Implementing a Universal Relation Interface Using Access Scripts with Binding Patterns

Christine Reck
Universität Karlsruhe
reck@ira.uka.de

Gerd Hillebrand*
Universität Karlsruhe
ggh@ira.uka.de

Abstract

We propose the use of the universal relation as a user interface to provide transparent access to a network of distributed, heterogeneous, and autonomous information sources. We implement this interface in two layers. The lower layer consists of *access scripts*, which encapsulate knowledge about information sources and are capable of answering basic queries. The upper layer uses combinations of these scripts to answer user queries phrased in terms of a universal relation. Access scripts know how to obtain information either directly from sources or from service providers (mediators, traders, and the like). They present this information in relational form, but with an inherent direction, in the sense that whenever values for a fixed subset of attributes of the relation are given, the access script will deliver values for the rest of the attributes in the relation. In this paper, we address the problem of defining the semantics of a user query posed against the universal relation and of finding a sequence of access script invocations that gathers the information requested in the query.

1 Introduction

Motivation. In the past few years the number of services being available on the net has grown rapidly. So far the user has to "surf" the net in order to get the information he needs. That is, he needs to find the adequate information sources or service providers on his own, perhaps assisted by search tools, and he needs to combine information from different sources without any further support. In our approach the user is supplied with a relatively simple model of the net, which he can use to formulate queries. The model offers complete transparency, that is, the user need not be aware of the existing information sources or service providers and their locations. Usually a number of service providers and information sources are involved in answering a user query. It is the task of the system to choose these service providers and information sources and to coordinate their execution via a query evaluation plan.

Approach. We assume that we have a network of information providers of various kinds: primary sources (connected to the network through wrappers), mediators, traders, providers of value-added services, etc., see Fig. 1. Together, they provide a set of services that we call the *services interface*. To make use of the services interface a number of *access scripts* are defined. An access script is a program that implements a basic interaction with the services interface that is deemed to be needed rather often. Examples include: accessing a trader to find a provider of a particular service, obtaining offers for goods or services, accepting an offer, and the like. An access script exports an interface that is a set of *attributes* drawn from some agreed-upon universe of attributes (there may be several distinct attribute universes corresponding to unrelated application domains). Some of the attributes in the interface are designated as *inputs*; the others are *outputs* of the script. Given values for the input attributes, the access script will produce (possibly many) values for the output attributes. Each access script therefore defines a relation over its input and output attributes, but a relation that can only be accessed in a particular way. The collection of available access scripts may be viewed together as constituents of a universal relation whose schema is the attribute universe. The user interacts with the system only through this universal relation. Given a user query referencing certain attributes in combination, it is the task of the system to find the proper connection between these attributes and to choose a sequence of access scripts implementing that connection. This task is complicated by the fact that input and output attributes of access scripts must be

*Corresponding author. Address: Institut fuer Programmstrukturen und Datenorganisation, Universitaet Karlsruhe, Am Fasanengarten 5, D-76131 Karlsruhe, Germany. Phone: +49 (721) 608-2080. Fax: +49 (721) 694092.

respected, that is, it is not possible to provide values for the outputs and deduce the corresponding inputs. Moreover, in our setting the universal relation represents a much looser aggregation of information than in traditional universal relation systems. In particular, no dependencies are available on the schema, and the semantics of attributes is typically less strictly defined. Therefore, classical work on the universal relation interface is only of limited help. In this paper, we present a formal model of a universal relation interface to a set of "directed" relations and describe possible ways of implementing it.

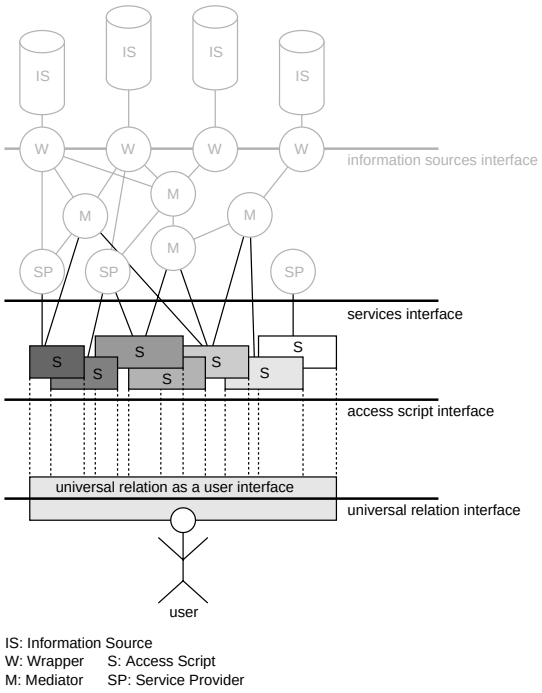

Figure 1: Interfaces to a Network of Information Sources

Example. Consider a travel information system. We assume that there are information sources on various means of transport (air lines, trains, car rental companies, etc.), traders that know the location of these sources, and mediators that perform semantic integration. We moreover assume that we have access scripts to implement the following functions:

- Script S_1: Given a service category, ask a trader for mediators that understand requests in that category (type $\mathit{Category} \rightarrow \mathit{Mediator}$).
- Script S_2: Given a mediator and a service category, determine the list of parameters needed in order to obtain a bid (type $\mathit{Mediator} \times \mathit{Category} \rightarrow \mathit{Parameters}$).
- Script S_3: Given a list of parameters, obtain values for them from the user (type $\mathit{Parameters} \rightarrow \mathit{Values}$).
- Script S_4: Given a mediator, a service category, and a list of values for the parameters of that service category, obtain bids (type $\mathit{Mediator} \times \mathit{Category} \times \mathit{Values} \rightarrow \mathit{Bid}$).

Suppose the user wants to rent a car. He selects the service category "car-rental" and asks for bids b. The system searches for a combination of access scripts that, given values for the *Category* attribute, finds values for the *Bid* attribute and derives the execution plan (written as a conjunctive query)

$$S_1(\text{"car-rental"}, m), S_2(m, \text{"car-rental"}, p), S_3(p, v), S_4(m, \text{"car-rental"}, v, b).$$

When this plan is executed, the following happens. S_1 locates a mediator m providing the "car-rental" service (it is assumed that, through mediation, this service comes with some standard interface). S_2 obtains from m a description p of the parameters of this service. S_3 uses p to build and display a form that the user fills in, providing the specifics of his request (pickup and return dates, car class, location, etc)., which S_4 then uses to obtain bids from m.

Related work. A number of architectures allowing integration of information from heterogeneous information sources has been developed. I^3, the ARPA-sponsored Intelligent Integration of Information program [7], includes a generic system architecture designed to present an initial categorization of the principal services that can be used to support the intelligent integration of highly heterogeneous information sources. TSIMMIS, the Stanford-IBM Manager of Multiple Information Sources [3], is a system implementing parts of the I^3 architecture using a simple object exchange model [12]. The focus of the TSIMMIS project is to support the development of components facilitating the integration of information, such as mediators [6] and wrappers [13]. DIOM [9] proposes a query mediation framework to support customizable information gathering across heterogeneous and autonomous information sources. Another approach is described in the Manifesto on Cooperative Information Systems [4]. There, a generic four-layer architecture for cooperative information systems is introduced, which also includes support for human collaboration and global organizational concerns.

In contrast, the focus of our work is not on the integration of information. We assume that semantic integration is handled by mediators and that the knowledge to carry out interactions with the network is encapsulated in access scripts. Our concern is how to present the functionality provided by these access scripts in a simple model that can be understood by non-expert users. For this, we adopt the *universal relation model* where the underlying relations can be understood as *views with binding patterns*.

There has been a lot of work on the design and implementation of universal relation systems (see, e.g., [11, 10, 17, 8] for surveys of various aspects and [16] for an introduction to the topic and its history). An important question in this context is the proper choice of the universal relation instance. We follow Rajaraman and Ullman [15] in adopting the *full disjunction* [5] of the underlying relations as the "right" instance of the universal relation. While [15] is concerned with ways of computing the full disjunction by a sequence of outerjoins, we do not want to materialize the universal relation, and focus on the translation of queries instead. Here the problem is that the underlying "relations" (i.e., the access scripts) cannot be combined arbitrarily, but only in ways prescribed by their input and output attributes. Rajaraman, Sagiv, and Ullman study a similar problem in [14]. They consider a setting where certain information sources are given and binding patterns are used to specify so-called query templates that a given source can answer. Each query template defines a view of the underlying information source that has certain input and output attributes and whose semantics is specified as a conjunction (or more generally, a datalog program) over certain base predicates. Queries are similar to query templates in that they are datalog rules with a binding pattern attached to the rule head and a body consisting of base predicates. The task is to find a conjunction of views, respecting their binding patterns, such that the semantics of the query is the same as the semantics of the conjunction of views. The problem studied in this paper is somewhat different, because the semantics of a query is not prespecified, but must be derived from the attributes that it mentions. Moreover, as we shall see later on, in our setting it is usually impossible to retrieve all parts of the universal relation relevant to a particular set of attributes, so we cannot expect to have a perfect match of the semantics of a query (defined in terms of the universal relation) and the semantics of an access script sequence that implements it. We must therefore be content to look for compositions of access scripts that do not produce any incorrect answers, and as many correct answers as possible.

Contributions. In summary, the technical contributions of this paper are as follows: First, we propose the use of *access scripts* to encapsulate basic functionality in accessing and manipulating a network of information sources, and to present this functionality to the non-expert user in terms of a universal relational model. Second, we study the notion of a universal relation where the underlying relations are "directed", and we define a notion of sound interpretations of user queries. Third, we show how to obtain sound interpretations of a given user query and discuss an execution model for such interpretations.

Organization of the paper. The rest of the paper is organized as follows. In Section 2 we present a formal description of the semantics of a universal relation interface to a set of access scripts. Sections 3 and 4 are concerned with the implementation of this interface. In Section 3, we describe the translation process that converts queries posed against the universal relation into a series of calls to access scripts, and in Section 4, we discuss how such a sequence is executed. We conclude in Section 5 with a discussion of our results and future work.

2 The Model

As stated in the introduction, we want to present the user with a view of the access script layer that is just a single universal relation. The user phrases queries as combinations of one or more predicates, which express relationships

between certain attributes of the universal relation. For example, suppose there are two access scripts: one that, given the name of an institution, produces a list of ftp servers at that institution (e.g., by probing all hosts within the institution), and another one that, given an ftp server and a file name pattern, produces a list of all files available on the server whose names match the specified pattern. To the user, these access scripts would appear as a single relation over attributes I(nstitution), S(erver), P(attern), F(ile), and a query might specify predicates such as P_{IPF}("uni-karlsruhe", "*.ps", f), which asks for all PostScript files available via ftp from some server at the University of Karlsruhe. This query can be answered in a straightforward way by first determining via Access Script 1 the set of ftp servers at the University of Karlsruhe and then for each server via Access Script 2 the set of "*.ps" files that it stores. However, the similar query $P_{IPF}(i$, "*.ps", "juicy-tales.ps"), which asks for all institutions that offer a file called "juicy-tales.ps" from one of their ftp servers, cannot be answered from the available access scripts, because they cannot be operated "backwards".

In the following, we formalize the notion of a universal relation interface to a set of access scripts. In particular, we define the universal relation corresponding to a set of access scripts, the syntax and semantics of queries, and a soundness criterion for access script sequences.

Attributes. As usual in the universal relation model, we assume that there is some finite universe **Attr** of *attributes* that suffice to describe all aspects of the world under consideration.[1] Moreover, we require that attributes are used consistently by all access scripts, in the sense that occurrences of the same attribute $A \in$ **Attr** always refer to the same "role". At the very least, such occurrences should refer to the same fixed domain, which we denote by $dom(A)$. We refer to domain elements as *constants*.

Other assumptions that are commonly made in connection with the universal relation model are the *relationship uniqueness* assumption and the *one flavor* assumption (cf. [11]). The first states that for any set of attributes $\mathcal{A}$, there should be a unique (or at least distinguished) conceptual relationship between them, which is sometimes called the *window* or *connection* on $\mathcal{A}$. We feel that for the purpose of facilitating the use of the network for non-expert users, a rather "wide" window is appropriate, which may include extraneous information rather than exclude relevant information. We therefore use a *full disjunction* window, explained below.

The one flavor assumption asserts that if multiple access paths are available to compute the window on $\mathcal{A}$, tuples from different access paths are "comparable" in the sense that they express the same relationship between their components. In our setting, this may be enforced to some degree by judicious design of access scripts and their attribute names. However, the semantics of access scripts ultimately depends on the availability and contents of the underlying information sources, which may change frequently and in ways beyond the control of the access script designer. We therefore adopt an ad-hoc approach to access path selection, where the system generates (in a way explained below) several plausible access paths, which may be examined, evaluated, and refined by the user, if he so chooses.

Access scripts. An *access script* is a combination of an *access script type* and an *extension* of that type. Access script types are triples $(S, \mathcal{I}, \mathcal{O})$, where S is a unique name and $\mathcal{I}$ and $\mathcal{O}$ are disjoint sets of attributes called the *input* and *output* attributes of S. We write access script types in the form $S{:}\mathcal{I} \rightarrow \mathcal{O}$.

An *extension* of an access script type $S{:}\mathcal{I} \rightarrow \mathcal{O}$ is a relation over $\mathcal{I} \cup \mathcal{O}$, that is, a subset of the Cartesian product $\bigotimes_{A \in \mathcal{I} \cup \mathcal{O}} dom(A)$. We do not require this relation to be finite or single-valued, because we want to allow access scripts that, for example, convert from dollars to German marks (infinite) or produce a set of tourist attractions for a given city (multivalued). The access script type restricts the way the underlying extension may be accessed: the only allowed operation is a "lookup" where values for the input attributes are given and the corresponding values for the output attributes are retrieved. In other words, the only way the access script extension may be accessed is by computing the natural join of the extension and some relation over the input attributes.

To simplify the following discussion, we assume that there is a fixed set **AS** of access script types describing the access scripts of interest and that **Attr** is the set of attributes mentioned in these types. An *instance* of **AS** is a mapping I that assigns to every access script type in **AS** an extension of that type.

Universal relation. We adopt the universal relation as a user interface (see "Related Work" above). Most universal relation models exploit dependencies in the data to define the connections between attributes. However, we believe that in our setting a fairly "loose" notion of connection, which treats every join-consistent combination of tuples as valid, is more appropriate. Therefore, we use as universal relation instance what Galindo-Legaria [5]

[1] There may be different attribute universes corresponding to unrelated contexts, in which case the user chooses a context before posing his query.

calls a *full disjunction*.[2] This is defined as follows.

Let $\mathcal{R}$ be a set of relations, and let $\mathcal{A}$ be the union of the schemas of the relations in $\mathcal{R}$. The *full disjunction* of $\mathcal{R}$, written $FD(\mathcal{R})$, is a relation over schema $\mathcal{A}$. It is obtained by taking all subsets $\mathcal{S} \subseteq \mathcal{R}$, forming for each subset $\mathcal{S}$ the natural join of its members (which may involve forming Cartesian products if the schemas are not connected), padding all tuples so obtained with null values to obtain tuples over $\mathcal{A}$, and collecting the padded tuples into $FD(\mathcal{R})$. Formally,

$$FD(\mathcal{R}) = \cup_{\mathcal{S} \subseteq \mathcal{R}} pad_{\mathcal{A}}(\bowtie \mathcal{S}),$$

where $pad_{\mathcal{A}}$ denotes the operation of padding with null values to obtain tuples over $\mathcal{A}$. If I is an instance of **AS**, we denote by $FD(I)$ the full disjunction of the extensions of the access scripts in I.

For example, suppose that **AS** is $\{S_1 : A \rightarrow B, S_2 : A \rightarrow C, S_3 : B \rightarrow C\}$ and that instance I assigns extensions $b_1 = \{\langle a, b\rangle\}$, $b_2 = \{\langle a, c\rangle\}$, and $b_3 = \{\langle b', c\rangle\}$ to S_1, S_2, and S_3, respectively. Then the full disjunction of I is

$FD(I) =$

A	B	C
a	b	
a		c
	b'	c
a	b	c
a	b'	c

where null values are represented as blanks.

It is important to observe that the full disjunction of an instance of **AS** may contain information that is not "reachable" in certain cases. In the example above, suppose the user provides the value a for attribute A and asks for corresponding values of attribute B. We can use S_1, which maps A values to B values, to obtain the value b. However, finding the value b', which is also associated with a in the full disjunction, would require going from a to c by means of S_2 and then operating S_3 "backwards" to find b', which is impossible. Thus, for certain queries it is entirely possible that, although the information is present in principle, there is no access path to compute it.

Queries. The queries we consider are conjunctive queries over a set of predicates that correspond to total projections[3] of the universal relation. To make the syntax precise, we introduce a countable set $\mathcal{V} = \{x, y, z, \ldots\}$ of variable symbols and for every attribute set $\{A_1, \ldots, A_n\} \subseteq \mathbf{Attr}$ an n-ary predicate symbol $P_{\{A_1,\ldots,A_n\}}$. A *query predicate*, then, is an expression of the form $P_{\{A_1,\ldots,A_n\}}(e_1, \ldots, e_n)$, where $e_i \in dom(A_i) \cup \mathcal{V}$ for $1 \leq i \leq n$, and a *query* is a conjunction of query predicates.

Queries are conveniently written as tableaux, e.g., the tableau corresponding to query $P_{\{AB\}}(a, x), P_{\{AC\}}(x, c)$ over attribute set $\mathbf{Attr} = \{A, B, C, D\}$ is

A	B	C	D
a	x		
x		c	

In general, the tableau corresponding to a query q is a matrix (a_{ij}) with one column for every attribute in **Attr** and one row for every predicate in q. For row i corresponding to query predicate $P_{\{A_1,\ldots,A_n\}}(e_1, \ldots, e_n)$ and column j corresponding to attribute A, the entry a_{ij} is e_k if $A = A_k$ for some $k \in \{1, \ldots, n\}$, and blank otherwise.

Given a universal relation instance u and a query q, an *answer* to q w.r.t. u is a valuation, i.e., a mapping from variables to constants, that makes all predicates in the query true. This means that for every predicate $P_{\{A_1,\ldots,A_n\}}(e_1, \ldots, e_n)$ occurring in q, the tuple $(e_1, \ldots, e_n)$ must, after replacement of variables with their assigned values, occur in the total projection of u onto $\{A_1, \ldots, A_n\}$. The set of all answers to q w.r.t. u is denoted by $[\![q]\!](u)$.

We have adopted this somewhat uncommon notion of answer (rather than saying that an answer is some set of tuples[4]), because we want to be flexible regarding the presentation of answers to queries. For example, one interface might present answers in a tableau style, where a copy of the query tableau is generated for each

[2] Actually, our definition of full disjunction is not quite the same as Galindo-Legaria's, because we allow Cartesian products in a full disjunction, and we do not minimize with respect to tuple subsumption. However, this is not essential to our model and merely a matter of technical convenience.

[3] Recall that the total projection $\Pi{\downarrow}_{\mathcal{A}}(r)$ of a relation r onto attribute set $\mathcal{A}$ is defined as the set of tuples in the ordinary projection of r onto $\mathcal{A}$ that do not contain any null values.

[4] Of course, answers in the sense of our definition can be regarded as tuples over $\mathcal{V}$, the set of variables.

answer, with variables replaced by their values. This style would be convenient for, e.g., queries requesting travel itineraries, where each resulting tableau corresponds to one possible itinerary. Another interface might support annotations in queries (e.g., in the form of a summary row or QBE-style "P." operators [18]), whereby the user can mark variables of interest to him, and in that case the result would truly be a set of tuples, constructed by projecting answers in the appropriate way. To factor out these presentation considerations from our model, we need a notion of answer that retains the maximal amount of information returned by a query.

Access script sequences. Queries are evaluated by translating them into a sequence of calls to access scripts and then executing the sequence. These sequences correspond to conjunctive queries against the available access script extensions, and their precise syntax and semantics is as follows.

We introduce for every access script type $S{:}\mathcal{I} \rightarrow \mathcal{O}$ in **AS** a predicate symbol $S_{\mathcal{I},\mathcal{O}}$ of arity $|\mathcal{I} \cup \mathcal{O}|$ and say that an *access script predicate* is an expression of the form $S_{\{I_1,\ldots,I_m\},\{O_1,\ldots,O_n\}}(e_1,\ldots,e_{m+n})$, where $I_1,\ldots,I_m$ and $O_1,\ldots,O_n$ are the input and output attributes of S, respectively, and each argument e_i is either a variable or a constant of the appropriate domain. Arguments $e_1,\ldots,e_m$ are called the *inputs* of the predicate and the remaining arguments are called the *outputs* of the predicate.

An *access script sequence* is a list of access script predicates. It is *executable* if every variable symbol that occurs as an input of some predicate also occurs as an output of an earlier predicate. This condition is clearly necessary for executing the sequence (the execution mechanism will be discussed in more detail later on), because an access script cannot be activated unless all its inputs are known, either from earlier computations or because they are constants.

Given an access script sequence s and an instance I of **AS**, an *answer* to s w.r.t. I is a valuation that makes all access script predicates in the sequence true w.r.t. I. This means that for every predicate $S_{\mathcal{I},\mathcal{O}}(e_1,\ldots,e_{m+n})$ occurring in s, the tuple $(e_1,\ldots,e_{m+n})$ must, after replacement of variables with their assigned values, occur in $I(S)$. The set of all answers to s w.r.t. I is written as $[\![s]\!](I)$.

Given a query q, selecting an access script sequence for evaluating q corresponds in some ways to selecting a window function for a query against an ordinary universal relation system. The difference is that the selection of an access script sequence depends not only on the set of attributes mentioned in the query, but also on the particular placement of constants and variables, because that affects which access scripts are applicable. We have already seen earlier that usually there is no hope of finding a single access script sequence that computes all answers to q (in fact, some answers may not be computable by any access script sequence), so the best we can do is to find access script sequences that compute no wrong answers and that do not unreasonably rule out correct answers. The following somewhat technical definition captures this intuitive notion.

Given a query q, we say that an access script sequence s is *sound* for q if it satisfies the following three conditions.

1. s is executable.
2. Suppose that q is written as a tableau (a_{ij}) as described above, and that all blank entries in (a_{ij}) are replaced by fresh variables that occur nowhere else. Then the arguments of every access script predicate $S_{\{I_1,\ldots,I_m\},\{O_1,\ldots,O_n\}}(e_1,\ldots,e_{m+n})$ in s must be projections of some row of (a_{ij}), in the sense that $(e_1,\ldots,e_{m+n})$ are the entries of that row in the columns corresponding to attributes $I_1,\ldots,I_m,O_1,\ldots,O_n$.
3. For every query predicate $P_{A_1,\ldots,A_n}(e_1,\ldots,e_n)$ in q, there exists a subsequence s' of s such that: (a) the argument tuple of every access script predicate in s' is a projection, in the sense of property (2), of the tableau row corresponding to $P_{A_1,\ldots,A_n}(e_1,\ldots,e_n)$, and (b) the union of the input and output attributes of the access script predicates in s' contains $\{A_1,\ldots,A_n\}$.

Condition (1) is clearly necessary, because we want to be able to execute the sequence. Condition (2) says that arguments of an access script predicate must be drawn from a single row of the tableau corresponding to the query. This restriction is necessary to eliminate unwanted answers, as the following example shows.

Suppose there is a single access script of type $S_1{:}\,A \rightarrow B$ with extension $\{\langle a,b\rangle, \langle a',b'\rangle\}$. Consider the query q

A	B
a	x
a'	y

and the access script sequences $S_{1\{A\},\{B\}}(a,x), S_{1\{A\},\{B\}}(a',y)$ and $S_{1\{A\},\{B\}}(a,y), S_{1\{A\},\{B\}}(a',x)$. The first is sound for q and produces the valuation $x=b, y=b'$, which is an answer to q. The second violates condition (2) and produces the valuation $x=b', y=b$, which is not an answer to q.

It should be noted, however, that condition (2) in some cases eliminates the only access script sequences that produce answers to a query at all. Suppose that in the example above, the query had been phrased as

A	B
a	x
	y

In this case it is easy to see that there is no sound access script sequence for q. However, the access script sequence $S1_{\{A\};\{B\}}(a,x), S1_{\{A\};\{B\}}(a,y)$, which violates condition (2) above, produces the answer $x = b, y = b$, which in fact is an answer to q. Nevertheless, it seems more appropriate in this case to reject the query as under-specified, rather than introducing arbitrary assumptions about the A values associated with y.

Condition (3) requires that each argument occurring in some query predicate be covered by some access script predicate in the access script sequence. Clearly, this should be true for variables occurring in the query, because they must be computed somewhere. But it also should apply to constants in the query, as the following example shows. Suppose that access script types $S_1 : A \rightarrow B$ and $S_2 : C \rightarrow B$ with extensions $\{\langle a, b\rangle, \langle a, b'\rangle\}$ and $\{\langle b', c\rangle, \langle b'', c\rangle\}$ are given. Their full disjunction u is

A	B	C
a	b	
a	b'	
	b'	c
	b''	c
a	b'	c

Assume that we want to find an access script sequence for the query

A	B	C
a	x	c

If we do not require each argument in the query to be covered by at least one access script predicate, we can produce the access script sequence $S1_{\{A\};\{B\}}(a,x)$. This sequence has two answers, namely $x = b$ and $x = b'$. However, the tuple $\langle a, b, c\rangle$ is not in u, and therefore the first valuation is not an answer to q.

If we do require each argument to be covered by at least one access script predicate, we obtain the access script sequence $S1_{\{A\};\{B\}}(a,x), S2_{\{C\};\{B\}}(c,x)$ (or $S2_{\{C\};\{B\}}(c,x), S1_{\{A\};\{B\}}(a,x)$) leading to the valuation $x = b'$, which is an answer to q.

Although condition (3) eliminates access script sequences that lead to "wrong" answers, it sometimes does not eliminate access script sequences that produce answers that are technically correct, but nevertheless unintuitive. To see this, assume that there is one more access script type $S_3 : C \rightarrow D$ with extension $\{\langle c, d\rangle\}$. Then u is given through the following relation:

A	B	C	D
a	b		
a	b'		
	b'	c	
	b''	c	
		c	d
a	b'	c	
	b'	c	d
	b''	c	d
a	b	c	d
a	b'	c	d

A sound sequence for the query given above is $S1_{\{A\};\{B\}}(a,x), S3_{\{C\};\{D\}}(c,y)$, where $y \neq x$ is a fresh variable, leading to the valuations $\{x = b, y = d\}$ and $\{x = b', y = d\}$, which are both answers to q. However, intuitively the "right" access script sequence is $S1_{\{A\};\{B\}}(a,x), S2_{\{C\};\{B\}}(c,x)$, which produces the valuation $x = b'$.

The reason for the problems arising in this example is that the user supplied values for attributes A and C, even though supplying a value for A (or C) is enough to compute values for attribute B. Hence, the query is over-specified in the sense that the user supplied more information than necessary. Condition (3) says that such extraneous information may not simply be ignored.

The following lemma shows that the designation "sound" is justified, in the sense that valuations produced by a sound access script sequence are always answers to the corresponding query.

Lemma 2.1 *Let q be a query, s be an access script sequence that is sound for q, I be an instance of $\mathbf{AS}$, and $u = FD(I)$ be the universal relation instance corresponding to I. Then $[\![s]\!](I) \subseteq [\![q]\!](u)$.*

Proof: Let σ be an answer to s w.r.t. I and $P_{A_1,\ldots,A_n}(e'_1,\ldots,e'_n)$ be a predicate in q. We need to show that σ makes this predicate true, i.e., that $(\sigma(e'_1),\ldots,\sigma(e'_n)) \in \Pi{\downarrow}_{\{A_1,\ldots,A_n\}}(u)$. Let (a_{ij}) be the tableau corresponding to q, with blanks replaced by distinct fresh variables, let $\vec{e}'$ be the row of (a_{ij}) corresponding to $P_{A_1,\ldots,A_n}(e'_1,\ldots,e'_n)$, and let $S_{1_{\mathcal{I}_1;\mathcal{O}_1}}(\vec{e}_1),\ldots,S_{k_{\mathcal{I}_k;\mathcal{O}_k}}(\vec{e}_k)$ be the subsequence of s that remains after deleting all predicates whose argument lists are not projections of $\vec{e}'$ in the sense of property (2) above. Since $\sigma \in [\![s]\!](I)$, we have $\sigma(\vec{e}_i) \in I(S_i)$ for $1 \le i \le k$. Moreover, since each tuple $\vec{e}_i$ is a projection of $\vec{e}'$, the tuples $\sigma(\vec{e}_1),\ldots,\sigma(\vec{e}_k)$ are join-consistent and therefore their join, after padding with nulls, occurs as some tuple t in u. Note that because of property (3) above, $\{A_1,\ldots,A_n\} \subseteq \bigcup_{1\le i\le k}(\mathcal{I}_i \cup \mathcal{O}_i)$ and hence t has non-null values in attributes $A_1,\ldots,A_n$. It follows that $(\sigma(e'_1),\ldots,\sigma(e'_n)) = \Pi_{\{A_1,\ldots,A_n\}}(t) \in \Pi\downarrow_{\{A_1,\ldots,A_n\}}(u)$. □

3 Generating Access Script Sequences

Given a query, the system has to produce one or more sound access script sequences for the query and execute them. It is unlikely that it will be feasible to produce all possible sound access script sequences. We expect that in practice, the system will, guided by heuristics and cost estimates, suggest some candidate sequences, which may then be refined in an interactive process, if desired. We have not studied such heuristics and cost models yet, but we believe that these will be an important part of any implementation.

There are several possible ways of generating sound access script sequences. In the following, we present a simple forward chaining algorithm.[5]

Given a query q in tableau form, say $q = (a_{ij})_{1\le i\le m, 1\le j\le n}$, where each a_{ij} is either a constant, a variable, or blank, the algorithm works as follows. First, every blank entry in (a_{ij}) is replaced by a fresh variable that occurs nowhere else. We call these newly introduced variables *anonymous* variables. Then, Boolean matrices (k_{ij}) and (t_{ij}) are initialized as follows:

$$k_{ij} \leftarrow \begin{cases} 1 & \text{if } a_{ij} \text{ is a constant} \\ 0 & \text{otherwise} \end{cases}$$

$$t_{ij} \leftarrow \begin{cases} 0 & \text{if } a_{ij} \text{ is an anonymous variable} \\ 1 & \text{otherwise} \end{cases}$$

The (k_{ij}) matrix (k for "known") flags those entries in the query tableau that may be used as inputs to subsequent access script predicates. The (t_{ij}) matrix (t for "to cover") flags entries that must still be used in some access script predicate in order to satisfy property (3) of sound access script sequences.

For each new access script predicate that it generates, the algorithm must choose an applicable access script type and a row of the query tableau where the arguments to the predicate should come from (cf. property (2) of soundness). To avoid generating the same predicate twice, we maintain a list *cand* of possible choices and delete choices from this list once they have been picked. Initially, the list contains all possible combinations of access script types and rows:

$$cand \leftarrow \{(S{:}\mathcal{I} \to \mathcal{O}, i) \mid S{:}\mathcal{I} \to \mathcal{O} \in \mathbf{AS}, 1 \le i \le m\}$$

The algorithm then executes the following loop (the double slash delimits comments):

$seq \leftarrow \emptyset$
while nonzero t_{ij}'s exist **do**
  // determine applicable choices
  $choices \leftarrow \{(S{:}\mathcal{I} \to \mathcal{O}, i) \in cand \mid k_{ij} = 1$ for all columns j corresponding to attributes in $\mathcal{I}\}$
  if $choices = \emptyset$ **then** fail **end**

[5] However, depending on the structure of the available access script types, backward chaining or other, more sophisticated search techniques may be more efficient.

choose $(S{:}\mathcal{I} \rightarrow \mathcal{O}, i) \in choices$ // heuristics or cost functions may be applied here
$cand \leftarrow cand \setminus \{(S{:}\mathcal{I} \rightarrow \mathcal{O}, i)\}$ // eliminate choice from further consideration
$\vec{e} \leftarrow \Pi_{\mathcal{I} \cup \mathcal{O}}(a_{i1}, \ldots, a_{in})$ // determine arguments of access script predicate
$seq \leftarrow append(seq, S_{\mathcal{I},\mathcal{O}}(\vec{e}))$ // append predicate to sequence
// update (k_{ij}) and (t_{ij}) matrices
$k_{ij} \leftarrow 1$ for all j corresponding to attributes in $\mathcal{O}$
$t_{ij} \leftarrow 0$ for all j corresponding to attributes in $\mathcal{I} \cup \mathcal{O}$
$k_{uv} \leftarrow 1$ for all u, v such that for some $1 \leq j \leq n$, $a_{uv} = a_{ij}$ and $k_{ij} = 1$
end while

The final value of seq is the desired access script sequence.

Lemma 3.1 *Given a query q, the algorithm above will produce an access script sequence that is sound for q if one exists.*

Proof: The algorithm always terminates because one pair (S, i) is removed from *cand* during each iteration. It is easy to see that if it terminates successfully, the resulting access script sequence seq is sound for q: property (1) is ensured by means of the (k_{ij}) matrix, property (2) because the argument list $\vec{e}$ of an access script predicate is constructed as a projection of a tableau row, and property (3) by means of the (t_{ij}) matrix.

On the other hand, suppose an access script sequence s sound for q exists. Produce a sequence s' that contains for each predicate p of s, in turn, a pair containing the access script type corresponding to p and the index of the row of q from which the arguments to p came (such a row exists because of property (2) of soundness). An induction then shows that on any run of the algorithm, the set *choices* will always contain one of the pairs in s': initially the first pair from s' will be in *choices*, and after the algorithm has made a sequence of choices that includes, possibly among others, the first i pairs from s', then pair $i+1$ from s will be in *choices*. Therefore, the algorithm will not run out of choices unless it produces an access script sequence that contains s as a subsequence. But at that point the matrix (t_{ij}) is guaranteed to be empty, because s must cover all non-blank entries in q according to property (3) of soundness, so the algorithm will terminate normally. □

As specified above, the algorithm will generate a single access script sequence. Multiple sequences can be generated by *backtracking* through the choices made at each iteration. We envision an interface where a proposed sequence, possibly annotated with information about the involved sources and execution costs, is presented to the user in response to his query. The user may execute the sequence and, if not satisfied with the results, try another one, or ask for an alternative right away.

4 Evaluating Access Script Sequences

Optimizations. An access script sequence produced by the algorithm above can in general be optimized (cf. [1, 2] for general optimization techniques for conjunctive queries). First, "useless" access script predicates can be recursively deleted. An access script predicate is useless in an access script sequence if: (1) all its outputs are variables, (2) none of these variables occur in a query predicate or some other access script predicate, and (3) the removal of the predicate does not violate soundness, which in this case requires that every variable or constant mentioned in the query still occurs somewhere else in the access script sequence.[6] Second, there is usually considerable potential for concurrent execution of access scripts. Since access scripts may have to wait for a significant amount of time for the underlying service providers to answer, concurrent execution of several access scripts may result in a substantial speedup. For this, it is necessary to establish the data dependencies between access script predicates. A data dependency between two access script predicates exists whenever the first predicate produces an output used as an input of the second predicate. By regarding the data dependencies as edges in a graph with the access script predicates as vertices, the access script sequence may be viewed as a directed acyclic graph. Nodes that are not connected by a directed path may be executed concurrently.

[6] Deleting useless predicates may alter the semantics of the access script sequence because the shorter sequence may have more answers. Nevertheless, we feel that this is a change for the better, because the deleted predicates represent connections that were not specified by the user.

Execution model. The actual execution of an access script sequence involves calling the various access scripts listed in the sequence with the proper arguments, while keeping track of the variable bindings generated during the process. Note that, given a set of tuples over the input attributes of the access script, each access script produces a set of tuples over the input and output attributes. The set of tuples produced is the natural join of the extension of the access script with the input set of tuples.

Variable bindings generated by access script invocations are kept in a *blackboard relation* B that is maintained throughout the evaluation process. This relation has one attribute for every variable mentioned in the access script predicates that have been processed so far. Initially, the schema of B is empty and B contains the single tuple $\langle\rangle$. Whenever an access script predicate $S_{\{I_1,\ldots,I_m\};\{O_1,\ldots,O_m\}}(e_1,\ldots,e_{m+n})$ is scheduled for execution, an *input relation* over $I_1,\ldots,I_m$ is built by projecting B onto the variables appearing in $(e_1,\ldots,e_m)$ and extending the resulting tuples with the constants appearing in $(e_1,\ldots,e_m)$ to obtain tuples over $I_1,\ldots,I_m$ (cf. the VTOA operator in [16], page 750). This input relation is then passed to the appropriate access script. The resulting *output relation* over $\{I_1,\ldots,I_m,O_1,\ldots,O_n\}$ is converted back to a relation b over the variables occuring in $(e_1,\ldots,e_{m+n})$ (cf. the ATOV operator in [16], page 747). This conversion may entail a selection if constants appear among $e_{m+1},\ldots,e_{m+n}$. The relation b is then joined with B (possibly extending the schema of B) to obtain an updated variable binding relation. After all access script predicates have been processed, the schema of B contains an attribute for every variable mentioned in the sequence, and the tuples in B represent valuations that make every access script predicate in the sequence true, i.e. answers to the sequence.

In order to reuse results produced by access scripts during the evaluation process, the result generated by an invocation of an access script may be cached (recall that the result is just some part of the extension of that access script). Whenever the access script is invoked again, the cache can be checked to see whether the relevant part of the extension is there. If so, the cached result may be reused.

5 Summary and Discussion

We have addressed the problem of providing a simple user interface for accessing a network of distributed, heterogeneous, and autonomous information sources. Our proposed solution is twofold: (1) encapsulate knowledge about the network in access scripts that are capable of handling basic queries and that present a directed relational interface to the outside, and (2) use combinations of these access scripts to answer more complex queries phrased in terms of a universal relation model. The main difficulty with this approach lies in determining how access scripts should be combined to implement a given query. This problem is related to the problem of implementing a universal relation interface to a set of ordinary relations, but due to the absence of dependencies in the schema and the restrictions on how access scripts may be combined, traditional solutions do not immediately apply in our setting. It does not seem possible in our case to give a simple denotational semantics for queries that can also be implemented operationally. Instead, we give a notion of "correct" answers to queries that serves as an upper bound for the set of acceptable answers and then show how to obtain those correct answers that can actually be retrieved given the restrictions on how access scripts may be accessed.

In future work, we plan to address the following issues:

1. Heuristics. It is usually not feasible to execute or even generate all sound access script sequences for a given query. It is therefore important that the system employ heuristics to quickly generate a sequence that matches the user's intuition about what a particular query means.
2. Cost models and optimization. So far, we have ignored the cost of invoking a particular access script. However, many information providers charge for their services, and therefore some ways of obtaining a piece of information may be more expensive than others. In addition, the quality (such as completeness or accuracy) of information providers may vary. The system should be capable of choosing optimal access script sequences based on cost/quality tradeoffs specified by the user.
3. User interfaces. Clearly, the user should have more intuitive means of expressing a query than conjunctions of predicates. We envision a user interface that uses menu trees to guide the user to the appropriate service category. Once the set of relevant attributes has been narrowed down sufficiently, a QBE-style interface may be used to obtain the actual query tableau.
4. Dynamic re-planning. If during the execution of an access script sequence an access script fails, the system should attempt to execute the query using an alternative sequence. This is easy to achieve if access script invocations have no side effects, but harder if an access script causes lasting changes in the environment (e.g., reserving a flight).

References

[1] A. Aho, Y. Sagiv, and J. Ullman. Efficient Optimization of a Class of Relational Expressions. *ACM Transactions on Database Systems*, **4** (1979), pp. 435–454.

[2] A. Aho, Y. Sagiv, and J. Ullman. Equivalences among Relational Expressions. *SIAM Journal of Computing*, **8** (1979), pp. 218–246.

[3] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proceedings of the IPSI Conference*, pp. 7–18, Tokyo, Japan, October 1994.

[4] G. de Michelis, E. Dubois, M. Jarke, F. Matthes, J. Mylopoulos, K. Pohl, J. Schmidt, C. Woo, and E. Yu. Cooperative Information Systems: A Manifesto. In *First IFCIS International Conference on Cooperative Information Systems, Architectural Issues in Cooperative Information Systems: A Tutorial*, Brussels, June 1996.

[5] C. Galindo-Legaria. Outerjoins as Disjunctions. In *Proceedings of ACM SIGMOD*, pp. 348–358, Minneapolis, USA, May 1994.

[6] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. In *Proceedings NGITS (Next Generation of Information Technologies and Systems)*, Naharia, Israel, November 1995.

[7] R. Hull and R. King. Reference Architecture for the Intelligent Integration of Information. Technical Report Version 1.0.1, Program on Intelligent Integration of Information, ARPA, May 1995. Available via ftp://isse.gmu.edu//pub/incoming/Gunning/I3_Arch_v._1.0.1.ps.Z.

[8] F. Leymann. A Survey of the Universal Relation Model. *Data & Knowledge Engineering* **4** (1989), pp. 305-320.

[9] L. Liu, C. Pu, and Y. Lee. An Adaptive Approach to Query Mediation across Heterogeneous Information Sources. In *Proceedings First IFCIS International Conference on Cooperative Information Systems*, pages 144–156, Brussels, Belgium, June 1996.

[10] D. Maier, D. Rozenshtein, and D. Warren. Window Functions. In P. Kanellakis and F. Preparata, editors, *Advances in Computing Research 3: The Theory of Databases*, pages 213–246. JAI Press, 1986.

[11] D. Maier, J. Ullman, and M. Y. Vardi. On the Foundations of the Universal Relation Model. *ACM Transactions on Database Systems*, **9** (1984), pp. 283-308.

[12] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *International Conference on Data Engineering*, pp. 251–260, Taipei, Taiwan, March 1995.

[13] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A Query Translation Scheme for Rapid Implementation of Wrappers. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, 1995.

[14] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering Queries Using Templates with Binding Patterns. In *Proceedings ACM PODS*, pp. 105–112, San Jose, California, May 1995.

[15] A. Rajaraman and J. Ullman. Integrating Information by Outerjoins and Full Disjunctions. In *Proceedings ACM PODS*, pp. 238–248, Montreal, Canada, June 1996.

[16] J. Ullman. *Principles of Database and Knowledge-Base Systems*, Volume 2. Computer Science Press, 1989.

[17] M. Y. Vardi. The Universal-Relation Data Model for Logical Independence. *IEEE Software*, pages 80–85, March 1988.

[18] M. Zloof. Query-by-Example: A Data Base Language. *IBM Systems Journal*, **16** (1975), pp. 324–343.