRoot AUnit shown in Figure 2. Note that the definition of these AUnits contains many details that have not been introduced yet; we describe these next.

3.2 User-Defined AUnits

Figure 1 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.

3.2.1 Schemas

As shown in Figure 1, a User-Defined AUnit has optional input and output schemas (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 short hand, a AUnit can also have an inout schema (line 5) when the same schema

```
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)
16) Activator ->
17)     'activator' ActName:STRING : AUnitName:STRING '{'
18)     ['activation' 'schema' '{' Schema '}'
19)     'activation' 'query' '{' Query '}']
20)     ['input' 'query' '{' Assignment* '}']
21)     Handler*
22)     '}'
23)
24) Handler ->
25)     [return] 'handler' HandlerName:STRING '{'
26)     ['condition' '{' Query '}']
27)     'action' '{' Assignment* '}'
28)     '}'
29)
30) Assignment -> TableName:STRING ':-' Query
```

Figure 1: This figure shows the 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` is a terminal, where `ActName` is the symbolic name and `STRING` is the type). Non-terminals are denoted as regular strings (e.g., `AUnit`). We use the standard notation of `[]` for optional elements and * for zero or more occurrences of an element.

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.

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 `Assignment`s, each of which assigns the result of an SQL query to a table in the persistent schema (line 30). 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.

MiniCMS Example: Consider the CMSRoot AUnit in Figure 2. As mentioned earlier, CMSRoot is the root AUnit. CMSRoot has an input schema (line 2) that specifies the name of a user logged in to the system.[1] 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 CourseAdmin AUnit in Figure 3. CourseAdmin captures the application logic for a course administrator, who can add and drop assignments in a course. CourseAdmin takes in the initial set of assignments and associated problems as input and returns the new set of assignments and problems as output. Since the schema for the input and output are the same, these are represented as an inout schema (lines 4-7). CourseAdmin has no persistent schema or local schema.

As a third example, consider the CreateAssignment AUnit in Figure 4. In MiniCMS, assignments have a name, a release date, and a due date. Each assignment consists of several problems. Each problem has a name and a weight to indicate how much the prob-

[1] In CMS, user login is done using Kerberos authentication (http://web.mit.edu/kerberos/), which is external to the application. Thus, user information is passed in as input to the CMSRoot AUnit.

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

3:  // Store information about admins, courses, students, etc.
4:  // Initially, all tables are empty.
5:  persist schema {
6:    course(cid:int, cname:string)
7:    staff(sid:int, cid:int, sname:string, role:string)
8:    student(sid:int, cid:int, sname:string)
9:    assign(aid:int, cid:int, name:string, release: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 CourseAdmin AUnit
16: // Can add or delete assignments
17: activator ActCourseAdmin : CourseAdmin {
18: // Activate one CourseAdmin for each course
19:   activation schema { acourse(cid:int) }
20:   activation query {
21:     SELECT C.cid
22:     FROM course C, staff S, user U
23:     WHERE C.cid = S.cid and S.sname = U.name
24:           and S.role ="admin"
25:   }
26: // Prepare the assignments corresponding to the course
27:   input query {
28:     //project the assignment on the course
29:     CourseAdmin.assign :-
30:       SELECT A.aid,A.name,A.release,A.due
31:       FROM assign A
32:       WHERE A.cid = activationTuple.cid
33:     //project the problems on assignments in the course
34:     CourseAdmin.problem :-
35:       SELECT *
36:       FROM problem P, CourseAdmin.assign A
37:       WHERE P.aid = A.aid
38:   }
39:   handler UpdateAssignments {
40:     action{
41:       //update assignment
42:       assign :-
43:         SELECT *
44:         FROM assign A
45:         WHERE A.aid not in
46:           (SELECT *
47:           FROM CourseAdmin.in.assign)
48:         UNION
49:         SELECT O.aid, activationTuple.cid,
50:                O.name, O.release, O.due
51:         FROM CourseAdmin.out.assign O

52:       //update problems
53:       problem :-
54:         SELECT *
55:         FROM problem P
56:         WHERE P.pid not in
57:           (SELECT *
58:            FROM CourseAdmin.in.problem)
59:         UNION
60:         SELECT *
61:         FROM CourseAdmin.out.problem
62:     }
63:   }
64: }

65: // Activator to activate a student AUnit for each course.
66: activator ActStudent : Student {
67:    activation schema { acourse(cid:int) }
68:    activation query {
69:      SELECT C.cid
70:      FROM course C, student S, user U
71:      WHERE C.cid = S.cid and S.sname = U.name
72:   }
73:   // Prepare the assignments corresponding to the course
74:   input query {
75:     Student.assign :-
76:       SELECT A.aid,A.name,A.release,A.due
77:       FROM assign A
78:       WHERE A.cid = activationTuple.cid
79:    ...
80: }

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

Figure 2: The CMSRoot AUnit

```
1:  AUnit CourseAdmin
2:  // The input is the current set of assignments for
3:  //the course, and the output is the modified set
4:  inout schema {
5:    assign(aid:int, name:string, release:date, due:date)
6:    problem(pid:int, aid:int, name:string, weight:float)
7:  }

8:  //Activator for creating a new assignment
9:  activator ActCreateAssign : CreateAssignment{
10:   return handler NewAssignment {
11:     action{
12:       assign :-
13:         SELECT * FROM in.assign
14:         UNION
15:         SELECT *
16:         FROM CreateAssignment.newassign
17:       problem :-
18:         SELECT * FROM in.problem
19:         UNION
20:         SELECT *
21:         FROM CreateAssignment.newproblem
22:     }
23:   }
24: }

25: // Show the student's grades for each assignment
26: activator ActShowAssignment : ShowRow(string) {
27:   activation schema {
28:     allassign(id:int,assignname:string) }
29:   activation query {
30:     SELECT aid, name
31:     FROM assign
32:   }
33:   input query{
34:     ShowRow.input :-
35:       SELECT activationTuple.assignname
36:   }
37: }

38: ... (activators for deleting/modifying assignments,
    adding/deleting student)
```

Figure 3: Course Administrator AUnit.

lem contributes to the assignment. CreateAssignment captures the application logic for creating a new assignment. CreateAssignment takes no input (hence it has no input schema) and returns a newly created assignment conforming to the output schema (lines 3-6). CreateAssignment also has a local schema (lines 9-11) that is used to store temporary information about an assignment while it is being created by the user (before being returned as output). The local schema is initialized with a local query (lines 12-14) every time a new instance of an AUnit is activated. The local query initializes the assign table with default values for the name of the assignment (the empty string), the release date (current date) and the due date (current date). Since the local query does not explicitly initialize the problem table in the local schema, the problem table is initialized to be the empty set, i.e., the assignment initially does not have any associated problems. CreateAssignment has no a persistent schema.

3.2.2 Activators: Introduction

Continuing our discussion of the grammar in Figure 1, 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. In the next section we describe the parts of the activator relevant to the activation phase. The other phases are described in later sections.

3.2.3 Activators: Activation Phase

As shown in Figure 1, 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 17). Each Activator also has an activation schema (line 18) and an activation query (line 19). An activation schema is a relational schema that contains exactly one table (the table can contain any number of columns). The activation 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, its local and persistent schemas are initialized as described in the previous section. In addition, 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. If the activation schema and query are absent in the activator specification, a single child AUnit instance is activated

```
1:  AUnit CreateAssignment
2:    //Returns the newly created assignment and problem
3:    output schema {
4:      newassign(aid:int, name:string, release:date, due:date)
5:      newproblem(pid:int, aid:int, name:string, weight:int)
6:    }
7:    //Stores the assignment information so far. Initialized
8:    //with default values for name, release date, due date
9:    local schema {
10:     assign(name:string, release:date, due:date)
11:     problem(pid:int, name:string, weight:int)}
12:   local query {
13:     assign :- SELECT "", curr_date(), curr_date()
14:   }
15:   // Get information about assignment
16:   activator ActAssignInfo : UpdateRow(string,date,date) {
17:     // Show the current assignment properties
18:     input query {
19:       UpdateRow.input :- SELECT * FROM assign
20:   }
21:     // update assignment with children's output tables
22:     handler updateAssign {
23:       assign :-
24:           SELECT * FROM UpdateRow.output
25:       }
26:   }
27:   // Add a problem
28:   activator ActNewProblem : GetRow(string,int){
29:     handler addProblem {
30:       problem :-
31:         SELECT * FROM problem
32:         UNION
33:         SELECT genkey(), O.1, O.2
34:         FROM GetRow.output O
35:     }
36:   }
37:   // Submit assignment creation
38:   activator SubmitAssignment : SubmitBasic{
39:     return handler success {
40:       condition {
41:         SELECT *
42:         FROM assign A
43:         WHERE A.release <= A.due }
44:       action {
45:         newassign :-
46:           SELECT genkey(), A.name, A.release, A.due
47:           FROM assign A
48:         newproblem :-
49:           SELECT P.pid, A.aid, P.name, P.weight
50:           FROM problem P, newassign A
51:       }
52:     }
53:     handler fail {
54:       condition {
55:         SELECT *
56:         FROM assign A
57:         WHERE A.release > A.due
58:       }
59:       action {
60:         assign :-
61:           SELECT "", curr_date(), curr_date()
62:     }
63:   }
```

Figure 4: AUnit for Creating Assignment.

whenever an instance of the containing AUnit is activated.

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*. Note that the final activation forest is independent of the order in which activators are processed within an AUnit and also independent of the order in which AUnits themselves are processed (e.g., depth-first or breadth-first).

Each activator also has an (optional) input query (line 18), which is used to compute the input for each activated 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 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. If the input query is absent, all the tables in the input schema of the child AUnit instance are initialized to be empty.

The result of the activation phase is a forest of recursively activated AUnits.

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 5 Session 1, shows the activation tree of a new instance of CMSRoot. When a new instance of CMSRoot is activated, CMSRoot uses its activators to activate its child AUnit instances. In this example, CMSRoot (Figure 2) contains the ActCourseAdmin (lines 17-64), ActStudent (lines 66-80), and other activators (not shown). These activators are used to activate child AUnit instances recursively. For ease of exposition, we now focus only on the ActCourseAdmin activator; the discussion for the other activators is similar.

The ActCourseAdmin activator is used to activate instances of the CourseAdmin AUnit (line 17) for each course for which the current user is an administrator. This activation is controlled by the activation schema (line 19), which contains the ids of the relevant courses, and the activation query (lines 20-25), which produces the ids of all courses for which the current user is an administrator. Note that the activation query refers

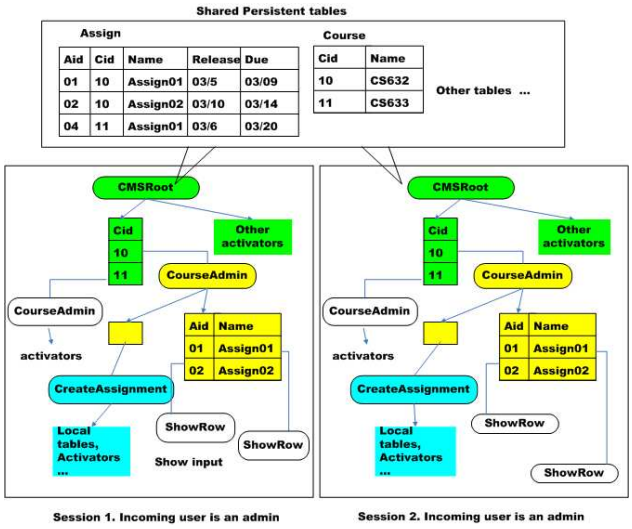

Figure 5: Activation Phase

to the input schema and the persistent schema of CMSRoot. In the example in Figure 5 Session 1, the set of course ids for which the current user is an administrator is $\{10, 11\}$. This information is obtained from a persistent table, which is not shown in Figure 5 for brevity. For *each* of these course ids, a new instance of the `CourseAdmin` AUnit is activated. The input to each of these new instances is computed using the input query (Figure 2, lines 27-38), which produces tables that conform to the input schema of `CourseAdmin` (Figure 3, lines 4-7). For each course id, the input query computes the set of assignments associated with that course id (Figure 2, lines 29-32; recall that activationTuple is bound separately to each tuple – i.e., course id – in the activation schema), and also computes the set of problems associated with the selected assignments (lines 34-37).

When an instance of CourseAdmin is activated, we recursively activate its child AUnit instances. We illustrate the activation of the instance of CourseAdmin corresponding to course id 10 (Figure 5 Session 1). When this instance is activated, CourseAdmin uses its activators to activate its child AUnit instances. In this example, CourseAdmin (Figure 3) contains the ActShowAssignment (lines 24-35), ActCreateAssign (lines 9-22), and other activators (not shown). The ActShowAssign activator shows the list of assignments in the course associated with the instance of CourseAdmin. The list is shown using the basic AUnit, ShowRow, which takes the row to be shown as input and displays it to the user (recall that the notation in.assign is used to refer to the input assign in an inout schema). The ActCreateAssign activator contains no activation query or schema; hence a single instance of the CreateAssignment AUnit is activated.

When a CreateAssignment AUnit instance is activated, it uses its activators ActAssignInfo (Figure 4, lines 16-26), ActNewProblem (lines 28-36), and Sub-

mitAssignment (lines 38-63) to activate its child AUnit instances. The ActAssignInfo and ActNewProblem activators uses the basic AUnits UpdateRow and GetRow, respectively, to allow the user to enter and update the properties of the assignment being created (without actually submitting the assignment for creation). The SubmitAssignment activator uses the basic AUnit, Submit, to allow the user to submit the created assignment.

Whenever a new user connects to MiniCMS, a new session is created by activating a new instance of CMSRoot (e.g., Figure 5 Session 2). The activation phase for the new session is similar to that described above. Note that the different instances of CMSRoot share the same persistent schema (by definition of the scope of persistent schemas).

3.2.4 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 updating an input row or clicking a submit button. 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 reactivation phase (described in the next section).

Returning to Figure 1, the return of a child AUnit instance is processed by zero or more `Handler`s in the activator that activated the child AUnit instance (Figure 1, line 21). Each handler has a name, `HandlerName` (line 25), which is unique within the scope of the containing activator. Each handler also has an (optional) condition (line 26) and an action (line 27). 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 (no action is performed if no handler's condition evaluates to true). Then, if the handler has the keyword `return` (line 25), 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 reactivation phase. If none of the handler conditions evaluate to true, then the system directly enters the reactivation phase.

The condition of a handler is specified as a `Query` (line 26). The query can refer to the input schema, local schema and persistent schema of the containing AUnit, and can also refer to the activationTuple table of the containing activator. The condition is said to evaluate to true iff the query returns at least one tuple. 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 5 and assume that the user in Session 1 submits a new assignment for creation. This causes an instance of the SubmitBasic Basic AUnit (Figure 4, line 38) to return.

The return of the Basic AUnit is processed by the two handlers (lines 39-52 and lines 53-62), which correspond to the successful and unsuccessful creation of the assignment, respectively. The condition of the first handler (lines 40-43) checks to make sure that the release date of the assignment occurs before its due date (the precondition for assignment creation), while the condition of the second handler (lines 54-58) performs the negation of this check (to handle error cases).

When the Basic AUnit returns, the conditions of both the handlers are checked. First, consider the case where the condition of the second handler is satisfied. In this case, the due date does not occur after the release date and the assignment should not be created. Hence, the action of the handler (lines 59-62) simply resets the release date and the due date. Note that the Basic AUnit instance's parent AUnit instance (i.e., the CreateAssignment AUnit instance) *does not* return because the handler is not a return handler. The system thus directly transitions to the reactivation phase.

Now consider the case where the condition of the first (and not the second) handler is satisfied. In this case, the action of the handler (lines 44-51) copies over the details of the assignment from the local schema to the output schema. Since the handler is a return handler, the parent AUnit instance (i.e., the CreateAssignment AUnit instance) returns. The return of the CreateAssignment AUnit instance is processed by the handlers of the appropriate activator in the CourseAdmin AUnit (Figure 3, lines 9-22). Since the activator only has a single handler (lines 10-21) and the handler does not have an associated condition, the action (lines 11-20) is performed unconditionally; the action copies the existing assignments and the newly created assignment to the output schema. Since the handler is a return handler, the CourseAdmin AUnit instance also returns. The return of the CourseAdmin AUnit instance is processed by the handler of the appropriate activator in the CMSRoot AUnit (Figure 2, lines 39-63). The action of the handler (lines 40-62) copies the output of the CourseAdmin AUnit instance to the persistent schema. Since the handler is not a return handler, the return phase ends and the system transitions to the reactivation phase.

Figure 6 shows the activation tree of our running example after the return phase. All the child AUnit

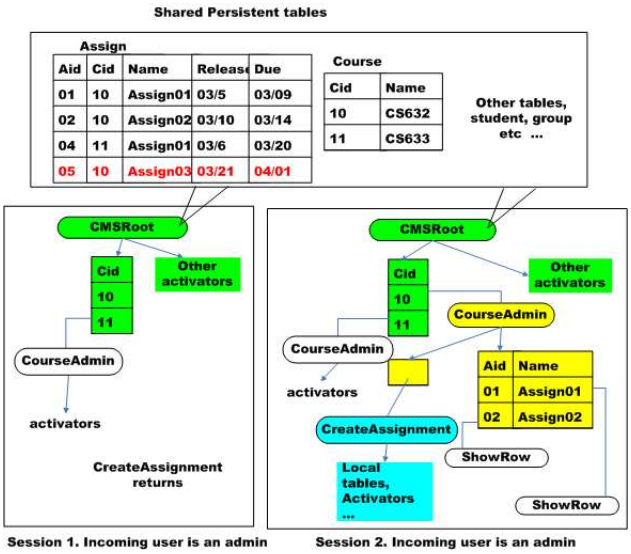

Figure 6: Return Phase

instances of the returned AUnit instances are deactivated, while the AUnit instances along other branches of the activation tree are still activated.

3.2.5 Activators: Reactivation Phase

As described above, during the return phase, the activator handlers can change the contents of the local and persistent schemas of AUnit instances along the branch of the activation tree that returns. Consequently, the activation forest has to be "reactivated" so that they are consistent with the new contents of the local and persistent schemas. The reactivation phase is identical to the activation phase, but with one important difference – some AUnit instances preserve the contents of their local schema. Specifically, all AUnit instances that did not return during the return phase, and which were active both before and after the return phase, preserve the contents of their local schema. The intuitive reason is that AUnits should not lose their temporary state (i.e., they should not lose any temporary work) as long as their activation is not affected by the return phase of another AUnit.

We now define the reactivation phase more precisely. Given an activation forest, each active AUnit instance is assigned a unique identifier called its *label* as follows. Each root AUnit instance is assigned a unique label when it is first created (for example, a unique session identifier). Each non-root AUnit instance is assigned a label that is a triple consisting of (a) the label of its parent AUnit instance, (b) the name of its activator, and (c) the primary key of its activation tuple. We refer to the activation forest just before the initiation of the return phase as $AF_{PreReturn}$.

The goal of the reactivation phase is to construct a new activation forest, which we call $AF_{PostReact}$. Initially, we add to $AF_{PostReact}$ every root AUnit instance from $AF_{PreReturn}$ that did not return during the just-completed return phase. Recursively, when an AUnit instance Y with label y is added to $AF_{PostReact}$, we first check to see if an AUnit X with identical label y existed in $AF_{PreReturn}$ and did not return during the return phase. If these conditions are met, the contents of Y's local schema retains the value of X's local schema. Otherwise, the contents of Y's local schema are initialized using the local query, exactly as in the activation phase. The activation and input queries of AUnit Y are then evaluated, possibly causing child AUnit instances to be added to $AF_{PostReact}$, again exactly as in the activation phase.

Intuitively, if an AUnit instance "survives" the return phase and remains activated during reactivation, it retains the local state that is has accumulated during the computation. An AUnit instance's local state is lost only if the instance is deactivated or returns. As in the activation phase, the final activation forest after the reactivation phase is independent of the order in which activators and AUnit instances are processed.

MiniCMS Example: Returning to our example, Figure 7 shows the result of the reactivation phase after the activation and return phases in Figures 5 and 6 (recall that the return phase added a new assignment for a course). For ease of exposition, we add a unique ID to identify each AUnit instance instead of showing its label (we will also describe other uses for the ID in the next section). In Session 1, the CMSRoot instance (ID 30) and the CourseAdmin instance corresponding to course id 11 (ID 33), and its descendant AUnit instances, retain their local state since they were activated during the activation phase and they did not return. The CourseAdmin instance corresponding to course id 10 (ID 45) and the CreateAssignment instance (ID 50) lose their local state and are reinitialized by their corresponding local queries (Figure 4, lines 12-14) since they returned during the return phase. The ShowRow AUnit instances with IDs 100 and 102 retain their local state since they were activated during the activation phase and they did not return. The ShowRow instance with ID 101 (corresponding to the newly created assignment) has a newly initialized local state since it was newly activated in the reactivation phase.

In Session 2, all the AUnit instances, except the ShowRow AUnit instance corresponding to Aid 5 (ID 71), retain their local state. The ShowRow AUnit instance corresponding to Aid 5 (i.e., the newly created assignment in the persistent schema) has a newly initialized local state since it was newly activated in the reactivation phase. Note how the CourseAdmin AUnit instance corresponding to course id 10 (ID 65) has a new child AUnit instance (ShowRow instance corresponding to Aid 5), even though it retains its local state. This occurs because the input to the CourseAdmin AUnit instance (i.e., the list of current assign-

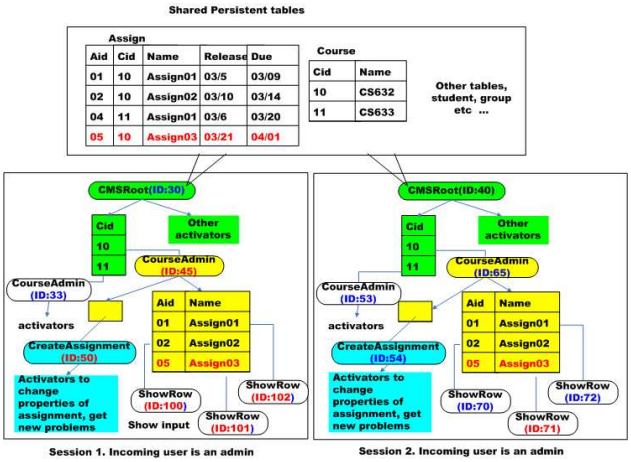

Figure 7: Reactivation Phase

ments for the course) changes due to corresponding changes in the persistent schema, even though the local state of the CourseAdmin AUnit instance is unchanged.

In a practical implementation of Hilda, changes to the activation forest (due to reactivation) do not have to be propagated proactively to all active users of the system. Instead, the changes can be propagated only when the user reloads the page or otherwise interacts with the system.

3.2.6 Activators: Concurrent Actions

So far, we have implicitly assumed that user actions (or AUnit returns) occur serially, i.e., the return phase and the reactivation phase corresponding to each user action is fully completed, before another user action is processed. In reality, of course, user actions can occur concurrently. In such cases, Hilda guarantees a notion of correctness analogous to database serializability: the resulting activation forest and user output are *as though* the user actions were performed in some serial order.

However, there is a subtle issue that arises in the case of data-driven applications: 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 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 occur concurrently, only one of them can complete successfully (since either action invalidates the other). A similar situation occurs when A withdraws the invitation to B, but B has still not refreshed his or 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 conditions of the activators 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 precondition for that operation is no longer true.

More precisely, each AUnit instance in an activation forest is assigned a unique ID every time it is activated. It retains the same ID when it is reactivated. If an AUnit instance is deactivated and is activated again later, it obtains a new ID. Each user action is associated with the ID of a Basic AUnit instance (recall that Basic AUnits are the primitives for Input/Output). Whenever a user action is to be performed on a Basic AUnit with a given ID, a check is first made to verify whether the Basic AUnit with that ID is still activated in the current activation forest. If so, the user action can be performed successfully; else the user action is rejected due to an application-level conflict.

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 [7] 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.

MiniCMS Example: Consider the ActStudent activator in the CMSRoot AUnit (Figure 2, lines 66-80). The ActStudent activator activates an instance of the Student AUnit (Figure 8) for each course for which the current user is a student. The Student AUnit instance for a course takes in the student id, the set of assignments for the course (lines 2-5), and group information for the course (lines 6-10) as input, and returns the new group information for the course (lines 6-10) as output. The Student AUnit has an activator to show the student grades for the course (lines 12-27), withdraw an invitation (lines 29-49), accept an invitation (lines 51-73), and so on.

The activation forest corresponding to two students S1 and S2 who connect to the MiniCMS application is shown in Figure 9. In Session 1, S1 is enrolled in the two courses with course ids 10 and 11. An instance of the Student AUnit is created for each course, and each Student AUnit has activators for withdrawing and accepting outstanding invitations (the part of the activation tree for accepting invitations is not shown for brevity). Session 2 is similar for student S2. Note

```
1:  AUnit Student
2:  input schema {
3:    curstudent(sid:int)
4:    assign(aid:int, name:string, release:date, due:date)
5:  }
6:  inout schema {
7:    group(gid:int, aid:int)
8:    groupmember(gmid:int, gid:int, sid:int, grade:float)
9:    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:     invitation :-
67:         SELECT *
68:         FROM invitation I, SelectRow.output O
69:         WHERE I.iid <> O.iid
70:     //update group, groupmember tables ...
71:     ...
72:   }
73: }
74: ... (place, decline invitations, etc.)
```

Figure 8: Student AUnit.

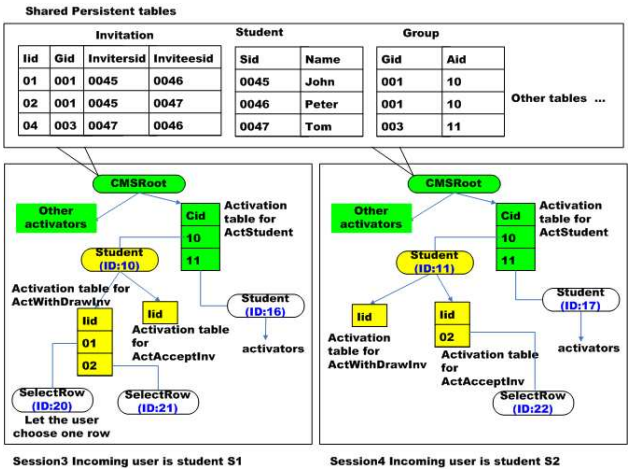

Figure 9: Activation Phase

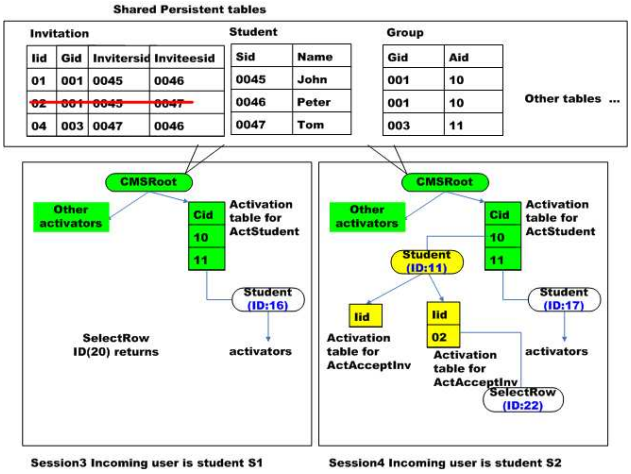

Figure 10: Return Phase

that the persistent schema shows that S1 has previously extended an invitation to S2 to join his or her group.

Now consider the scenario where S1 withdraws the invitation to S2, while S2 concurrently accepts the invitation. This corresponds to operations performed on the Basic AUnits with IDs 20 and 22, respectively. By the semantics of Hilda, the two actions will be performed in some serial order. Let us assume that the invitation withdrawal is performed first. Figure 10 shows the state of the activation forest after the invitation withdrawal is processed. Note that the invitation from S1 to S2 no longer appears in the persistent schema. Figure 11 shows the state of the activation forest on reactivation. Note that the Basic AUnit with ID 22 is no longer activated because the invitation has been withdrawn. Consequently, the second operation of S2 accepting the invitation will be automatically rejected by the system since the AUnit instance with ID 22 no longer exists.

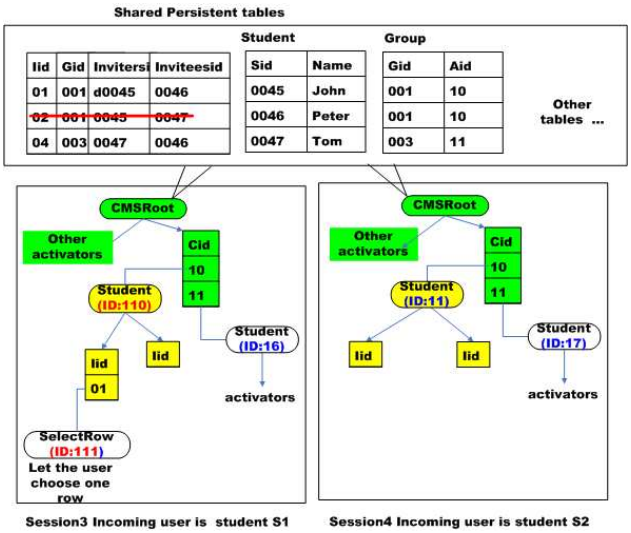

Figure 11: Reactivation Phase

```
01) ExtendedAUnit ->
02)     'aUnit' AUnitName:String 'extends' BaseAUnitName: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 | ExtendedActivator)*
14)
15) ExtendedActivator ->
16)     'extend' 'activator' BaseActName:STRING '{'
17)     ['filter' 'activation' '{' Query '}']
18)     Handler*
19)     '}'
```

Figure 12: Grammar for AUnit Inheritance.

3.3 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 soon 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 has a name and specifies the name of the base AUnit from which it inherits (Figure 12, line 2). An extended AUnit inherits all the input, output, inout, persistent and local schemas from its base AUnit. In addition, an extended AUnit can extend these schemas by specifying additional tables and their initialization queries (lines 3-11).

An extended AUnit also inherits all the activators from the base AUnit. In addition, an extended AUnit can add new activators (line 13, `Activator`) and extend existing activators (line 13, `ExtendedActivator`, line 15) in the base AUnit. An activator in a base AUnit can be extended in two ways[2]: (1) by adding new handlers (line 18), and (2) by filtering the set of activation tuples so that only a subset of the child AUnit instances are activated (line 17). The filtering 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.

```
1:  Aunit NavCMS extends CMSRoot
2:  // Keeps track of the currently active course
    local schema { currcourse(cid:integer) }
3:  //Allows user to select from list of courses
4:  activator ActSelectCourse : SelectRow(integer,string){
5:    input query {
6:      SelectRow.input :- SELECT * FROM course
7:    }
8:    handler {
9:      currcourse :- SELECT O.1 FROM SelectRow.output
10:   }
11: }
12: activator extending ActCourseAdmin {
13:   filter activation {
14:     SELECT *
15:     FROM currcourse CC
16:     WHERE activationTuple.cid = CC.cid
17:   }
18: }
19: ... (similarly for showing student courses, etc.)
```

Figure 13: NavCMS inherits from CMS

MiniCMS Example: Consider the NavCMS AUnit in Figure 13. 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 ActCourseAdmin activator (Figure 2, line 17) in CMSRoot so that the CourseAdmin child AUnit is only activated for the currently active course; this condition is specified in the activation filter query (Figure 13, lines 13-17), which returns a non-empty result only for the current active course.

3.4 PUnits

As described above, AUnits use a unified model to describe the application logic and the structure of the application website. However, AUnits do not specify

[2]Hilda also allows a third form of extension whereby the child AUnit associated with a base AUnit activator can be replaced with a inherited class of the child AUnit. However, we omit the description of this extension since the details are not relevant in the current context.

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 UpdateRow Basic AUnit can have one or more Basic PUnits that specify how UpdateRow is to be presented to the user (e.g., as form entries or pull-down menus).

Hilda allows users to develop User-Defined PUnits corresponding to User-Defined AUnits. Each User-Defined PUnit is associated with a User-Defined AUnit and has embedded HTML code that generates the HTML page corresponding to that AUnit. Since a User-Defined AUnit (say, NavCMS) has nested AUnits (ChooseCourse, DisplayCourse), a User-Defined PUnit can recursively invoke the PUnits associated with the child AUnits to build up the HTML page. This idea of recursively building up presentation units is similar to the technique proposed for Haystack [12].

MiniCMS Example: An example User-Defined PUnit specification for MiniCMS is given below. 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, ShowCourseAdmin – 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. ShowCourseAdmin is similarly associated with the CourseAdmin AUnit, which is invoked by the ActCourseAdmin activator.

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

4 CMS Revisited in Hilda

We now show how the Hilda implementation of MiniCMS addresses the issues discussed in Section 2.

4.1 Assignment Creation

No Impedance Mismatch: Hilda eliminates the impedance mismatch during assignment creation since it uses a single data model – the relational model – to represent all application state. Specifically, in the `CreateAssignment` AUnit shown in Figure 4, the AUnit input and output are represented using the inout schema (lines 3-6) and the temporary state corresponding to the user input is represented using the local schema (lines 9-11); the inout and local schemas are both relational schemas.

No Mixing of Application Logic and Presentation: In Hilda, all application logic is handled in AUnits and all presentation is handled in PUnits. Specifically, in the `CreateAssignment` AUnit shown in Figure 4, all application sanity checks, such as whether the due date of the assignment occurs after the release date, are performed in the conditions of the AUnit activator handlers (lines 40-43, 54-57) instead of being embedded in presentation code. Thus, the PUnit associated with the `CreateAssignment` AUnit can deal solely with presentation issues.

4.2 Viewing Student Grades

No Impedance Mismatch: Since Hilda represents all data as relations, there is no impedance mismatch between accessing persistent and local data - both are declaratively accessed using SQL. For instance, consider the `Student` AUnit in Figure 8, which displays grades using the activator `ActShowGrades` (lines 14-29). The activator `ActShowGrades` uses a SQL activation query (lines 15-21) to specify the list of grades to be shown. The SQL query can be efficiently evaluated by the underlying database system without the application developer having to work with multiple data models.

4.3 Student Group Management

Support for Conflict Detection: As discussed in Section 3.2.6, Hilda handles conflict detection automatically since all the necessary pre-conditions for a user action are encoded in the activation queries. In case a student interacts with a "stale" Basic AUnit and the system has already transitioned to a new state, Hilda can automatically detect such conflicts and avoid performing the invalid actions.

Declarative: In Hilda, all application preconditions are specified declaratively using (SQL) activation queries. This enables the system to optimize execution by choosing a good implementation strategy. In the student group example, whenever a student is logged on, the system can pessimistically lock the student invitations to prevent conflicting concurrent actions by other students. Alternatively, the system can optimistically allow for concurrent actions and handle conflicts during the AUnit return phase. Since the preconditions are specified declaratively, the system can make either choice depending on various factors such as data contention and query workload.

4.4 Web Site Structure

No Impedance Mismatch: As described in Section 3.3, Hilda allows application developers to specify the structure of a web site structure using AUnits, which is the same programming construct used to specify the rest of the application logic. For example, the `NavCMS` AUnit shown in Figure 13 supports navigation between different courses uses the same declarative AUnit specification language as the rest of the application.

Structured Programming for Web Sites: Consider the `NavCMS` AUnit in Figure 13. The `ActSelectCourse` activator (lines 4-11) obtains a user course selection using an instance of the `SelectRow` AUnit, while the `ActCourseAdmin` activator activates the course corresponding to the user selection. From the user's point of view, it is as though he or she clicked on a course and the application "jumped" to the appropriate course web page. However, the application control flow is much more structured: the instance of `SelectRow` that corresponds to the user selection returns to its parent AUnit instance during the return phase, and the activation forest is reactivated during the reactivation phase to produce the new application state. In other words, Hilda uses single-entry single-exit AUnits with well-defined control flow to capture complex web site structure; this makes the web site structure easy to understand and maintain.

5 Hilda Semantics

We now define the formal semantics of a hilda program. Our goal is to give a precise semantics that will help validate our language design, while being simple enough so that it is accessible to application programmers. 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 later. The semantics is based on execution *histories* [16], where the set of acceptable histories yields a correctness criterion analogous to serializability.

We use words relational schema and instance of schema in their traditional meaning. By x.y we mean the field y of tuple x.

Definition 1. s is a relational schema, we define UI(s) as the set of all possible instances(tables) of s, UIS(s) as the set of all possible instance of s with only one row. ■

Please notice that s can contain the definitions for more than one relations.

Definition 2. AUnit is defined as (*name, in, out, local, persist, LocalInitializationFunction, PersistInitialInterpretation, Type, SYN, Activators*),

- *name* is the name of the AUnit which is unique for the whole program.
- *in* is the schema for input relations
- *out* is the schema for output relations
- *local* is the schema for local relations.
- *persist* is the schema for persistent relations.
- *PersistInitialInterpretation* is an interpretation which maps $persist$ to tuples. *PersistInitialInterpretation* ∈UI(persist). It gives the initialization value persistent tables.
- *LocalInitializationFunction*: $UI(in) \times UI(persist) \rightarrow UI(local)$, is a mapping from instances of *in* and $persist$ to instances of *local*. It provides the query to calculate content of local tables.
- $TYPE \in \{BASIC, USER\_DEFINED\}$ shows if the AUnit is basic one or user defined.
- $SYN \in \{true, false\}$ shows if the AUnit is synchronized or asynchronized.
- Activators is a set of *Activator* which is defined below.

Activator is defined as *(name, A', activate, ActivateFunction, InputFunction, OutputHandlers, ReturnOutputHandlers)*.

- A' is the name for an AUnit. It refers to a child AUnit of A.
- *activate* is the schema for *activation table*.
- *ActivateFunction*: $UI(in) \times UI(persist) \times UI(local) \rightarrow UI(activate)$. It gives the query to calculate the content of *activation table*.
- *InputFunction*: $UI(in) \times UI(persist) \times UI(local) \times UIS(activate) \rightarrow UI(A'.in$[3]$)$. It takes in an instance of *in,persist,local* and an instance of *activate* with only one row(one tuple from *activation table*) to an instance of *in* of A'. The functions compute the input relations for A'.
- $OutputHandlers \in 2^{OutputHandler}$. *OutputHandler* is the type for a nonreturn output handler.
- $ReturnOutputHandlers \in 2^{ReturnOutputHandler}$. *ReturnOutputHandler* is the type for a return output handler.

A nonreturn output handler *OutputHandler* is defined as *(conditionFunction, actionFunction)*

- *conditionFunction*: $UI(A'.out) \times UI(in) \times UI(local) \times UI(persist) \times UIS(activate) \rightarrow \{true, false\}$. The function determine if the *actionFunction* should be triggered or not.
- *actionFunction*: $UI(A'.out) \times UI(in) \times UI(local) \times UI(persist) \times UI(activate) \rightarrow UI(local) \times UI(persist)$. It updates the values of instances of *local* and *persist*.

A return output handler *ReturnOutputHandler* is defined as *(conditionFunction, actionFunction)*

- *conditionFunction*: $UI(A'.out) \times UI(in) \times UI(local) \times UI(persist) \times UIS(activate) \rightarrow true, false$. The function determine if the *actionFunction* should be triggered or not.

[3]By A'.x, we mean the field x of the AUnit identified by it's name A'

- *actionFunction*: $UI(A'.out) \times UI(in) \times UI(local) \times UI(persist) \times UI(activate) \rightarrow UI(local) \times UI(persist) \times UI(out)$. Besides instances of *local* and *persist*, it also compute values of instance of *out* and cause the AUnit to return.■

The difference between non return and return output handler is that the former can only update local and persist relations while the latter can also create output relations.

A hilda program is a set of definition of AUnit types and one of them is specified as *root AUnit*.

Definition 3. Similar as relational schema, for an *Activator*, we can define $UI(Activator) = \{(ias, f)|ias \in UI(Activator.activate), f \in UIS(Activator.activate) \rightarrow UI(Activator.A')$ and $dom(f)$[4] $= ias\}$. Then for an AUnit A, we can define $UI(A)=\{(id, iis, ios, ils, ips, acts)|id$ is unique across all AUnit instances, $iis \in UI(A.in), ios \in UI(A.out), ils \in UI(A.local), ips \in UI(A.persist)$, $acts = \{act|act \in UI(Activator), Activator \in A.Activators\}\}$.■

An *Activator* is used by the AUnit to create child AUnit instances, each Activator instance contains an activation table and a function which maps each tuple in the activation table to an instance of *Activator.A'*. Function f is determined by the state of the system at run time, since what the child instances created should be is determined by the result of *Activator.InputFunction*.

We call x an instance of AUnit Type A, if x is in UI(A). For simplicity, by $x.in$ we mean the input relations(instance of $A.in$ of x). It is similar for $out, local, persist$ and $activate$.

When an instance of AUnit x is activated, $x.in$ is provided. Local and activation tables(one for each activator of x) are created with corresponding queries.

Definition 4. Let x denote an *instance* of AUnit type A. A Hilda application *state* S is just a forest of AUnit instances(trees rooted at instances of root AUnit). We write $x \in S$ when x is an instance contained in S.■

Since we encode the application logic as state transitions in hilda program, we need to define the initial state and state transition of a hilda program.

Definition 5. IS is the initial state of the system. 1)For each user session, a root AUnit instance r is activated and $r \in IS$. 2)If AUnit instance $x \in IS$, then for each activator $act \in x.Activators$ and each tuple t in relation $x.act.activate$, we activate an instance x'. $x.act.A'$ specifies the name of AUnit type of x'. We call t the *activationtuple* of x and x is the parent of x', x' is the child of x. We have $x' \in IS$. ■

When the system starts, for each user sessions, an instance of root AUnit is activated. Descendants of root AUnit instances are activated recursively and forms an activation tree for each root. This forest of activation trees are the initial state of the system.

Definition 6. *ID* of an instance x(non *root* instance) is defined as the primary key of x's *activation tuple* plus a's parent's *ID*. For the root instance, *ID* is defined as the empty sequence.■

Definition 7. We define an operation $Op = (a, S_i)$ where a is an AUnit instance and $S_i, a \in S_i$ is a state of the system when the returning of a is attempted.■

Intuitively one operation is defined as the returning(triggered by users) of an instance of *Basic* AUnit at a certain state.

Definition 8. At state S_j, an operation $Op(a, S_i)$, $a \in S_j$ will trigger a state transition $apply : S_j \times Op \rightarrow S_{j+1}$. *apply* is composed of two phases: Return phase and Reactivation phase.

- Return Phase: Let x be the parent of a, and a is activated through activator $act \in x.Activators$. If $handler.conditionFunction$ returns *true*, $handler \in act.ReturnOutputHandlers \cup act.OutputHandlers$, then $handler.actionFunction$ take effects and $x's$ tables will be updated accordingly. If $handler \in act.ReturnOutputHandlers$ then the process will be repeated for x and $x's$ parent. If there are more than one *handler* satisfies the condition, we nondeterministically pick one.

- Reactivation Phase: Recalculate each activation tree and get state S_j, based on the updated information. It's almost the same as how we get activation trees in *initial state*. The difference is how we activate child AUnits. For AUnit instance x, let $activate_j$ be an activation table of x in state S_j and $activate_{j+1}$ be the new activation table calculated based on relations updated after *ReturnPhase*. We compare tuples in $activate_j$ and $activate_{j+1}$ by their primary key.

  - For $t \in activate_{j+1}$ if $\neg\exists t' \in activate_j(t = t')$[5], we activate an instance x' for t and we have $x' \in S_{j+1}$.
  - For $t \in activate_j$ if $\neg\exists t' \in activate_{j+1}(t = t')$, we deactivate the corresponding instance x' for t and we have $x' \notin activate_{j+1}$.
  - For $t \in activate_j t' \in activate_{j+1}(t = t')$, the instance x corresponding to t will be preserved for t' and we have $x \in S_j$. If $x.SYN = true$, $x.local$ is recalculated using $x.LocalInitializationFunction$. Otherwise, $x.local$ keeps unchanged after return phase even if $x.in$ is changed by $x's$ parent. ■

The state transition function representing the effect of a particular operation performed on the application

[4]dom(f) means the domain fo function f

[5]t=t' iff t and t' have the same primary key

state; *apply* captures the effect of an instance return on all local, output, input and persistent tables. There is some flexibility in the definition of the *allowable* relation. Clearly, we must require that $a \in S'$ is a basic AUnit instance and that $a \in S$. However, it can be desirable to make the relation more restrictive. 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'.

Definition 9. We define a relation $allowable : State \times Op$ holds on $(S, (a, S'))$ whenever (a, S') is an operation that could be preformed (a is still active in state S) in state S.■

Definition 10. We 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. ■

Definition 11. 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

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

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

6 Hilda Compiler

We now describe the implementation of a simple (unoptimized) Hilda compiler, which translates a Hilda program into executable code. Using this compiler, we have developed a simple CMS application available at http://www.cs.cornell.edu/database/hilda. As mentioned earlier, the declarative nature of Hilda allows for rich optimization opportunities. While the development of such optimization techniques is beyond the scope of this paper, we highlight some of the specific optimization opportunities made possible by Hilda.

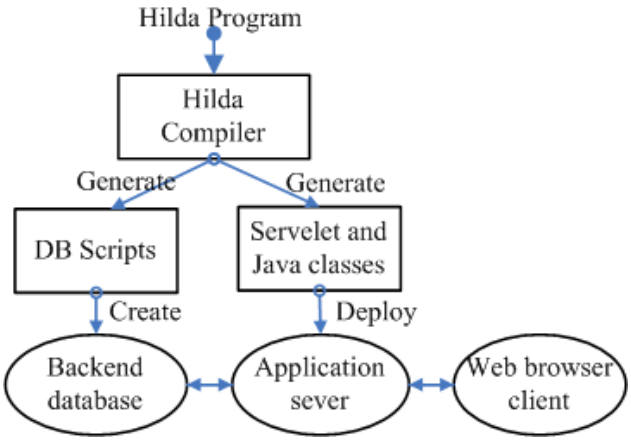

Figure 14: Hilda compiler

6.1 Proof-of-Concept Implementation

The architecture of our Hilda compiler is shown in Figure 14. 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 (which runs the generated Java Servlet code) and a relational database.

The generated Java Servlet code is structured as follows. Each AUnit in the Hilda program is translated into a Java class that contains methods to compute the contents of the local, persistent, input, output and activation schemas. The contents of the local and persistent schemas are stored in the database, while the contents of the input and output schemas are passed around as query (view) definitions so that they are only materialized on demand. The contents of the activation schemas alone are stored in main memory to enable the fast creation of the activation forest. The generated Java Servlet also deals with all incoming http requests from the client.

The Java class corresponding to each AUnit has a `toHMTL` method to present the AUnit content in HTML format. The `toHTML` method is generated based on a default PUnit specification. The `toHTML` method of a parent AUnit instance recursively calls the `toHTML` methods of its child AUnit instances. A client page is constructed by invoking the `toHTML` method of the root AUnit instance.

6.2 Optimization Opportunities

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 building into our compiler.

Client-Server Code Partitioning. For a given application program, 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 side before shipping the data to the server, 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. Note that unlike many current systems, the Hilda application developer *does not* have to worry about such partitioning – it can be automatically done by the Hilda compiler.

Data Caching. There are various opportunities for precomputing data to improve the performance of an application. In CMS, read-mostly data such as student grades can be cached and incrementally maintained at the application server 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 [13]. 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 without any manual intervention by the application programmer.

Application Concurrency Control. To avoid inconsistent application 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. For instance, the compiler can choose to pessimistically enforce the condition by holding locks, or can improve performance in low contention environments by optimistically checking for the condition just before the action is performed, or can perform other optimizations by placing triggers on relevant data items that could potentially violate the condition. Again, the Hilda programmer need not be aware of these alternatives, which can be automatically chosen by the Hilda compiler either statically or during run-time.

7 Related Work

Commercial tools. There are powerful commercial tools for building Web applications, e.g., 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. Other widely used tools for designing Web sites and html pages are surveyed by Fraternali [20]. However, as discussed in Sections 1 and 2, current tools suffer from problems like impedance mismatch, are not declarative, and/or do not enforce structured programming for Web sites and separation of application logic and presentation.

Research prototypes with 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. The Active View system enables declarative specification of E-commerce applications by using views over XML repositories [1]. However, some core parts of the application logic that manipulate the database state (called repository methods in [1]) are specified using imperative programming languages such as Java or C++. Other recent proposals cleanly separate tasks like data management (database schema), navigation (page content and link structure), and presentation (actual displaying of information). Strudel [15] defines the content of Web pages in StruQL, a declarative language for accessing and integrating semistructured data sources. The WebOQL language [3] supports querying of existing Web content to produce views of Web sites. AutoWeb [21], WebML [19, 14], and Araneus [24] include advanced tools for specifying structure of data, navigation, and presentation. These approaches focus mostly on presentation (querying data); only WebML's model explicitly includes data entry [4]. However, none of these systems is designed for applications with concurrent updates to the degree that Hilda is. Hilda's unified handling of queries and updates and its built-in support for conflict detection at application level go well beyond previous work in the area.

Deductive databases, formal specification. Deductive database systems like LDL++ [2] provide powerful models for declaratively specifying knowledge-intense applications. However, in contrast to Hilda they do not support end-to-end design of Web applications. Abiteboul et al. [18] specify business processes as relational transducers that map sequences of input relations into sequences of output relations. Their goal, like in [9], is to make application verification efficient. This work is complementary to Hilda.

Object-oriented database systems. One of the key arguments for introducing object-oriented database systems (OODBMS) [23] was to avoid the impedance mismatch caused by mapping application ob-

jects into relational tables. However, most existing database systems today are relational and therefore the impedance mismatch problem continues to exist when interfacing with object-oriented technology. Further, an OODBMS is focused mostly on the layers below the application layer. In contrast, Hilda is based uniformly on the relational model and supports the entire application stack.

Specialized solutions. Workflow management systems (WFMS) [25] focus on certain parts of application logic dealing with orchestration and failures (see also [14]). The ZOO system [22] extends workflows to support the life cycle of scientific experiments based on an object-oriented design. Visual query systems for databases [6] concentrate on presentation issues. In contrast, Hilda is a general-purpose tool for building complete data-driven applications with a unified declarative model.

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) [26]. A number of major database vendors like IBM and Oracle support MDA and data-driven application development ([5, 17], http://www.oracle.com/technology/products/designer). Current MDA solutions include flexible graphical modeling tools, but unlike Hilda, there is no unified model for end-to-end specification of all application layers. Structured programming is not enforced, and impedance mismatch and conflict detection are only partially addressed.

8 Conclusion

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. 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.

References

[1] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active views for electronic commerce. In *Proc. VLDB*, pages 138–149, 1999.

[2] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo. The deductive database system ldl++. *Theory and Practice of Logic Programming*, 3(1):61–94, 2003.

[3] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring documents, databases, and webs. In *Proc. ICDE*, pages 24–33, 1998.

[4] A. Bongio, S. Ceri, P. Fraternali, and A. Maurino. Modeling data entry and operations in webml. In *The World Wide Web and Databases (WebDB, Selected Papers)*, pages 201–214, 2000.

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

[6] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8(2):215–260, 1997.

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

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

[9] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A verifier for interactive, data-driven web applications. In *Proc. SIGMOD*, 2005. To appear.

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

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

[12] 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.

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

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

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

[16] P. Bernstein et al. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

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

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

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

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

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

[22] Y. E. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: A desktop experiment management environment. In *Proc. VLDB*, pages 274–285, 1996.

[23] C. Lamb, G. Landis, J. A. Orenstein, and D. Weinreb. The ObjectStore database system. *CACM*, 34(10):50–63, 1991.

[24] P. Merialdo, P. Atzeni, and G. Mecca. Design and development of data-intensive web sites: The Araneus approach. *ACM TOIT*, 3(1):49–92, 2003.

[25] C. Mohan. Workflow management in the Internet age. In *Proc. ADBIS*, pages 26–34, 1998.

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