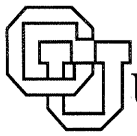
Squirrel Phase 1: Generating Data Integration Mediators that Use Materialization

Gang Zhou
Richard Hull
Roger King

CU-CS-793-95



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Squirrel Phase 1: Generating Data Integration Mediators that Use Materialization*

Gang Zhou† Richard Hull, and Roger King

Computer Science Department
University of Colorado
Boulder, CO 80309-0430
{gzhou, hull, king}@cs.colorado.edu

Technical Report CU-CS-793-95

November 30, 1995

Abstract

This paper presents a framework for data integration that is based on using "Squirrel integration mediators" that use materialization to support integrated views over multiple databases. These mediators generalize techniques from active databases to provide incremental propagation of updates to the materialized views. A framework based on "View Decomposition Plans" for optimizing the support of materialized integrated views is introduced. The paper describes the Squirrel prototype currently under development, which can generate Squirrel mediators based on high-level specifications.

The integration of information by Squirrel-generated mediators is expressed primarily through an extended version of a standard query language, that can refer to data from multiple information sources. In addition to materializing an integrated view of data, these mediators can monitor conditions that span multiple sources. The Squirrel framework also provides efficient support for the problem of "object matching", that is, determining when object representations (e.g., OIDs) in different databases correspond to the same object-in-the-world, even if a universal key is not available.

To establish a context for the research, the paper presents a taxonomy that surveys a broad variety of approaches to supporting and maintaining integrated views.

1 Introduction

Given the advent of the information super-highway, an increasingly important computer science problem today is to develop flexible mechanisms for effectively integrating information from heterogeneous and geographically distributed information sources. The traditional approach is to support a *virtual* integrated view, and to support queries against the view by query decomposition, query shipping, and integration of query results [ACHK93, LMR90, T+90]. More recently, the use of *materialization* has been gaining increasing attention in connection with supporting both single-source and integrated views [ADD+91, IK93, WHW89, ZGHW95]. There are a variety of situations under which materialization is preferable to the virtual approach, e.g., cases where network connectivity is unreliable, where response-time to queries is critical, or where it is cheaper to materialize and incrementally maintain intricate relationships rather than re-compute them each time they are needed for a particular query answer. The primary contribution of the research presented in this paper is the development of a prototype tool called Squirrel, that can generate systems that support data integration using materialized integrated views.

A central component of our framework is the notion of "Squirrel integration mediator"[1]. As detailed below, these provide a variety of mechanisms for supporting and incrementally maintaining materialized integrated views. Integration mediators are implemented as special purpose "active modules" [BDD+95, Dal95]; these are software components whose behavior is specified largely by rules, in the spirit of active databases [WC95]. The rules permit a relatively declarative style of programming, thus increasing reusability and maintainability. The primary components of an integration mediator are

*This research was supported in part by NSF grant IRI-931832, and ARPA grants BAA-92-1092 and 33825-RT-AAS. A shorter version of this paper with less technical details appears in [ZHK95a].

†A student at the University of Southern California, in residence at the University of Colorado.

[1]These mediators are also simply called Squirrel mediators or integration mediators in the rest of this paper.

a local store for the materialized integrated view and auxiliary information, rules for incremental maintenance of the view, and an execution model for applying these rules.

Squirrel mediators extend existing techniques [BLT86, CW91, Cha94, GMS93] for the incremental maintenance of materialized views defined over a single database in two fundamental ways. First, integration mediators can support integrated views over multiple databases, which may be modern or legacy. Second, integration mediators are based on "View Decomposition Plans" (VDPs), which serve as the skeletons for supporting materialized integrated views, providing both data structures for holding the required auxiliary information, and serving as the basis for the rulebase. VDPs provide a broad framework for optimizing support for integrated views, in a manner reminiscent of query execution plans used in traditional query optimization (as described in, e.g., [Ull82]).

The Squirrel prototype can be used to generate integration mediators. Squirrel takes as input the specification of the integrated view to be constructed, expressed in a high-level Integration Specification Language (ISL). The specification includes primarily how the data from various sources is to be integrated. For this purpose, a generalization of a standard query language is used. As output, Squirrel produces an integration mediator. When invoked, the mediator first initializes the integrated view and sends to the source databases specifications of the incremental update information that they are to transmit back to the mediator. Then the mediator maintains the integrated view and answers queries against it. In order to construct Squirrel, we have developed a systematic approach to building integration mediators, that is based largely on the use of VDPs.

A novel feature of the integration mediators generated by Squirrel is that they can provide efficient support for monitoring conditions based on information from multiple sources. This is accomplished by materializing and incrementally maintaining information relevant to these conditions. In this manner, a mediator can send an alert as soon as updates received from the source databases indicate that a condition has been violated.

A second novel feature of Squirrel-generated integration mediators is the support they can provide for "object matching", that is, determining when two object representations (e.g., keys in the relational model or object identifiers in an object-oriented model) from two different databases refer to the same object-in-the-world. In this regard, integration mediators build on previous systems that support full [WHW89, WHW90] or partial [ADD+91, KAAK93] materialization for supporting integrated views. In particular, integration mediators can accommodate a variety of complex criteria for matching objects, including "look-up tables", user-defined functions, boolean conditions, historical conditions, and intricate heuristics.

The current Squirrel prototype is focused on a small portion of the full space of possible approaches to data integration. Indeed, modern data integration applications involve a broad array of issues, including the kinds of data, the capabilities of data repositories, the resources available at the site of the mediator (e.g., storage capacity), and the requirements on the integrated view (e.g., query response time and up-to-dateness). No single approach to supporting data integration can be universally applied. To better understand the impact of those issues on data integration, and provide a larger context within which to understand the Squirrel framework, we include in this paper a survey of issues and techniques that arise in data integration, with an emphasis on those issues that affect approaches based on materialization. This survey is presented in the form of a taxonomy based on several spectra, including for example a spectrum about the degree of materialization, which ranges from fully materialized to fully virtual, and spectra concerning different ways to keep materialized data up-to-date. This taxonomy will be used in the future development of Squirrel, both guiding the choice of extensions, and in permitting modular support for different kinds of features.

The rest of the paper is organized as follows: Section 2 describes related work that this research is based upon. Section 3 gives a motivating example that illustrates several aspects of our approach. Section 4 gives a high level description of the Squirrel framework, including descriptions of the ISL, View Decomposition Plans, and the generation of integration mediators from ISL specifications. Section 5 presents the taxonomy of the space of approaches to data integration. Brief conclusions are given in Section 6.

2 Background and Related Research

This section briefly surveys several technologies that are used in the development of the Squirrel framework for data integration, namely (1) integrated views, (2) object matching, (3) maintenance of materialized views, (4) active databases and active modules, (5) the Heraclitus paradigm, and (6) immutable OIDs for export. The relationships of the Squirrel framework to these investigations are also indicated.

2.1 Integrated views

There is a broad literature on integration of schemas or views. Early work focused on view integration as the basis of a methodology for designing global schemas. [BLN86, BM81].

The Multibase system [SBG+81, DH84] is one of the first systems to support integrated views against multiple databases. This uses a virtual representation of the view, along with query decomposition and query shipping. Several systems based on this approach have followed [LMR90, T+90]. One of the recent multidatabase systems is Pegasus [ADD+91], which applies various aspects of object-oriented database technology to address heterogeneity problems arising in the creation of integrated views. (Pegasus also supports limited materialization; see below.) Another recent system is SIMS [ACHK93], where integrated views are represented in the frame-based model of LOOM [Mac88] (a descendant of KL-ONE [BS85]), and query plans are generated dynamically using the LOOM inference engine. The commercial UniSQL system also provides support for the construction of virtual integrated views.

One of the early projects advocating materialization of integrated views is WorldBase [WHW89, WHW90, WWH90]. This provides mechanisms for selecting, restructuring and merging relevant portions of source databases, with an emphasis on the impact of object identifiers. WorldBase does not provide support for the incremental propagation of updates from the source databases to views.

Integration mediators generated by Squirrel support integrated views using the materialized approach, and provide a variety of mechanisms for keeping materialized data up-to-date. The generalization of the Squirrel framework to support a hybrid of virtual and materialized approaches is currently under way [ZHK95b].

2.2 Object matching

An important aspect of data integration concerns object *matching*, i.e., determining when two object representations (e.g., key-values or object identifiers) in different databases correspond to the same object-in-the-world.

Most systems that support integration using the virtual approach, including e.g., Multibase and SIMS, assume that a universal key (possibly involving derived attributes) is available for performing object matching.

References [KAAK93, WHW90] are among the earliest that consider the problem of object matching in contexts where universal keys are not available. In addition to universal keys, WorldBase [WHW89, WHW90] incorporates *look-up tables* that hold matching information, and introduces *negative keys* which, intuitively, help in deciding if two objects do not match. The Pegasus system [ADD+91, KAAK93] supports the specification of derived functions that, given an object representation from one database of an object-in-the-world, returns the corresponding object representation from a second database. (In [KAAK93], the term 'proxy object' is used to refer to an object representation, and the term 'entity object' is used to refer to an object-in-the-world.) These derived functions might be based on a (materialized) look-up table, or on more complex matching criteria. Pegasus provides elegant access to these derived functions, allowing transparent access to information in one database via object representations from a second database.

The Squirrel framework uses materialization to support a wide variety of intricate matching criteria, that may involve look-up tables, boolean conditions, user-defined functions, and historical information. Also, the high-level mechanisms for specifying these criteria are incorporated into the Squirrel's Integration Specification Language.

The area of user-specified equivalences between objects in the same or different databases raises many subtleties (e.g., see [CZN95]). For the current Squirrel prototype we assume that the match criteria specified by the integration mediator are consistent and define (partial) 1:1 correspondences between the objects in families of corresponding classes.

2.3 Maintenance of materialized views

Several investigations have considered the issue of maintaining materialized views, from both practical and algorithmic perspectives.

On the algorithmic side, several works present algorithms and formalisms for specifying incremental propagation of updates from source data to views, including [QW91, GLT94, BLT86, GMS93, GL95]. A general consensus of these works is to use bags (i.e., multisets) for representing the materialized views and intermediate information, so that incremental update propagation can be more efficient. The formalism underlying the support for incremental update in integration mediators is based on the bag algebra $\mathcal{BA}$ for flat relations (i.e., no nested bags) described in [GL95].

From the systems perspective, active databases are emerging as the paradigm of choice for performing incremental update propagation. For example, references [CW91, Cha94] describe comprehensive frameworks for using active database techniques to support incremental maintenance of materialized views.

Integration mediators use techniques that synthesize and generalize the algorithmic work of [BLT86, GMS93, GL95] and systems work such as [CW91, Cha94]. All of that work is focused on maintaining materialized views over a single database; integration mediators support views over multiple databases. Integration mediators provide a full implementation of an algorithm that is based largely on [BLT86, GMS93, GL95]. Furthermore, the Squirrel prototype can automatically generate these integration mediators from high-level specifications. This paper introduces "View Decomposition Plans" (VDPs) (see Subsection 4.3), which provide the skeleton for

supporting incremental view maintenance in integration mediators. VDPs also provide a starting point for both optimizing and evolving integration mediators.

The Cactis system [HK89] supports incremental update of derived attributes in databases based on a specialized object-oriented model. The Cactis system uses a mix of materialized and virtual derived attributes, and a mix of lazy and "predictive" strategies, in order to optimize the usage of CPU and I/O time. The algorithm is based on the use of a generalization of attributed grammars. We plan to incorporate techniques from CACTIS in a future version of Squirrel.

Reference [ZGHW95] also addresses the problem of supporting materialized views in a warehouse environment. That reference describes how to maintain a materialized join of two or more remote relations, by a combination of receiving incremental updates from the source relations and also polling the source relations. The "Eager Compensation Algorithm" [ZGHW95] is used to overcome subtle forms of inconsistency that might arise in this context. In its current form the Squirrel framework does not support this kind of polling; rather it is assumed that the relevant portions of all classes are materialized and maintained by the integration mediator.

2.4 Active databases and active modules

Active databases [WC95] support rulebases and automatic triggering of rules. The rulebases provide a mechanism for specifying some of the behavior of a database in a relatively declarative fashion. As detailed in [HW92, HJ91], a variety of different execution models have been developed for rule application. This variety of alternatives highlights the fact that the "knowledge" represented in an active database stems from two distinct components: the rulebase and the execution model [Abi88].

A recent generalization of active databases is the concept of *active module* [Dal95, BDD+95]. An active module is a software module that incorporates:

- a rulebase, that specifies the bulk of the behavior of the module in a relatively declarative fashion;
- an execution model for applying the rules (in the spirit of active databases);
- (optionally) a local persistent store

An active module can be viewed as capturing some of the spirit and functionality of active databases, without necessarily being tied to a DBMS. While an active module might be a full-fledged active database, it might also be a lightweight process supporting a focused family of functionalities. In particular, the separation of rules (logic/policy) from execution model (implementation/mechanism) allows a more declarative style of program specification, and facilitates maintenance of the active module as the underlying environment evolves. Reference [Dal95] describes an implemented prototype system that uses several active modules with different execution models to support complex interoperation of software and database systems.

As noted in the Introduction, integration mediators are a specialized class of active modules. The execution model currently used by integration mediators was carefully designed for this application to permit the rulebase to be highly modular and focused on semantic manipulations (see Subsection 4.4).

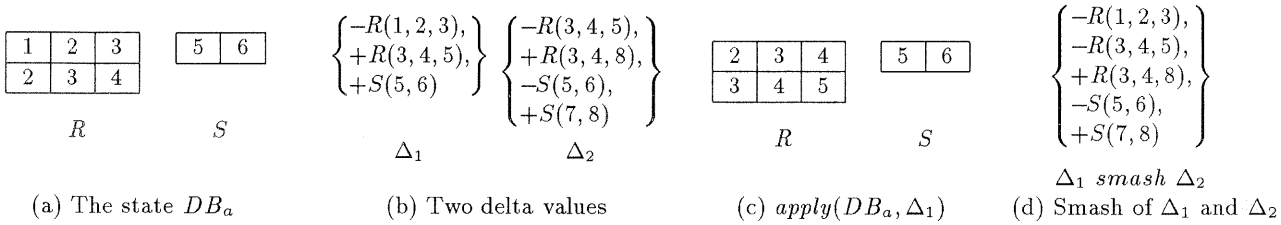
2.5 The Heraclitus paradigm

Another important tool used by the Squirrel prototype is the Heraclitus paradigm [HJ91, GHJ94] for database programming languages (DBPLs). In particular, this paradigm provides notation and constructs for the rules used in integration mediators. The paradigm also served as a key enabling technology in the development of active modules.

Heraclitus[Alg,C] [GHJ+93, GHJ94] is an implemented DBPL that supports the Heraclitus paradigm in the context of the relational model. A team at University of Colorado, Boulder, is currently developing Heraclitus[OO], abbreviated H2O, [BDD+95, DHDD95], which extends and generalizes the paradigm to object-oriented databases. The rules and execution model for integration mediators (Subsection 4.4) are currently implemented in Heraclitus[Alg,C]. We expect the port to H2O to be straightforward as that becomes available.

Fundamental to the Heraclitus paradigm is the notion of *delta value*, often called simply *delta*, that corresponds to a difference between database states. In the pure relational model (no tuple identifiers and no duplicates), delta values are represented as sets of atomic inserts and atomic deletes (possibly involving more than one relation), with the restriction that a delta value cannot contain both $+R(t)$ and $-R(t)$ for relation R and tuple t. (The special delta value $fail$ is used to denote inconsistency.) To illustrate, suppose that R has arity 3 and S has arity 2, with all columns of type integer, and that the "current" state is DB_a as shown in Figure 1(a). Two delta values are shown in part (b) of that figure. The result of *applying* Δ_1 to state DB_a is the state $apply(DB_a, \Delta_1)$ shown in part (c) of that figure. Note that the atomic insert $+S(5,6)$ in Δ_1 has no effect (because we are using the pure relational model). We call such atomic inserts *redundant*; atomic deletes may also be redundant.

An important operator in Heraclitus is *smash*, which provides a form of composition for deltas. More precisely,



(a) The state DB_a (b) Two delta values (c) $apply(DB_a, \Delta_1)$ (d) Smash of Δ_1 and Δ_2

Figure 1: Examples of constructs from Heraclitus

the smash of two deltas Δ and Δ' has the property that for each state DB,

$$apply(DB,\ \Delta\ smash\ \Delta') \ =\ apply(\ apply(DB, \Delta),\ \Delta')$$

In the relational case the smash of two deltas Δ and Δ' can be computed by taking the union of the deltas, resolving conflicts in favor of Δ' (i.e., eliminating those atomic updates of Δ that conflict with an atomic update of Δ'). For example, Δ_1 *smash* Δ_2 is shown in Figure 1(d). Smash is associative.

Another operator of Heraclitus is `when`, which permits hypothetical access to deltas. If ϵ is an arbitrary side-effect free expression (e.g., a query, or an expression whose value is a delta) and δ is an expression whose value is a delta, then ϵ `when` δ has the value that ϵ *would* have, if (the value of) δ *were* applied to the current database state. The implementation of Heraclitus[Alg,C] [GHJ94, ZGH94] supports such hypothetical access in an efficient manner, without modifying the database state in which the expression is evaluated.

The above discussion was for the pure relational model; the current implementation of Heraclitus[Alg,C] also includes deltas for the relational model with duplicates, i.e., for bags of tuples. In the context of integration mediators a delta for a bag is a bag of atomic inserts and deletes, with the restriction that a delta value cannot contain both one or more occurrences of $+R(t)$ and one or more occurrences of $-R(t)$ for relation R and tuple t. (In the general setting, the form of deltas for bags is more intricate [DHR95].) The operators *apply*, *smash*, and `when` are defined analogously to the pure relational case. Integration mediators generated by the current Squirrel prototype use deltas and the *apply*, *smash*, and `when` operators, for both pure relations and for bags of tuples.

2.6 Immutable OIDs for export

One subtlety concerning object identifiers (OIDs) is that from a formal perspective, only the relationship of the OID to values and other OIDs in a database instance is important [Bee89]; the particular value of an OID is irrelevant. As a result, a DBMS is free to change the specific values of OIDs, as long as its internal state remains "OID-isomorphic" [AHV95] to the original state. This may create a problem if OIDs from a source database are used to represent information in the local store of an integration mediator.

To overcome this problem, we generally assume that the physical OIDs associated with a given entity class in a source database are immutable. If a source database does not use immutable OIDs, then we follow the technique of [EK91], and assume that these source databases have been wrapped to support immutable OIDs for export. (Reference [EK91] uses the phrase 'global OIDs' for this.) A simple approach to accomplish this is for the source database to maintain a binary relation with first coordinate holding internal, physical OIDs, and second coordinate holding symbolic "export" OIDs.

3 A Motivating Example and Intuitive Remarks

This section gives an informal overview, based on a very simple example, of several key aspects of the Squirrel framework for data integration using integration mediators. Section 4 describes the Squirrel framework in more detail.

In the example there are two databases, `StudentDB` and `EmployeeDB`, that hold information about students at a university and employees in a large nearby corporation, respectively. An integration mediator, called here `S_E_Mediator`, will maintain an integrated view about persons who are both students and employees, providing their names, majors, and names of the divisions where they work. The mediator will also monitor the condition that no more than 100 students are employees.

Figure 2 gives a high level specification (in our ISL language, see Section 4.2) of the data integration problem. This ISL specification includes primarily the relevant subschemas of the two source databases (in the `Source-DB` parts), and the definition of the integrated view (in the `Export classes` part). In this example the view consists of only one class; in general the view might include several classes. In the example, there is not a universal key between students and employees. However, the ISL specification includes a description of how students and employees can be matched, in the `Correspondence` part (see below). Note that the function `S_E_match` defined by that correspondence is

```
Source-DB: StudentDB
    interface Student {
        extent     students;
        string     studName;
        integer[7] studID;
        string     major;
        string     local_address;
        string     perm_address;
    };
    key: studID

Source-DB: EmployeeDB
    interface Employee {
        extent     employees;
        string     empName;
        integer[9] SSN;
        string     divName;
        string     address;
    };
    key: SSN
```

```
Correspondence S_E_match:
    Match classes:
        s IN StudentDB:Student,
        e IN EmployeeDB:Employee
    Match predicates:
        close_names(s.studName, e.empName)
        AND (e.address = s.local_address
        OR e.address = s.perm_address)
    Match object files:
        $home/demo/close_names.o

Export classes:
    DEFINE VIEW Student_Employee
    SELECT s.studName,s.major,e.divName
    FROM s IN StudentDB:Student,
         e IN EmployeeDB:Employee
    WHERE S_E_match(s,e);

Conditions:
    Condition:
        count(Student_Employee) =< 100
    Action:
        send_warning('count exceeded')
```

Figure 2: The ISL specification of the example problem

used in the specification of the view. Finally, the `Conditions` part of the ISL specification includes the condition to be continuously monitored.

We now consider in more detail how `S_E_Mediator` (a) provides support for object matching, (b) uses rules to support incremental maintenance of materialized data, and (c) monitors the condition.

With regards to issue (a), the `Match predicate` in the `Correspondence` part of the ISL specification incidates that a student object s *matches* an employee object e if (1) either s.`local_address` = e.`address` or s.`perm_address` = e.`address`, and (2) their names are "close" to each other according to some metric, for instance, where different conventions about middle names and nick names might be permitted. The "closeness" of names is determined by a user-defined function, called here `close_names()`, that takes two names as arguments and returns a boolean value. (More intricate match criteria can also be supported.)

Following the default approach used by Squirrel, object matching between students and employees is supported in `S_E_Mediator` by having the local store hold a "match" class, in this case called `match_Stud_Emp`, that essentially holds the "outer join" of the `Student` and `Employee` classes. For each person who is both student and employee there will be one "surrogate" object in `match_Stud_Emp` that represents this person; for each person who is a student but not an employee there will be one "surrogate" object in `match_Stud_Emp`, several of whose attributes will be `nil`; and likewise for employees who are not students. This match class is used by the integration mediator to support the derived boolean relation `S_E_match` referred to in the definition of the view class `Student_Employee`.

The class `match_Stud_Emp` illustrates one kind of intricate relationship between data from multiple sources which is expensive to compute. By using materialization, this relationship can be computed when `S_E_mediator` is initialized, and then maintained incrementally as relevant data in the source databases changes. In general, the query response time obtained by using this materialized approach to data integration will be faster than when using the virtual approach, where the potentially expensive step of identifying matching pairs of objects may be incurred with each query against the view. Also, we expect that if the update-to-query ratio is sufficiently small, then the materialized approach will also be more efficient on average than the virtual approach.

In this example, the export class `Student_Employee` of the view is a simple projection and selection of the class `match_Stud_Emp`. Thus, `S_E_Mediator` can support this class in a virtual fashion, translating queries against the view into queries against `match_Stud_Emp`. In general, an integration mediator may materialize some export view classes, and support others as selections and projections of other materialized classes.

We now turn to issue (b), that of incrementally maintaining materialized data in the integration mediator. Two basic issues arise: (i) importing informa-

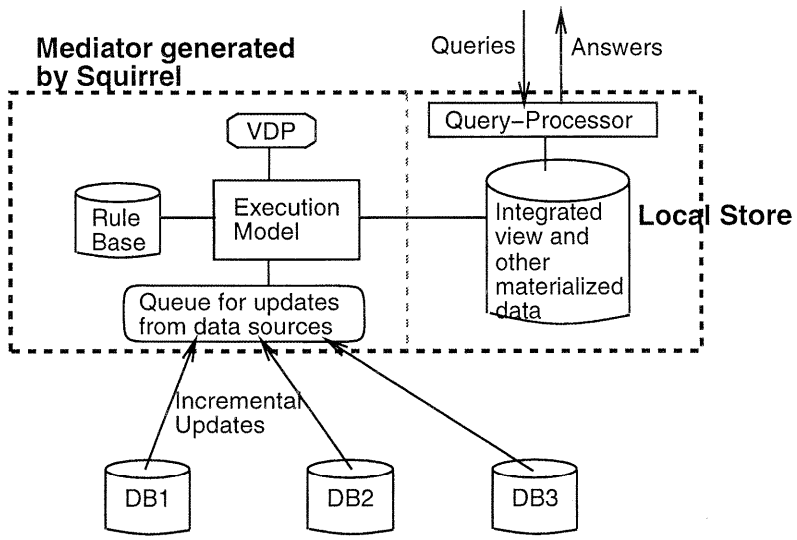


Figure 3: Configuration of an integration mediator connected with three source DBs

tion from the source databases and (ii) correctly maintaining the materialized data to reflect changes to the source databases. For this example, with regards to (i) we assume that both source databases can actively send messages containing the net effects of updates (i.e., insertions, deletions, and modifications) to `S_E_Mediator`. A rulebase in the integration mediator is used to perform (ii). To illustrate briefly, we informally describe two representative rules involved in supporting the class `match_Stud_Emp`. The two rules correspond to the creation of new `Student` objects in the source database `StudentDB`.

Rule `R1`: If an object of class `Student` is created, insert a corresponding new object into class `match_Stud_Emp` whose Employee-attributes are `nil`.

Rule `R2`: Upon the insertion of a `match_Stud_Emp` object x with `nil` Employee-attributes, if there is a corresponding object y in `match_Stud_Emp` with `nil` Student-attributes that matches x, then delete x and modify y by replacing its `nil` attributes with values from x.

The complete rulebase would include rules dealing with creation, deletion, and modification of objects in both source databases. Subsections 4.5 and 4.6 describe how this is generalized to support n-ary matches and conventional query-langauge operators (e.g., selection, join, etc.), and how the rulebase is generated automatically by Squirrel.

Finally, we indicate (c) how the condition `count(Student_Employee) =< 100` is monitored. This is a particularly simple case, because the only class mentioned in the condition is one of the export view classes. In this case, the condition is monitored by rules that incrementally maintain the count of tuples in `match_Stud_Emp` that have no `nil` values. More generally, a condition may refer to data that is not represented by any of the export view classes. In that case, the mediator materializes the classes holding data relevant to the condition, and rules are used to incrementally maintain these classes and monitor the truth-value of the condition on them.

Importantly, the integration mediator can alert a user that the condition has been violated as soon as the relevant updates to the source databases are transmitted to the mediator. If a virtual approach to supporting the integrated view were used, then the condition could be monitored only by periodically asking a query that called for the count of `Student_Employee`. This would involve repeated accesses to the two source databases, and might not alert the user of violation of the condition as quickly as the materialized approach.

4 The Squirrel Integration Mediator Generator

We are currently developing a prototype tool called Squirrel that generates integration mediators. Squirrel takes as input a high-level specification of an integrated view to be supported, and produces as output an integration mediator that supports it. One of the challenges in designing Squirrel was to develop a systematic and uniform methodology for constructing integration mediators from high-level specifications. In this section we describe

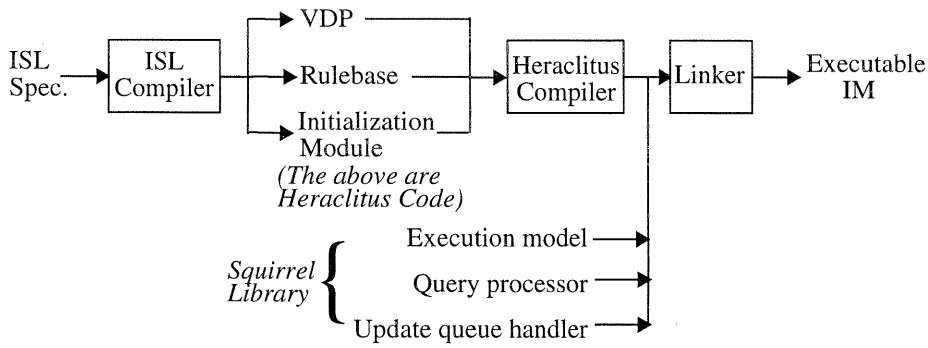


Figure 4: The process of automatically generating integration mediator from an ISL specification

both the methodology and the integration mediators that are produced by it.

The section begins with a high-level description of how Squirrel generates integration mediators (Subsection 4.1). Next, the high-level Integration Specification Language (ISL) (Subsection 4.2) is presented. The skeleton of a Squirrel-generated integration mediator is provided by its View Decomposition Plan (VDP); these are described in Subsection 4.3. The next two subsections (4.4 and 4.5) describe the execution model used by integration mediators, and also indicate how incremental updates are propagated through the various materialized classes stored by these mediators. Subsections 4.6 and 4.7 describe how VDPs and rulebases are constructed. Subsection 4.7 also touches on the issue of evolving an integration mediator. A final component of our solution is the automatic generation of rules to be incorporated into the rulebases of the source databases, so that relevant updates will be propagated to the mediator. We do not address the generation of those rules here.

4.1 An overview of the automatic generation of integration mediators

This subsection gives a brief overview of how Squirrel generates integration mediators. Further detail is given in the subsequent subsections, where the various components of integration mediators are described.

The overall architecture of a Squirrel-generated integration mediator is shown in Figure 3. An integration mediator consists of six components – an update-queue that holds incremental updates from remote information sources, a VDP, a rulebase, an execution model, a local persistent store, and a query processor that accepts queries against the mediator. Communication between the mediator and remote sources is based on the Knowledge Query and Manipulation Language (KQML) [FWW+].

There are two kinds of information flow within an integration mediator. One involves incremental updates against the source databases, which flow into the queue; as a result of the execution model (applied to the rulebase and VDP) these incremental updates then propagate into the integrated view. The other kind of information flow involves queries posed against the integrated view, and answers made in response to them. Importantly, humans and processes that query the integration mediator need only be aware of the query processor and the local store, i.e., the part of the integration mediator shown on the right side of the gray dashed line in Figure 3.

The process of generating integration mediators from an ISL specification is illustrated in Figure 4. The software modules corresponding to the components of an integration mediator can be divided into two groups with regards to the construction of integration mediators. The first group includes three modules, namely the execution model, query processor, and update-queue handler. These modules are independent from any particular ISL specification and are kept in the *Squirrel library*. The second group of modules includes the VDP, the rulebase, and the initialization module. The latter initializes the local store and (possibly) creates rules for the remote source databases. Those modules must be tailored to particular ISL specifications, and are generated dynamically by Squirrel's *ISL compiler* from the ISL specification. More specifically, the ISL compiler reads in an ISL specification and outputs the three modules in Heraclitus[Alg,C] code. As mentioned in Subsection 2.5, Heraclitus[Alg,C] is a database programming language that provides notation and constructs that are convenient for implementing various software modules of the integration mediator. Since these modules are in Heraclitus[Alg,C] code which is relatively high-level, the user has the freedom to modify them, e.g., by adding new rules or modifying the VDP. The final executable integration mediator is created by pushing the three generated modules through the Heraclitus[Alg,C] compiler, and linking the result with the modules from the Squirrel library. In the remainder of this section we discuss the ISL and the

```
<ISL>              ::= <Src-Subschema>+ <Correspondence>*
                         [<Internal>] <Export> [<Conditions>]
<Src-Subschema>    ::= Source DB: <string> {<ODL-Class-Def> [<Key>]}+
<Key>              ::= key: <string> {, <string>}*
<Correspondence>   ::= Correspondence <string>: <Match>+
<Match>            ::= Match classes: <string> IN <string>
                                {, <string> IN <string>}+
                       Match predicates: <OQL-Condition>
                       [Match object files: <string> {, <string>}*]
<Internal>         ::= Internal classes: <OQL-View-Def>
<Export>           ::= Export classes: <OQL-View-Def>
<Conditions>       ::= Conditions: {<Condition>
                         <Action> [<Action-Obj-File>]}+
<Condition>        ::= Condition: <OQL-Condition>
<Action>           ::= Action: <string>
<Action-Obj-File>  ::= Action object files: <string> {, <string>}*
```

Figure 5: Grammar for Integration Specification Language (ISL)

three most important components of integration mediators, namely, the VDP, the execution model, and the rulebase.

4.2 Squirrel Integration Specification Language (ISL)

The Integration Specification Language (ISL) allows users to specify their data integration applications in a largely declarative fashion. The primary focus of ISL is on the specification of matching predicates, of integrated views, and of conditions to be monitored. In the current version of ISL, users can specify (1) (relevant portions of) source database schemas; (2) the predicates to be used when matching objects from families of corresponding classes in the source databases; (3) distinguished classes, which include both classes for export, and additional "internal" classes that are explicitly defined in the ISL; and (4) conditions to be monitored by the integration mediator. Internal classes might be defined because they are used in the specification of monitored conditions, or because they serve as intermediate classes from which other distinguished classes are defined. As will be seen in Subsection 4.7, internal classes can also provide hints to the ISL compiler, so that the VDP can avoid redundant work.

An extended BNF grammar for the top level of ISL is given in Figure 5, and the ISL specification for the Student/Employee example is shown in Figure 2 in Section 3. The ISL is based on the ODL and OQL of the ODMG [Cat93] standard.

We now consider the four parts of an ISL specification in slightly more detail.

(1) Source DB subschemas: These describe relevant subschemas of the source databases using the Object Definition Language (ODL) of the ODMG standard [Cat93]. <ODL-Class-Def> denotes a class definition in ODL syntax. A key may optionally be specified for each class.

(2) Correspondence specifications: These describe match criteria between objects of families of corresponding classes. A correspondence specification for a given family of classes has three parts:

Match classes: This lists the classes that are matched in this specification, and indicates the ranges of variables used in the match predicates.

Match predicates: A *binary matching predicate* specifies correspondences between objects from two classes. We use <OQL-Condition> (this is denoted <query> in p. 79 of [Cat93]) to specify such predicates. The predicates can be based on, among other things, boolean relations or user-defined functions (that may in turn refer to "look-up tables" or intricate heuristics). In the case of n-ary matching, the full correspondence is expressed using a set of binary match predicates. Although not shown here, we are developing extensions to incorporate historical conditions and heuristics expressed as rules.

Match object files (optional): specifies the path(s) of the object file(s) containing the implementation of user-defined comparison function(s).

(3) Distinguished classes: This part of the ISL defines the export classes and internal classes. Distinguished classes are specified by <OQL-View-Def>, which extends OQL to have view definition capabilities. The definition of a distinguished class may refer

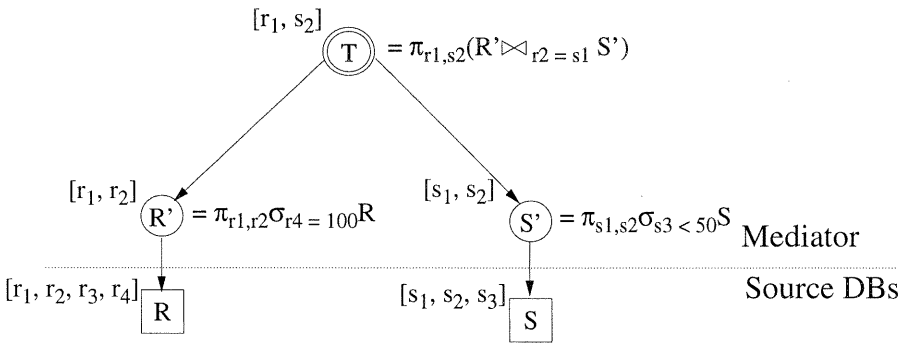


Figure 6: A VDP of a view $T = \pi_{r_1,s_2}(\sigma_{r_4=100}R \bowtie_{r_2=s_1} \sigma_{s_3<50}S)$

both to source database classes and to distinguished classes that are already defined.

(4) **Conditions:** Finally, rules are included for monitoring conditions. The conditions are specified by <OQL-Condition>, these may refer to source classes, distinguished classes, and or user-defined functions.

4.3 View decomposition plans (VDPs)

The skeleton of a Squirrel-generated integration mediator is provided by its View Decomposition Plan (VDP). A VDP specifies the classes (both distinguished and other) that the integration mediator will maintain, and provides the basic structure for supporting incremental maintenance. As noted in the Introduction, VDPs are analogous to query execution plans as used in query optimization [Ull82]. This subsection presents a definition of VDP and gives several examples.

As will be defined formally below, the VDP of an integration mediator is a directed acyclic graph (dag) that represents a decomposition of the integrated view supported by that integration mediator. The leaf nodes correspond to classes in the source databases, and the other nodes correspond to derived classes which are materialized and maintained by the integration mediator. Some non-leaf nodes, including all maximal nodes of the VDP, correspond to the distinguished (i.e., export and internal) classes in the ISL specification for the integration mediator. An edge from node u to node v in a VDP indicates that the class of v is used directly in the derivation of the class of u. In general, the propagation of incremental updates will proceed along the edges, from the leaves to the top of a VDP. Analogous to query execution plans, different VDPs for the same ISL specification may be appropriate under different query and update characteristics of the application.

The language currently supported by Squirrel for specifying integrated views includes rich object matching criteria and a subset of ODMG's OQL that corresponds to the relational algebraic operators selection, projection, join, union, and set difference, where both imported and exported classes may be sets or bags. In the discussion here we focus on the case where the imported and exported classes are sets; the extension to bags is straightforward. Importantly, even in the case where imported and exported classes are restricted to be sets, some of the classes stored inside an integration mediator may be bags; this occurs if the integrated view involves projection or union. The framework developed here can be used with both the object-oriented and relational data models. For the sake of conciseness, we describe the framework by using the relational algebra syntax, which can be mapped to the OQL syntax.

Formally, a VDP is a labeled dag $\mathcal{V} = (V, E, class, Source, def, Dist)$ such that:

1. The function *class* maps each node $v \in V$ into a specification of a distinct class, which includes the name of the class and its attributes. We often refer to a node v by using the name of $class(v)$.

2. *Source* is a possibly empty subset of V that contains some or all of the leaves of the dag. Nodes in *Source* correspond to classes in the source databases, and are depicted using the □ symbol. Other nodes are depicted using a circle. In a "complete" VDP each leaf is a source database class; VDPs whose leaves are not source database classes are used in Subsections 4.6 and 4.7 to describe the construction of "complete" VDPs.

3. An edge $(a, b) \in E$ indicates that $class(a)$ is directly derived from $class(b)$ (and possibly other classes).

4. For each non-leaf $v \in V$, $def(v)$ is an expression in the view definition language that refers to $\{class(u) \mid (v, u) \in E\}$. Intuitively speaking, $def(v)$ defines the population of $class(v)$ in terms of the classes corresponding to the immediate descendants of v.

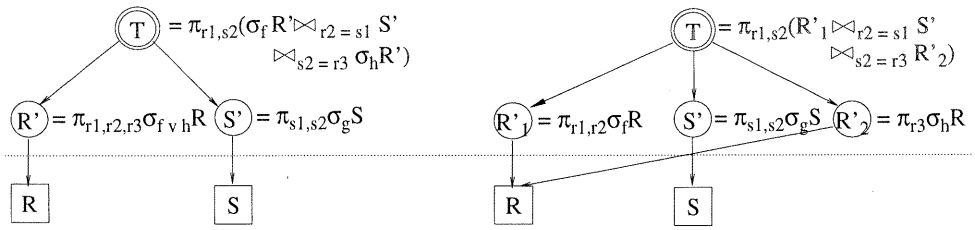


Figure 7: Two alternative VDPs of a view $T = \pi_{r_1,s_2}(\sigma_f R \bowtie_{r_2=s_1} \sigma_g S \bowtie_{s_2=r_3} \sigma_h R)$

The expressions used to define a class in terms of other classes are restricted. Specifically, a non-leaf class can be defined to be either (a) a projection and/or selection of another class (this includes the degenerate case where a source class is replicated as a non-leaf node); (b) a (projection and/or selection of a) join of (projections and/or selections of) two or more non-source classes; (c) a union of (projections and/or selections of) two or more non-source classes; (d) a difference of (projections and/or selections of) two non-source classes; or (e) a match class based on two or more source classes. (Analogous to query execution plans, it is sometimes useful to combine several operators into a single node of a VDP; see Example 4.2. However, some restrictions apply to ensure that incremental update propagation can be performed in a systematic manner.) Non-leaf nodes of the first three kinds are called *bag nodes*, and of the latter two kinds are called *set nodes*. This is because the classes associated with the first three kinds of nodes will be stored as bags, while the classes associated with the other two kinds of nodes will be stored as sets.

5. $Dist \subset V$ denotes the set of distinguished classes. These correspond to the internal and export classes specified in the ISL. Each maximal node (i.e., node with no in-edges) is in $Dist$; other non-source nodes may also be in $Dist$. Elements of $Dist$ are depicted using a double circle.

Example 4.1: Let $R(r_1, r_2, r_3, r_4)$ and $S(s_1, s_2, s_3)$ be two classes from distinct databases. Suppose that the integrated view for an integration mediator has the single class $T = \pi_{r_1,s_2}(\sigma_{r_4=100}R \bowtie_{r_2=s_1} \sigma_{s_3<50}S)$. A VDP for T is shown in Figure 6. The dotted line separates the mediator classes from the source database classes. There are three non-leaf classes in the VDP, namely T, R', and S'. The attributes of the classes are shown to the upper left of the non-leaf nodes. R' and S' serve as auxiliary data, so that T can be maintained using incremental updates from the source databases and information local to the mediator. (This contrasts with the approach of [ZGHW95], where only T would be materialized. Under that approach, T is maintained using incremental updates from the source databases and polling of the source databases.) Each of T, R' and S' are bag nodes. □

The above example gives a VDP for a very simple integrated view. In the next example, we show that more than one VDP might be used to represent the same integrated view.

Example 4.2: Based on classes R and S of Example 4.1, an export class is defined as: $T = \pi_{r_1,s_2}(\sigma_f R \bowtie_{r_2=s_1} \sigma_g S \bowtie_{s_2=r_3} \sigma_h R)$. Figure 7 gives two alternative VDPs, $\mathcal{V}_1$ and $\mathcal{V}_2$, of the view T. For the sake of conciseness, we omit the attribute sets of the VDPs in the figure. The three-way join T involves two occurrences of R. $\mathcal{V}_1$ uses one projection/selection of R to support both of these occurrences, whereas $\mathcal{V}_2$ uses two separate projection/selections of R. The join in $\mathcal{V}_2$ can use R'_1 and R'_2 directly, while the join in $\mathcal{V}_1$ must use selections of R'. It is typical that when nodes (such as R) are shared in a VDP, then projection and/or selection may need to be applied to them as part of a node based on join, union, or difference.

The combination of R'_1 and R'_2 will generally take less space than R'. For example, if an object (a_1, a_2, a_3) satisfies only the selection condition f, the whole object is in the class R', but only the projection (a_1, a_2) would be in R'_1. On the other hand, incremental maintenance of R' may be more efficient than that of R'_1 and R'_2, because an update to the class R needs to be processed only once in the former case. Each of the non-leaf nodes in both VDPs here are bag nodes. □

In the previous examples, the integrated view involves only one export class. In the next example we give a complex VDP involving several distinguished nodes.

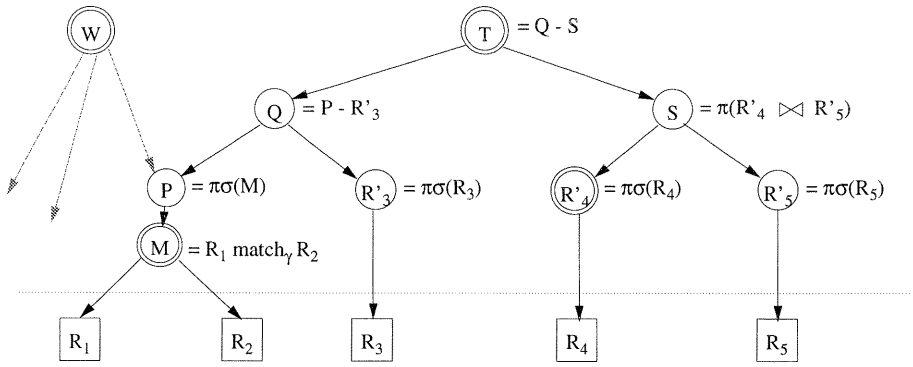


Figure 8: $T = (\pi\sigma(R_1\ match_\gamma\ R_2) - \pi\sigma(R_3)) - \pi\sigma(R_4 \bowtie R_5)$

Example 4.3: Among other export classes of some integration mediator, let class T be defined as $T = (\pi\sigma(R_1\ match_\gamma\ R_2) - \pi\sigma(R_3)) - \pi\sigma(R_4 \bowtie R_5)$. Here $M = (R_1\ match_\gamma\ R_2)$ is a match class based on matching predicate γ (as illustrated in the Student/Employee example in Section 3), which contains object correspondence information for objects in classes R_1 and R_2. To simplify the exposition, selection and join conditions and projection attributes are omitted in the view definition. A VDP supporting this class is shown in Figure 8. The VDP includes three distinguished classes, W, R'_4, and M, in addition to T. The grey edges coming from W indicates that W also relies on other classes not shown in the figure. In this VDP, M, Q, and T are set nodes, and P, R'_3, R'_4, R'_5 and S are bag nodes. □

4.4 An execution model for integration mediators

Together this and the next subsections present the execution model used by integration mediators generated by the current Squirrel prototype. The execution model is called the "bottom-up VDP-based execution model" (BV execution model). This is just one of several possible and reasonably efficient execution models; a topic of continuing research is the comparison of alternative execution models. This subsection focuses on how the execution model supports the traditional query operators (selection, projection, join, union, difference), and the following subsection describes how support for object matching is incorporated.

As noted in Section 2, the approach to maintaining integrated views presented here (based on the execution model, VDPs, and rulebases) provides a systematic and comprehensive implementation of an algorithm that follows the spirit of and generalizes the algorithms of [BLT86, GMS93, GL95] for maintaining materialized views over a single database, using the active paradigm as in [CW91, Cha94].

As with all active databases, the BV execution model permits a separation of the logic of an integration mediator from the control. As with other active databases, the control aspect of the integration mediator is performed by the execution model. Unlike most other active databases, the logic of an integration mediator is found in two components: the rulebase and the VDP. In essence, the execution model uses the VDP to guide the order of rule application. Furthermore, the rulebase is closely related to the VDP. In general, each rule specifies how updates are propagated along a specific edge of the VDP. (The support for match classes is slightly more intricate.) In the execution model, rules are fired in a set-at-a-time forward-chaining manner, in a deferred mode reminiscent of some aspects of the AP5 [Coh86] and Starburst Rule System [CW90] execution models.

The BV execution model ensures that incremental updates of source data are correctly reflected in the integrated view. The model offers some freedom with regards to the specific order in which rules are fired, thereby offering opportunities for optimization at that level. The BV execution model is currently implemented using Heraclitus[Alg,C].

We now describe several aspects of the BV execution model in some detail, and then present its specification. The execution model uses several notions from the Heraclitus paradigm, including deltas and the *smash* and `when` operators (see Subsection 2.5).

A *VDP-rulebase* is a pair ($\mathcal{V}$,*edge_rule*), where

(a) $\mathcal{V} = (V, E, class, Source, def, Dist)$ is a VDP; and

(b) *edge_rule* is a function that maps each edge in E to a rule (or a set of rules, in case the edge is from a leaf node into a match-class node).

A description of the rules, and how they are generated, is given in Subsection 4.6. The following description of

the BV execution model provides important context for understanding those rules.

Speaking at an abstract level, incremental updates will arrive in the queue of the integration mediator in a serial but asynchronous manner. We assume that each incremental update arriving to the queue is in the form of a delta against one or more of the classes in a single source database. (A simple optimization would be to let the source databases filter the updates, so that they correspond to deltas against the lowest non-leaf nodes of the VDP.)

Each invocation of the execution model will form a separate transaction. Let the sequence of starting times of these transactions be $t_1, t_2, \ldots$. These are called *execution invocation times*. We require for each i that t_{i+1} is a point in time that is after the completion of the transaction for time t_i. By the phrase "*the state of the source databases at time* t_i" we mean the state of the source databases as reflected by the updates they have reported to the integration mediator up to time t_i.

Suppose that an integration mediator with VDP $\mathcal{V}$ has been deployed. Two repositories are associated with each non-leaf node v of $\mathcal{V}$. Suppose that $class(v) = R$. The first repository is denoted simply as 'R', and holds the "current" population of class R. The second repository is denoted by 'ΔR', and holds the smash of incremental changes for R that result from the incremental propagation of updates during a single execution of the execution model. Recall that classes of bag nodes (i.e., those defined using selection and/or projection and/or join, or union) will hold bags, and set nodes (i.e., those defined using difference or matching) will hold sets.

Before continuing, it is convenient to extend the function *edge_rule* in the definition of VDP to apply to nodes as follows:

$$edge\_rule(v) = \{edge\_rule(v', v) \mid (v', v) \in E\}$$

Intuitively, $edge\_rule(v)$ holds all rules of in-edges to v, i.e., all rules that propagate updates out of v to its parents.

Let v be a node with $class(v) = R$. By the phrase "*process node* v" we mean to fire all eligible rules in $edge\_rule(v)$ (in any order) and then to execute the following steps:

$$\begin{aligned} R &:= apply(R, \Delta R); \\ \Delta R &:= \emptyset; \end{aligned}$$

In the BV execution model a node v will not be processed until all of its children have been processed. As a result, all incremental changes to a node v are accumulated before any of these changes are propagated to parents of v.

The execution model enforces several properties that make it easy to reason about the net effect of applications of the rulebase. These are now described. At an execution invocation time t_i, each of the ΔR repositories is empty, and for each class R, the repository R holds the relevant parts of the population of R, according to the equation defining R and the contents of the source databases at time t_{i-1}. The first step of the invocation of the execution model at time t_i involves emptying the input queue and propagating the impact of the queue to the lowest non-leaf nodes of $\mathcal{V}$. In fact, this step populates the Δ repositories of these nodes, to accurately hold the deltas for these classes that are implied by the queue contents between times t_{i-1} and t_i. In subsequent steps of rule firing, the following properties will hold:

(A) If a non-leaf node v with $class(v) = R$ has been processed, then

  (i) the corresponding repository R will hold the population for R that reflects the state of the source databases at time t_i, and

  (ii) the associated repository ΔR is empty.

(B) If a non-leaf node v with $class(v) = R$ has not been processed, then

  (i) the corresponding repository R will hold the population for R that reflects the state of the source databases at time t_{i-1}.

  (ii) the associated repository ΔR may hold information corresponding to some or all of the incremental updates implied for class R by the incremental updates to the source databases reported between times t_{i-1} and t_i.

We now present the BV execution model for VDPs that do not have match classes. Suppose that an integration mediator has been deployed with VDP-rulebase $\mathcal{R} = (\mathcal{V}, edge\_rule)$, where $\mathcal{V} = (V, E, class, Source, def, Dist)$. We assume that the queue holding incremental updates from the source databases is nonempty. The execution model proceeds as follows:

(1) Initialization: Let Δ hold the smash of all incremental updates held in the queue at a given time t. Δ can be broken into a set $\Delta R_1, \ldots, \Delta R_k$ of subdeltas that refer to some set $R_1, \ldots, R_k$ of source database classes that are associated with leaf nodes $v_1, \ldots, v_k$ (respectively) of $\mathcal{V}$.

  During this phase, two things occur:

(1a) All eligible rules in $\cup\{edge\_rule(v_i) \mid i \in [1, k]\}$ are fired, in any order.

(1b) All entries in the queue that contributed to Δ are deleted. (It may be that during the execution of step (1a) additional deltas were added to the queue. These will remain in the queue until the next cycle of rule firing is initiated.)

(2) "Upward" traversal of (V, E): During this phase each non-leaf node is processed in an order that satisfies the following restriction:

> A node v cannot be processed until all of its children have been processed.

It is straightforward to verify that when using rules generated by the templates of Subsection 4.6, executions of BV execution model satisfy the properties listed above.

4.5 Providing support for n-ary matches

The Student/Employee example of Section 3 gave an overview about how binary match classes are supported in integration mediators. This subsection presents the general framework used to support n-ary match classes, and describes how the BV execution model is generalized to accommodate this.

Suppose now that classes $A_1, \ldots, A_n$ from various source databases represent the same or overlapping sets of objects-in-the-world. An integration mediator can support matching of objects from these classes by maintaining a match class $match\_A_1\_\ldots\_A_n$. Each of the source classes will contribute three kinds of attributes to the match class (these sets may overlap):

identification attributes: These are used to identify objects from source databases. These might be printable attributes known to be keys, or might be immutable OIDs from the source databases (see Subsection 2.6). Although OIDs are not technically attributes, we view them as such here.

match attributes: These are the (possibly derived) attributes referred to in the match predicates.

data attributes: These are attributes that are used in distinguished classes or by other intermediate classes in the VDP.

Speaking loosely, the class $match\_A_1\_\ldots\_A_n$ will hold an "outer join" of the underlying source classes, where each object in $match\_A_1\_\ldots\_A_n$ represents a single object-in-the-world. Each element of $match\_A_1\_\ldots\_A_n$ is called a *surrogate* object. A given surrogate object might represent objects from essentially any subset of the associated source database classes.

The interface of the match class `match_Stud_Emp` for the Student/Employee example of Section 3 is shown in Figure 9. The left column of 5 attributes of this class come from the `Student` class; the other 4 attributes in the right column come from the `Employee` class. The identification attributes are `studID` and `SSN`, which are printable keys; the match attributes are `studName`, `local_address`, `perm_address`, `empName`, and `address`; and the data attributes are `studName`, `major`, and `divName`.

The use of a match class such as $match\_A_1\_\ldots\_A_n$ is just one possible way of using materialization to support intricate object matching. Indeed, if there are relatively few matches, then it is possible that the class $match\_A_1\_\ldots\_A_n$ will waste a great deal of space on `nil` values. In the current Squirrel prototype, we allow the underlying physical implementation to optimize the internal representation of match classes.

In its current form, when specifying the match criteria for a family of corresponding classes, the ISL can support the specification of several binary match predicates. (More complex match predicates that simultaneously involve three or more classes may be useful in some applications, but these are not currently supported). Suppose that γ is the conjunction of all of the binary match predicates for classes $A_1, \ldots, A_n$. For objects a_i in A_i and a_j in A_j, we write $a_i \sim_\gamma^{A_i,A_j} a_j$ if a_i and a_j satisfy the match predicate of γ specified for the pair A_i, A_j.

There may be complex interaction between the match predicates in a specification γ. Let $\sim_\gamma$ be the reflexive, symmetric, and transitive closure of $\cup\{\sim_\gamma^{A_i,A_j} |$ a match predicate is specified for $A_i, A_j\}$. In the integration mediators generated by the current Squirrel prototype, if a conflict is found then a warning is generated and a human must resolve the problem. (One kind of conflict arises if for a from A_i and b from A_j we have $a \sim_\gamma b$ by transitivity but $a \not\sim_\gamma^{A_i,A_j} b$. Another kind is if we have $a \sim_\gamma b$ and $a' \sim_\gamma b$ where $a \neq a'$ are from the same source class.) We also permit the integration administrator to add further rules to the rulebase so that heuristics for resolving such conflicts can be invoked automatically.

The BV execution model presented in Subsection 4.4 above must be modified to accommodate match classes. Speaking intuitively, there are two reasons for this. The first stems from the fact that a match class can have more than one child from the source databases. (No other kind of node in VDPs has this property.) This complicates the initialization step of the execution model, because the incremental updates of all children of a match node must be brought up to the match node. These incremental updates should not be applied to the match class, because one of the guiding philosophies of the BV execution model is that incremental updates to a class are applied only when that class is "processed". However, when bringing incremental updates from one source class into the match class we need to refer to the

```
interface match_Stud_Emp {
    string      studName;                    string      empName;
    integer[7]  studID;                      integer[9]  SSN;
    string      major;                       string      divName;
    string      local_address;               string      address;
    string      perm_address;                              };
```

Figure 9: The class interface of `match_Stud_Emp`, used by `S_E_Mediator`

effect of incremental updates already brought from other source classes. As a result, we use the Heraclitus operator **when** to obtain efficient hypothetical access to a match class and the incremental updates already propagated to it. The second reason that the BV execution model needs to be modified stems from the possibility of complex interaction between the possibly many binary match predicates that contribute to the definition of an n-ary match. Speaking intuitively, rules corresponding to these binary match predicates must be fired repeatedly until a fixpoint is obtained.

More formally, if match classes are present, then the notion of *VDP-rulebase* is modified to be a triple $(\mathcal{V}, edge\_rule, match\_rule)$ where $\mathcal{V}$ and *edge_rule* are as before, and *match_rule* is a mapping from each match node to a set of rules. The BV execution model is modified to have the following initialization step:

(1') Initialization': Let Δ correspond to the smash of all incremental updates held in the queue at a given time t. As before, Δ holds a family $\Delta R_1, \ldots, \Delta R_k$ of subdeltas, where $R_1, \ldots, R_k$ are classes associated with leaf nodes $v_1, \ldots, v_k$ (respectively) of $\mathcal{V}$.

During this phase, three things occur:

(1'a) All eligible rules in $\cup\{edge\_rule(v_i) \mid i \in [1,k]\}$ that do not involve edges from match classes are fired, in any order.

(1'b) For each match class $M = match\_A_1\_\ldots\_A_n$ nondeterministically choose a permutation $i_1, \ldots, i_n$ of $1, \ldots, n$. Recall that at this point ΔM is empty.

(1'b1) For each j from 1 to n do the following: Hypothetically apply rules in $edge\_rule($ $match\_A_1\_\ldots\_A_n, A_{i_j})$ against M **when** Δ_M to obtain ΔM_j. (That is, compute the net effect ΔM_j on M **when** Δ_M that applying the rules of $edge\_rule(match\_A_1\_\ldots\_A_n, A_{i_j})$ on ΔA_{i_j} would have.) Then replace ΔM by ΔM *smash* ΔM_j.

(1'b2) Hypothetically apply rules in $match\_rule($ $match\_A_1\_\ldots\_A_n)$ to M **when** ΔM and update ΔM accordingly, until a fixpoint is reached.

(1'c) All entries in the queue that contributed to Δ are deleted.

It is straightforward to verify that when using rules generated by the templates of Subsection 4.6, executions of this modified BV execution model satisfy the properties listed in Subsection 4.4.

4.6 Default VDP and rule templates

VDPs provide very flexible representations of the skeletons of update processing strategies in the mediator, which can be tailored to optimize the support of integrated views. The current Squirrel prototype constructs a reasonably efficient default VDP for a given integrated view. However, the user who created the ISL specification can explicitly modify the default VDP, so that the final integration mediator will be generated according to the revised VDP. (For example, the VDPs of Figures 6, 7(b) and 8 are default VDPs, and the VDP of 7(a) is not.)

In this and the next subsection we give a brief overview of the construction of default VDPs for supporting arbitrary ISL specifications. This subsection focuses on constructing "simple VDPs", that support individual class definitions of an ISL specification. The next subsection shows how simple VDPs can be combined to form VDPs that support arbitrary integrated views. As mentioned at the beginning of Subsection 4.3, in this discussion we focus exclusively on constructing VDPs whose imported and exported classes are sets.

A *simple VDP* is a VDP that (i) has a single root, that is distinguished; (ii) has only source classes or distinguished nodes as leaves; and (iii) has no distinguished non-leaf, non-root nodes. Intuitively, a simple VDP can be constructed for each distinguished class defined in an ISL specification. Indeed, in this subsection we describe how we build default simple VDPs for each distinguished class in an ISL specification. Although simple VDPs might have distinguished nodes as leaves, these will be replaced through "Macro-expansion" (see the next subsection) when forming the complete VDP for an ISL specification.

Every edge (a, b) in a VDP is associated with an update propagation rule which computes an incremental update ($\Delta class(a)$) to the class $class(a)$ based on an update $\Delta class(b)$. (More than one rule is generated for each edge from a match class, and additional rules are

associated with the match classes.) A family of *rule templates* is used to generate the update propagation rules. Translation of the templates into actual rules uses information about the source database classes, the classes of the integration mediator, and possibly user-defined functions.

We now describe an algorithm for building a simple VDP for an individual class definition in an ISL specification. The algorithm uses an induction on the structure of the expression. We define a subexpression to be *special* if it is a source class, a distinguished class, or if its root operator is union, difference, or match.

There are two base cases in this construction, when the subexpression is simply a source class or simply a distinguished class. In both cases, the VDP for the subexpression has one node, labeled by that class.

There are five inductive cases in the construction; we now consider these in turn. Importantly, the deltas created by the rules presented below do not include any redundant inserts or deletes.

(1) Union: Suppose that the subexpression is $T = R_1 \cup \ldots \cup R_n$. Let $\mathcal{V}_i$ be the VDP for R_i ($i \in [1..n]$). The VDP for T is constructed as the union of $\mathcal{V}_1, \ldots, \mathcal{V}_n$, along with an additional node labeled by T, and edges (T, R_i) ($i \in [1..n]$).

The rule template for the edge (T, R_i) is now given. Because T is a bag node, the deltas here are interpreted in the bag semantics.

rule template for union: edge (T, T_i)
ON new ΔR_i
IF true
THEN $(\Delta T)^+ = (\Delta R_i)^+; (\Delta T)^- = (\Delta R_i)^-;$

(2) Difference: Suppose that the subexpression is $T = R_1 - R_2$. Let $\mathcal{V}_i$ be the VDP for R_i ($i = 1, 2$). The VDP for T is constructed as the union of $\mathcal{V}_1$, $\mathcal{V}_2$, along with an additional node labeled by T, and edges (T, R_1) and (T, R_2).

The rule templates for edges (T, R_1) and (T, R_2) are now presented. Recall that T is a set node. In these templates, $\Delta' R_i$ denotes the net change (as a set-based delta) between R_i, considered as a set, and $apply(R_i, \Delta R_i)$, considered as a set. Also, the operands for $-$ and $\cap$ are interpreted as sets.

rule template for diff1: edge (T, R_1)
ON new $\Delta' R_1$
IF true
THEN $(\Delta T)^+ = (\Delta' R_1)^+ - R_2;$
$(\Delta T)^- = (\Delta' R_1)^- \cap R_2;$

rule template for diff2: edge (T, R_2)
ON new $\Delta' R_2$
IF true
THEN $(\Delta T)^+ = (\Delta' R_2)^- \cap R_1;$
$(\Delta T)^- = (\Delta' R_2)^+ \cap R_1;$

(3) SP: Suppose now that the subexpression is T, which is non-trivial and does not have union or difference as root. Suppose further that $T = \epsilon(R)$, where ϵ is an operator involving one or more selections and projections, and R is a special node. (The case where join is involved is considered shortly.) Generalizing a well-known normalization result (e.g., see [AHV95]), the expression $\epsilon(R)$ can be normalized into the form: $T = \pi_p \sigma_f R$, where p is a subset of attributes of the class R, and f is a selection predicate.

In this case, let $\mathcal{V}_1$ be the VDP for R. The VDP for T is constructed as the union of $\mathcal{V}_1$ along with an additional node labeled by T, and an edge (T, R). (An example of this case is illustrated in Figure 6 of Example 4.1, which consists of nodes R and R'.)

The rule template for the edge (T, R) is now presented (using the bag semantics).

rule template for SP: edge (T, R)
ON new ΔR
IF true
THEN $(\Delta T)^+ = \pi_p \sigma_f (\Delta R)^+;$
$(\Delta T)^- = \pi_p \sigma_f (\Delta R)^-;$

(4) SPJ: Suppose now that the subexpression is T, which is non-trivial and does not have union or difference as root. Suppose further that $T = \epsilon(R_1, \ldots, R_n)$ where ϵ is an operator involving at least one join, along with zero or more selections and projections, and each R_i is a special node. Again generalizing a well-known normalization result, the expression $\epsilon(R_1, \ldots, R_n)$ can be normalized into the form: $T = \pi_p \sigma_f (R_1 \bowtie_{g_1} \ldots \bowtie_{g_{n-1}} R_n)$, where p is a subset of attributes of the join, f is a selection predicate, each of $g_1, \ldots, g_{n-1}$ is a join condition, and the R_i's are not necessarily distinct.

In order to construct a default VDP for T we introduce n *pre-classes*, one each for $R_1, \ldots, R_n$. Intuitively, the pre-class R_i' for R_i will be a selection and projection of R_i, that includes all attributes needed for T, and includes only those tuples of R_i that will impact the join. More precisely, when constructing the pre-class R_i' for R_i we incorporate the set p_i of all attributes of R_i that are referred to in the join conditions $g_1, \ldots, g_{n-1}$, the selection condition f, or the projection list p. Also for each i we let f_i be a selection condition implied by f relevant to R_i. (We do not insist that f_i captures all of the restrictions that f makes on R_i; if f is complex, that might be inconvenient to compute.) The i-th pre-class is

now defined as: $R'_i = \pi_{p_i}\sigma_{f_i}R_i$. Finally, since some of the arguments to the join may be projections of the original arguments, we may need to modify the join conditions $g_1, \ldots, g_{n-1}$ into corresponding conditions $g'_1, \ldots, g'_{n-1}$. (The VDP $\mathcal{V}_2$ in Figure 7(b) in Example 4.2 illustrates this construction.)

The VDP for T is now constructed from the VDPs for the R'_i's (constructed as in the previous case), along with the node T and edges (T, R'_i) for $i \in [1..n]$. The rule template for the edge (T, R'_i) is (using the bag semantics):

rule template for SPJ: edge (T, R'_i)

ON	new $\Delta R'_i$
IF	true
THEN	$(\Delta T)^+ = \pi_p\sigma_f(R'_1 \bowtie_{g'_1} \ldots \bowtie_{g'_{i-1}} (\Delta R'_i)^+ \bowtie_{g'_i} \ldots \bowtie_{g'_{n-1}} R'_n)$; $(\Delta T)^- = \pi_p\sigma_f(R'_1 \bowtie_{g'_1} \ldots \bowtie_{g'_{i-1}} (\Delta R'_i)^- \bowtie_{g'_i} \ldots \bowtie_{g'_{n-1}} R'_n)$;

(5) Match: A default VDP for an n-ary match involving source classes $A_1, \ldots, A_n$ consists of a node for $match\_A_1\_\ldots\_A_n$ with an edge from $match\_A_1\_\ldots\_A_n$ to each of the A_i's. (Figure 8 illustrates a binary matching.)

We present here the rule templates for supporting a match node that concern creation of objects for a source class A_i. The two rule templates presented here would be used to generate the rules `R1` and `R2` informally described in Section 3. The modification updates indicated in the second rule action is shorthand for a deletion followed by an insertion. Although the rules generated from the templates described here refer to individual objects, the execution model apply the rules in a set-at-a-time fashion.

rule template for match_edge insertion:

	edge $(match\_A_1\_\ldots\_A_n, A_i)$
ON	new ΔA_i
IF	(insert $A_i(x : a_1, \ldots, a_n)$) in ΔA_i
THEN	[insert $match\_A_1\_\ldots\_A_n(new : \ldots, nil, x.m_1, \ldots, x.m_j, nil, \ldots)$];

where $m_1, \ldots, m_j$ are attributes contributed to $match\_A_1\_\ldots\_A_n$ by A_i.

Description: if a new object x of class A_i is inserted, insert a corresponding new object into class $match\_A_1\_\ldots\_A_n$, with *nil* for non-A_i attributes.

rule template for match_node insertion:

	node $(match\_A_1\_\ldots\_A_n)$
ON	insert $match\_A_1\_\ldots\_A_n(x : \ldots, nil, x.m_1, \ldots, x.m_j, nil, \ldots)$
IF	exists a unique y in $match\_A_1\_\ldots\_A_n$ such that $match(x, y)$
THEN	[delete $match\_A_1 \ldots\_A_n(x)$; modify $match\_A_1\_\ldots\_A_n$ $(y : existing\ attr.\ of\ y, x.m_1, \ldots, x.m_j, \ldots)$];

Description: when a new $match\_A_1\_\ldots\_A_n$ object x is inserted, if an object y of class $match\_A_1\_\ldots\_A_n$ matches x, delete x and modify y by setting the values of attributes $m_1, \ldots, m_j$ to $x.m_1, \ldots, x.m_j$.

Analogous rule templates for deletions of source objects are also included. The match_edge deletion rule template is a bit more complex than for insertion. Recall that a surrogate object in $match\_A_1\_\ldots\_A_n$ may correspond to one or more *source objects*. If a source object a_i with surrogate object x is deleted, then the match_edge deletion rule will delete x from $match\_A_1\_\ldots\_A_n$, and then insert new surrogate objects into $match\_A_1\_\ldots\_A_n$, one corresponding to each of the source objects (other than a_i) that x corresponded to. (Note that the match_node insertion rules may now recombine some of these newly inserted surrogate objects.) In this manner, information inferred from the presence of a_i will not be retained in $match\_A_1\_\ldots\_A_n$ after a_i has been deleted from the source database.

4.7 Support for VDPs of multiple export classes and VDP evolution

The previous subsection introduced default VDPs for the definitions of individual distinguished classes in an ISL specification. This subsection discusses the issue of constructing VDPs for a full ISL specification, i.e., for multiple distinguished classes. Our approach is to first create a VDP for each distinguished class individually, and then combine these into single VDP. We also briefly discuss the related issue of supporting the evolution of VDPs in response to changes of the definition to the integrated view.

Merging two VDPs means identifying and merging pairs of sharable nodes between the VDPs. The problem of merging VDPs is closely related to the problem of detecting common subexpressions in queries. For the general case, it is known that the equivalence of two subexpressions involving negation is undecidable [AHV95]. If the expressions involve only selection, projection, and equi-join, the problem is decidable but still NP-complete [LMS95].

For the present, we only merge VDP nodes that are based on a single class, i.e., leaf nodes, distinguished nodes, and nodes that correspond to the selection and/or projection of leaf or distinguished nodes. This follows the spirit of [Jar85], which uses essentially the same technique for finding common subexpressions of multiple relational queries. We now give our algorithm for merging a pair $\mathcal{V}_1$ and $\mathcal{V}_2$ of VDPs in more detail; this is used repeatedly to merge multiple VDPs:

(1) "Macro-expansion" of distinguished nodes: If $\mathcal{V}_1$ has a distinguished node n as leaf and $\mathcal{V}_2$ has a distinguished node n' as root where n and n' represent the same class, then merge n and n'.

(2) Merging source nodes or distinguished nodes: If n is a source (distinguished) node in $\mathcal{V}_1$ and there exists a source (distinguished) node n' in $\mathcal{V}_2$ that corresponds to the same class as n, drop n (drop n and all nodes below n that are not used in the definition of other nodes of $\mathcal{V}_1$ that are incomparable to n) and change all the in-edges of n to in-edges of n'

(3) Merging selection/projection nodes: If n is a selection/projection node n in $\mathcal{V}_1$, and if there exists a selection/projection n' is a selection/projection node in $\mathcal{V}_2$ that shares the same single child (source or distinguished node) as n, then replace n and n' with a new node *new*. Let $class(n) = \pi_p \sigma_f R$ and $class(n') = \pi_{p'} \sigma_{f'} R$. The class corresponding to *new* is: $class(new) = \pi_{p \cup p'} \sigma_{f \vee f'} R$. All edges to/from n and n' are changed to *new*. The definitions for affected nodes may need to be modified accordingly.

Finally, we briefly discuss the impact of changes to the distinguished nodes and/or conditions of an ISL specification on the corresponding VDP. There are two ways to deal with such changes. The straightforward way is to regenerate a Squirrel integration mediator and repopulate its local store based on the new ISL-specification. However, if the changes are limited, it might be more efficient to "adapt" the integrated view in the local store of the mediator. This would allow the use of existing data in the old VDP as much as possible, and reduce the amount of polling of the source databases. The first step of this approach is to find correspondences between nodes in the old and new VDPs. Again, we only consider correspondences between the types of nodes that were considered in VDP merging. If there is a correspondence between a node n in the old VDP and a node n' in the new VDP, the data of $class(n')$ can be derived (perhaps partially) from the data of $class(n)$ using techniques developed in [GMR95], that "adapt" a view in response to changes to the view definition. Those techniques primarily use existing data in the materialized view with minimal access to the source classes.

5 A Taxonomy of the Solution Space for Data Integration

In its current form, the Squirrel system can be used to generate a rather narrow class of mediators, that assume the underlying data is stored in a relational or restricted object-oriented form, that are based exclusively on the materialized approach, etc. In this section we provide a survey of additional possibilities for supporting read-only integrated views, that covers both different aspects of the underlying application environment, and different approaches to supporting the view. While the survey does include the virtual as well as the materialized approach, more emphasis is placed on the materialized approach. The survey is presented in the form of a taxonomy, which is summarized in Table 1 at the end of this section.

Our taxonomy is based on seven spectra. The first four spectra are relevant to all solutions for data integration; these are (1) Data model heterogeneity, (2) Expressiveness of the integration language, (3) Object matching criteria, and (4) Materialized vs. virtual. The other three spectra are relevant to solutions that involve materialization; these are (5) Activeness of the source databases, (6) Maintenance strategies, and (7) Maintenance timing. We feel that these spectra cover the most important design choices that must be addressed when solving a data integration problem. In the discussion below we have identified within each spectra what we believe to be the most important points, relative to the kinds of data integration problems and environments that arise in practice. While the spectra are not completely orthogonal, each is focused on a distinct aspect of the problem.

A primary motivation for developing the taxonomy is to aid in the development of modular implementations for a broad array of mediators that support integrated views. As just one example, the taxonomy suggests that the implementation strategy used for incremental update can be largely independent from the choice and implementation of maintenance timing. Such modularity facilitates the reusability and maintainability of different components of mediators. We expect to use the taxonomy when choosing future extensions of Squirrel.

We now describe each of the seven spectra in turn.

5.1 Data model heterogeneity

This spectrum concerns the kind of data model that is used by the underlying data sources. The primary possibilities include files, legacy and *ad hoc* models, the relational model, and object-oriented database models. Constructs for modeling temporal, geographic, manufacturing and other specialized kinds of information also arise. To construct an integrated view across different data models, some data restructuring will be necessary. This may also be necessary if one or more of the underlying models is different from the data model used by the integration mediator. The different data models will generally entail different access languages; integration across multiple models will thus require language translation or wrapping.

The current Squirrel prototype assumes that both the source databases and view for export are represented in the relational or (ODMG) object-oriented database model.

5.2 Expressiveness

This spectrum concerns the expressive power of the language(s) used to specify integrated views. One aspect of this spectrum concerns the expressive power in terms of conventional query languages. In terms of the relational data model, some possibilities here include the relatively simple conjunctive queries (in other words, algebra expressions built up from selection, projection and join); these extended using negation (i.e., the relational algebra), or with aggregation, or with both; and the inclusion of recursion [AHV95]. A somewhat orthogonal aspect concerns whether intricate object matching criteria are supported. Another orthogonal aspect is whether explicit constructs are provided in the language for temporal, geographical, and other specialized kinds of information.

A related aspect of the expressiveness spectrum concerns whether the integrated view can monitor conditions across multiple information sources, and if so, how expressive the language for specifying the conditions is.

Squirrel-generated mediators can support integrated views and monitor conditions expressed using a subset of ODMG's OQL that has roughly the expressive power of the relational algebra, extended with object matching capabilities.

5.3 Object Matching Criteria

In some cases the problem of identifying corresponding pairs of objects from different databases can be straightforward; in other cases this can be quite intricate or even impossible. We mention some key points from the spectrum, combinations and variations of these can also arise:

Key-based matching is the most straightforward one; it relies on the equality of keys of two objects to match them. WorldBase [WHW90] and SIMS [ACHK93] are two examples using this approach. A generalization of this is to permit keys that involve derived attributes, as in [DH84].

Lookup-table-based matching uses a lookup-table that holds pairs of immutable OIDs or keys of corresponding objects. References [WHW90] and [KAAK93] support look-up tables.

Comparison-based matching provides in addition the possibility of comparing (possibly derived) attributes of two objects, either with arithmetic and logic comparisons or user-defined functions that take the attributes as arguments and return a boolean value, such as the function `close_names()` in the rule `R2` of the Student/Employee example.

Historical-based matching can be used to supplement other matching methods. For instance, an application can specify that two already matching objects stay matched, even if they cease to satisfy the other matching conditions.

The current Squirrel prototype supports all of the kinds of matching criteria mentioned above.

As an aside, we note that in the Student/Employee example, the `Student` class and the `Employee` class refer to the same kinds of objects in the world, namely, people. In the terminology of [Cha94, CH95], two entity classes from different databases that refer to the same or overlapping domains of underlying objects are called *congruent* classes. In some cases objects from non-congruent classes may be closely related. For example, one database might hold an entity class for individual flights of an airline, while another database might hold an entity class for "routes" or "edges" (connecting one city to another) for which service is available. The current Squirrel prototype focuses exclusively on matching objects from congruent entity classes.

5.4 Materialized vs. virtual

This spectrum concerns the approach taken by an integration mediator for physically storing the data held in its integrated view. The choices include

fully materialized approach, as presented in the current paper, which materializes in the persistent store of the mediator all information relevant to the integrated view and maintenance of it;

hybrid approach that materializes only part of the relevant information; and

fully virtual approach, as presented in references [DH84, ACHK93, FRV95], which uses query pre-processing and query shipping to answer queries that are made against the integrated view.

The current Squirrel prototype focuses exclusively on the fully materialized approach. Reference [ADD+91] describes a system in which integrated views are primarily virtual, but some match information is materialized. Reference [ZGHW95] describes a different kind of hybrid, in which the integrated view is materialized, but the source databases must be polled when incorporating new updates.

5.5 Activeness of Source Databases

This spectrum concerns the active capabilities of source databases, and is relevant only if some materialization occurs. This spectrum allows for both new and legacy databases. The three most important points along this spectrum represent three levels of activeness.

#	Spectra	Range
1	Data model heterogeneity	file, legacy and *ad hoc* models, relational, object-oriented, ...
2	Expressiveness	integr. view (conj. query, neg., ...), obj. match., temporality,
3	Matching criteria	key $\leftrightarrow$ lookup-table $\leftrightarrow$ comparison $\leftrightarrow$ historical ...
4	Materialized vs. virtual	fully materialized $\leftrightarrow$ hybrid $\leftrightarrow$ fully virtual
5	Activeness of source DB	sufficient activ. $\leftrightarrow$ restricted activ. $\leftrightarrow$ no activ.
6	Maintenance strategies	local incr. update $\leftrightarrow$ polling-based incr. update $\leftrightarrow$ refresh
7	Maintenance timing	trans. commit, net change, network reconnect, periodic, ...

Table 1: Solution space of the data integration problem

Sufficient activeness: A source database has this property if it is able to send deltas corresponding to the net effect of all updates since the previous transmission, with triggering based either on physical events or state changes.

Restricted activeness: A source database has this property if it cannot send deltas, but it has triggering based on some physical events (e.g., method executions or transaction commits), and the ability to send (possibly very simple) messages to the integration mediator. One useful case of restricted activeness is provided by "asynchronous replication servers". These systems, that are becoming commercially available for relational DBMSs [Sta94], permit one database to hold an exact copy (no selections or projections) of a relation in another database. Another useful possibility here is the case that on a physical event the source database can execute a query and send the results to the integration mediator. Even if the source database can send only more limited messages, such as method calls (with their parameters) that were executed, the mediator may be still able to interpret this information (assuming that encapsulation can be violated).

No activeness: This is the case where a source database has no triggering capabilities. In this case the mediator can periodically poll the source databases and perform partial or complete refreshes of the replicated information.

The current Squirrel prototype is focused on the case of sufficient activeness. It would be relatively straightforward to extend Squirrel to make use of asynchronous replication servers within the restricted activeness case.

5.6 Maintenance Strategies

Maintenance strategies are meaningful only if some materialization occurs in the mediator. We consider three alternative maintenance strategies:

local incremental update approach, as presented in the Student/Employee example in Section 3, that stores relevant portions of source data in the mediator so that the incremental maintenance can be performed locally after the source notifies the mediator of relevant updates,

polling-based incremental update approach, as presented in [ZGHW95], that does not store extra data for the purpose of incremental maintenance, but polls for data as needed from the sources, and

refresh of the out-of-date classes in the mediator by regenerating all their objects.

The current Squirrel prototype is focused on local incremental update.

5.7 Maintenance Timing

Maintenance timing concerns when the maintenance process is initiated. Many different kinds of events can be used to trigger the maintenance. Some typical kinds of events include: (i) a transaction commits in a source database, (ii) a query is posed against out-of-date objects in the mediator, (iii) the net change to a source database exceeds a certain threshold, for instance, 5% of the source data, (iv) the mediator explicitly requests update propagation, (v) the computer holding the mediator is reconnected via a network to the source databases, and (vi) a fixed period of time has passed.

The current Squirrel prototype is focused on the first case mentioned above. However, the execution model used by Squirrel is quite independent of the maintenance timing, so other points on this spectrum would be relatively easy to incorporate.

6 Conclusions and Current Status

This paper presents a framework and prototype tool for generating integration mediators, that use materialization to support integrated views over multiple data sources. The paper makes several contributions towards

database interoperation. To provide context for research in this area, we (a) present a broad taxonomy that surveys much of the solution space for supporting and maintaining integrated views. At a more concrete level, we (b) introduce "integration mediators"; these are a special class of active modules that support incremental maintenance of materialized integrated views in a relatively declarative fashion. Furthermore, (c) we develop a uniform approach for generating integration mediators based on the use of "View Decomposition Plans", and describe (d) the prototype Squirrel system, which can generate integration mediators automatically. In Squirrel, (e) integration mediators are specified using a high-level Integration Specification Language (ISL). Finally, (f) our framework provides substantial support for intricate object matching criteria.

The research presented here will provide the starting point for several investigations. One direction is to use Squirrel in studies comparing the performance of the materialized and virtual approaches. These will be compared according to a variety of criteria, including query response time, average times of CPU processing and disk access, and average network traffic. We are also extending the framework of View Decomposition Plans to incorporate a hybrid of the virtual and materialized approaches [ZHK95b]. In terms of the data models supported by Squirrel, we hope to extend the framework to support more complex structures as found in some object-oriented database models (e.g., nested sets, lists, etc.). We also plan to incorporate mechanisms for integrating data that involves related but "non-congruent" classes, in the spirit of [Cha94, CH95].

Acknowledgements

We are grateful to Omar Boucelma, Ti-Pin Chang, Jim Dalrymple, Mike Doherty, and Jean-Claude Franchitti for numerous interesting discussions on topics related to this research.

References

[Abi88] Serge Abiteboul. Updates, A new frontier. In *Proc. of Intl. Conf. on Database Theory*, 1988.

[ACHK93] Y. Arens, C.Y. Chee, C.N. Hsu, and C.A. Knoblock. Retrieving and integrating data from multiple information sources. *Intl. Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[ADD+91] R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, and M. C. Shan. Pegasus heterogeneous multidatabase system. *IEEE Computer*, December 1991.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, MA, 1995.

[BDD+95] O. Boucelma, J. Dalrymple, M. Doherty, J. C. Franchitti, R. Hull, R. King, and G. Zhou. Incorporating Active and Multi-database-state Services into an OSA-Compliant Interoperability Framework. In *The Collected Arcadia Papers, Second Edition.* University of California, Irvine, May 1995.

[Bee89] C. Beeri. Formal models for object oriented databases. In *Proc. of First Intl. Conf. on Deductive and Object-Oriented Databases*, 1989.

[BLN86] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[BLT86] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 61–71, 1986.

[BM81] P. Buneman and A. Motro. Constructing superviews. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 56–64, 1981.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[Cat93] R.G.G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[CH95] T.-P. Chang and R. Hull. Using witness generators to support bi-directional update between object-based databases. In *Proc. ACM Symp. on Principles of Database Systems*, pages 196–207, 1995.

[Cha94] T.-P. Chang. *On Incremental Update Propagation Between Object-Based Databases.* PhD thesis, University of Southern California, Los Angeles, CA, 1994.

[Coh86] D. Cohen. Programming by specification and annotation. In *Proc. of AAAI*, 1986.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, 1990.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[CZN95] M. Cherniack, S.B. Zdonik, and M.H. Nodine. To form a more perfet union (intersection, difference). In *Proc. of Intl. Workshop on Database Programming Languages*, 1995. to appear.

[Dal95] J. Dalrymple. *Extending Rule Mechanisms for the Construction of Interoperable Systems.* PhD thesis, University of Colorado, Boulder, 1995.

[DH84] U. Dayal and H.Y. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Trans. on Software Engineering*, SE-10(6):628–644, 1984.

[DHDD95] M. Doherty, R. Hull, M. Derr, and J. Durand. On detecting conflict between proposed updates. In *Proc. of Intl. Workshop on Database Programming Languages*, September 1995. to appear.

[DHR95] M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases, 1995. Technical report in preparation.

[EK91] F. Eliassen and R. Karlsen. Interoperability and object identity. *SIGMOD Record*, 20(4):25–29, 1991.

[FRV95] D. Florescu, L. Raschid, and P. Valduriez. Using heterogeneous equivalences for query rewriting in multidatabase systems. In *Proc. of Third Intl. Conf. on Cooperative Information Systems (CoopIS-95)*, Vienna, Austria, May 1995.

[FWW+] T. Finin, J. Weber, G. Wiederhold, et al. DRAFT Specification of the KQML Agent-Communiction Language. June 15, 1993.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs, et al. On implementing a language for specifying active database execution models. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 441–454, 1993.

[GHJ94] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus[Alg,C]: Elevating deltas to be first-class citizens in a database programming language. Technical Report USC-CS-94-581, Computer Science Department, Univ. of Southern California, 1994. revised August, 1995.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 328–339, 1995.

[GLT94] T. Griffin, L. Libkin, and H. Trickey. A correction to "Incremental recomputation of active relational expressions" by Qian and Wiederhold. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1994.

[GMR95] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting materialized views after redefinition. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 211–222, San Jose, CA, May 1995.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 157–166, 1993.

[HJ91] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 455–468, 1991.

[HK89] S.E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Trans. on Database Systems*, 14(3):291–321, 1989.

[HW92] E. Hanson and J. Widom. An overview of production rules in database systems. Technical Report RJ 9023 (80483), IBM Almaden Research Center, October 12, 1992.

[IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, Boston, Massachussetts, 1993.

[Jar85] M. Jarke. Common subexpression isolation in multiple query optimization. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, pages 191–205. Springer-Verlag, 1985.

[KAAK93] W. Kent, R. Ahmed, J. Albert, and M. Ketabchi. Object identification in multidatabase systems. In D. Hsiao, E. Neuhold,

and R. Sacks-Davis, editors, *Interoperable Database Systems (DS-5) (A-25)*. Elsevier Science Publishers B. V. (North-Holland), 1993.

[LMR90] W. Litwin, L. Mark, and N. Roussopolos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[LMS95] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views. In *Proc. ACM Symp. on Principles of Database Systems*, pages 95–104, 1995.

[Mac88] R. MacGregor. A deductive pattern matcher. In *Proc. AAAI-88, The Natl. Conf. on Artif. Intell., St. Paul, MN*, 1988.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[SBG+81] J. M. Smith, P. A. Bernstein, N. Goodman, U. Dayal, T. Landers, K.W.T. Lin, and E. Wong. Multibase – Integrating heterogenous distributed database systems. In *National Computer Conference*, pages 487–499, 1981.

[Sta94] D. Stacey. Replication: DB2, Oracle, or Sybase? *Database Programming and Design*, December 1994.

[T+90] G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22(3):237–266, September 1990.

[Ull82] Jeffrey D. Ullman. *Principles of Database Systems (2nd edition)*. Computer Science Press, Potomac, Maryland, 1982.

[WC95] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, Inc., San Francisco, California, 1995.

[WHW89] S. Widjojo, R. Hull, and D. Wile. Distributed Information Sharing using WorldBase. *IEEE Office Knowledge Engineering*, 3(2):17–26, August 1989.

[WHW90] S. Widjojo, R. Hull, and D. S. Wile. A specificational approach to merging persistent object bases. In Al Dearle, Gail Shaw, and Stanley Zdonik, editors, *Implementing Persistent Object Bases*. Morgan Kaufmann, December 1990.

[WWH90] Surjatini Widjojo, Dave Wile, and Richard Hull. WorldBase: A new approach to sharing distributed information. Technical report, USC/Information Sciences Institute, February 1990.

[ZGH94] G. Zhou, S. Ghandeharizadeh, and R. Hull. Benchmarking the Heraclitus[Alg,C] prototype. Technical Report USC-CS-94-582, University of Southern California, August 15 1994. Available in file HERACLITUS/TR-582 at ftp@perspolis.usc.edu.

[ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 316–327, San Jose, California, May 1995.

[ZHK95a] G. Zhou, R. Hull, and R. King. Generating data integration mediators that use materialization, 1995. To appear, *Journal of Intelligent Information Systems*; Available via anonymous ftp at ftp://ftp.cs.colorado.edu/users/hull/squirrel:materialiJIIS.ps.

[ZHK95b] G. Zhou, R. Hull, and R. King. Generating data integration mediators that use materialization. Technical report, Computer Science Department, University of Colorado, October 1995.

To: Gang Zhou <gzhou@lheureux.cs.colorado.edu>
From: libhart@cs.colorado.edu (Pat Libhart)
Subject: Re: new tech. report
Cc:
Bcc:
X-Attachments:

>Hi Pat,
>
>Could you please assign us a # for tech report?
>
>Title: Squirrel Phase 1: Generating Data Integration Mediators that
> Use Materialization
>
>Authors: Gang Zhou, Richard Hull, and Roger King
>
>Abstract:
>
>This paper presents a framework for data integration
>that is based on using "Squirrel integration mediators"
>that use materialization to support integrated views over multiple databases.
>These mediators generalize techniques from
>active databases to provide incremental
>propagation of updates to the materialized views.
>A framework based on "View Decomposition Plans"
>for optimizing the support of materialized
>integrated views is introduced.
>The paper describes the
>Squirrel prototype currently under development,
>which can generate Squirrel
>mediators based on high-level specifications.
>
>The integration of information by
>Squirrel-generated mediators is
>expressed primarily through an extended version
>of a standard query language, that can refer to data from
>multiple information sources.
>In addition to materializing an integrated
>view of data, these mediators can monitor
>conditions that span multiple sources.
>The Squirrel framework also provides efficient support
>for the problem of
>"object matching", that is, determining when object
>representations (e.g., OIDs)
>in different databases correspond to the same object-in-the-world,
>even if a universal key is not available.
>
>To establish a context for the research,

>the paper presents a taxonomy that surveys
>a broad variety of approaches
>to supporting and maintaining integrated views.
>
>
>
>Thanks
>
>Gang

Gang:
Please use TR #CU-CS-793-95 for your tech report. Please give me a hard copy as soon as you can.
Pat