RC23245 (W0406-079) June 15, 2004
Computer Science

IBM Research Report

What the Meaning of *What* Is: Descriptive Naming of Data Providers in Context Weaver

Norman H. Cohen, Paul Castro, Archan Misra
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

What the Meaning of *What* Is: Descriptive Naming of Data Providers in Context Weaver

Norman H. Cohen, Paul Castro, Archan Misra
IBM T. J. Watson Research Center, Hawthorne, New York, USA
{ncohen, castrop, archan}@us.ibm.com

Abstract

Systems that use mobile, transient, or unreliable resources typically use descriptive names (also known as intentional or data-centric names) to specify those resources. Such names identify what data is needed rather than where that data is to be found. The Context Weaver middleware for collecting and composing data from pervasive networked data providers uses a descriptive naming system that is easily extended to handle the wide variety of data sources that exist today, as well as future data sources still unimagined. The system is based on a hierarchy of "provider kinds" and exploits emerging XML standards so that arbitrarily complex constraints can be specified.

1. Introduction

A number of systems are designed to obtain services from network resources such as sensors, cameras, printers, and web services. These resources may be mobile, they may be ephemeral, and their quality of service may fluctuate. It has become widely accepted that such systems should not require users to name a specific resource from which they wish to obtain services, but rather, to describe what the resource is expected to provide, so that the system can discover an appropriate resource.

This approach, known as descriptive [5], data-centric [11], or intentional [1] naming, has a number of advantages. It allows the system to select the best available resource, based on current conditions (including processor loads, network congestion, and other factors affecting quality of service), and to select a new resource when those conditions change. It makes an application robust against the failure of any one device. It accommodates the frequent addition of resources to, or removal of resources from, the system, without modification of the application that uses such resources. It allows an application written for one environment to be ported easily to another environment with a different set of resources.

Despite the consensus that has developed around queries that describe *what* is to be retrieved rather than *where* it is to be retrieved from, there is little consensus on what the meaning of *what* is. In much of the literature describing systems that support descriptive queries, the actual content of those queries is treated almost as an afterthought. We believe that there are important issues that must be addressed in defining a querying scheme that is capable of describing detailed constraints on any kind of existing resource, yet flexible enough to evolve gracefully as new kinds of resources are invented.

This paper examines the nature of the descriptive queries supported by the Context Weaver system, an evolution of iQueue [6]. Context Weaver *data providers* are categorized as belonging to *provider kinds*. Provider kinds are organized in a hierarchy of *subkinds* and *superkinds*, so that a query for providers of a kind *K* can be satisfied by providers belonging to any subkind of *K*. A data provider has certain properties, depending on its provider kind. A Context Weaver *provider query* names a provider kind and specifies a test to be applied to the properties meaningful for providers of that kind. This test may be an arbitrary boolean combination of arbitrary conditions. As new sources of data are devised, new provider kinds with their own sets of properties can be added. A provider kind can be inserted above or below specified already-existing provider kinds in the subkind-superkind hierarchy, allowing the hierarchy to evolve flexibly.

Our focus is on the query model, not the implementation, of Context Weaver. Section 2 gives a brief external view of Context Weaver and its requirements for descriptive-name queries. Section 3 explains the nature of these queries and the underlying model. Section 4 presents an enhancement to make these queries more expressive, and more likely to be satisfied. Section 5 discusses related work and Section 6 presents our conclusions.

2. Context Weaver

Context Weaver is designed to collect and combine data from a wide variety of resources called data providers. These include data providers external to the system—such as fixed and mobile sensors, web services, publish/subscribe services, and databases—as well as programmed entities that reside inside the system and generate values based on input data from other providers. A data provider may be passive, reporting its current value when asked; active, generating new values without being asked; or both. A Context Weaver application

issues a provider query and Context Weaver replies with a set of handles corresponding to data providers that have been registered with the system and satisfy the query. The application may query a handle for the current value of its data provider, and it may issue a subscription to a handle to be notified when the handle's data provider generates a new value. Thus, the subject of a provider query is not a value, but the continuously evolving stream of values associated with a data provider. A query may be reprocessed periodically as the dynamic properties of data providers fluctuate, possibly yielding different results. In addition, the query itself can be modified dynamically, based on previously received data. For example, as a mobile entity moves from point *a* to point *b*, a query for all data providers of a certain kind within a given radius of point *a* may be reprocessed as a query for all data providers of that kind within the given radius of point *b*.

Because Context Weaver is targeted to a wide variety of applications, the scheme for writing a provider query must be flexible enough to describe any data provider. Different applications may need to query, for example, for providers of Fahrenheit temperatures at a given latitude and longitude, Celsius temperatures of the patient in a given hospital bed, current prices of IBM stock in U.S. dollars, the number of the room where a given active badge was last sensed, and the identification numbers of all vehicles in a specified zone with excessive engine temperatures. Clearly, it is untenable to establish a fixed vocabulary of concepts and data types to be used in queries.

Neither can we rely on natural-language understanding, because we want a query to be precise. A given data provider should unambiguously satisfy, or fail to satisfy, a given provider query: A query for "providers of identification numbers of nearby lawn mowers" is ambiguous both because the notion of "nearby" is not well defined, and because (as a too-clever knowledge-based system would deduce) a "lawn mower" can be either a piece of machinery or a person using such machinery to mow a lawn.

So that our scheme is extensible to arbitrary domains, we must recognize arbitrarily complex data types, such as a location data type consisting of latitude and longitude components, each in turn consisting of degree, minute, and second components. We must also be able to test arbitrary conditions, such as that one point is within a given distance of another, that a given point lies within one of three specified polygons, or that a given point lies within one polygon but not within another, overlapping, polygon.

To satisfy these requirements, we turn to two emerging XML standards, XML Schema [8] and XQuery [4]. Every data provider registered with Context Weaver is described by an XML document called a *provider descriptor*. XML schemas can define arbitrarily complex data structures that can be contained in provider descriptors, and XQuery expressions can specify arbitrarily complex computations on the contents of these XML documents.

3. The nature of a provider query

We discuss underlying concepts related to provider queries in Sections 3.1 and 3.2, and turn to provider queries themselves in Section 3.3.

3.1. Provider kinds

Context Weaver provider queries are based on the notion of provider kinds. Every data provider is registered with Context Weaver as belonging to a particular provider kind. The definition of a provider kind specifies the data type of the values returned by the provider, the names and types of its *activation parameters*, and a set of attributes describing properties of the provider. Activation parameters provide the information needed to initialize a data-provider handle. Activation parameters might include, for example, the unique identifier of a particular real-world entity about which data is to be collected, or an authentication token. As Section 3.3 will explain in greater detail, a provider query names a provider kind; the query must specify a value for each of that provider kind's activation parameters, and the predicate of the query may refer to any of that provider kind's properties.

Figure 1 defines a provider kind for providers of the location of a vehicle with a specified vehicle identification number (VIN). This definition indicates that a provider of kind `VINToLocation` provides values of type `LatLongType` and is activated with a parameter `vehicleID` of type `VINType`, but it says nothing about the semantic relationship between the `LatLongType` value provided and the `VINType` value used for activation—that is, that the location provided is the location of the vehicle with the specified VIN. The definition in Figure 1 could apply just as easily to a kind for providers that give the location of *the registered owner of* the vehicle with the specified VIN. We expect a given provider kind to reflect a particular semantic relationship; providers with different semantics belong to different provider kinds, say `VINToVehicleLocation` and `VINToOwnerLocation`, that may

```
Provider kind VINToLocation:
    Type of provided values: LatLongType
    Activation parameters:
        vehicleID: VINType
    Properties:
        radiusOfErrorInMeters: float
        freshnessInSeconds: int
```

Figure 1. Definition of provider kind VINToLocation

happen to have the same provided type, activation parameters, and properties.

These semantic relationships reflect a human's view of the world. We do not attempt to formalize the semantics of a provider kind. Rather, we rely on the humans who (aided by search tools) name provider kinds in queries to be familiar with the intended semantics of those provider kinds, just as users of a relational database are expected to be familiar with the semantics of the tables they name in SQL queries.

The definition of new provider kinds is an ongoing administrative activity, as is the registration of new data providers whose provider kinds have been defined. Our work is largely shaped by the need to ensure that the provider-kind hierarchy can evolve smoothly over the course of weeks, months, and years. However, during the instant that a provider query is processed, the set of provider kinds and the set of registered providers can be regarded as static.

3.1.1.Subkinds and superkinds. Provider kinds can be organized into hierarchies of *superkinds* and *subkinds*, such that a query for a provider of kind *k* can be satisfied by a provider of any subkind of *k*. To formalize this hierarchy, we assume that the types to which provided values and activation-parameter values belong are themselves organized in a supertype-subtype hierarchy. (This is true of Context Weaver types, which are based on XML Schema types.) A provider kind *p* can be the *direct parent* of a child provider kind *c* only if each of the following conditions holds:

- The type of value provided by *c* is a subtype of the type of value provided by *p*.
- For each activation parameter of kind *c*, kind *p* has an identically named activation parameter, and the type of each parameter of *c* is a supertype of the type of the corresponding parameter of *p*. (Thus the set of parameter values that can be understood by a provider of kind *c* includes *at least* every parameter value that can be understood by a provider of kind *p*; *p* may have "extra" parameters that have no counterpart in *c*, which are ignored when activating a data provider of kind *c* as if it were of kind *p*.)
- The set of properties of *c* is a superset of the set of properties of *p*.

To these formal conditions, we add an informal one:

- The semantics of *c* (as understood informally by a human) should be consistent with the semantics of *p*.

(The formal conditions determine when it is *legal* to declare *p* to be a direct parent of *c*, and the informal condition determines when it is *appropriate* to do so.)

```
Provider kind VINTo3DLocation:
    Type of provided values: LatLongElevType
    Activation parameters:
        vehicleID: VINType
    Properties:
        radiusOfErrorInMeters: float
        freshnessInSeconds: int
```

Figure 2. Definition of provider kind VINTo3DLocation

The *superkinds* of a provider kind *k* consist of *k* and the superkinds of all direct parents of *k*; if *x* is a superkind of *y*, then *y* is a *subkind* of *x*. (Every provider kind is a subkind and a superkind of itself.)

For example, suppose the type `LatLongType`, giving a two-dimensional location in terms of latitude and longitude, has a subtype `LatLongElevType`, giving a three-dimensional location that also includes elevation above sea level. Figure 2 defines a kind for providers of three-dimensional locations of vehicles with a given VIN. Because `LatLongElevType` is a subtype of `LatLongType`, `VINTo3DLocation` is a subkind of `VINToLocation`. That is, a query for a provider of kind `VINToLocation` could be satisfied by provider of kind `VINTo3DLocation`; an application would use the `LatLongElevType` values it receives from the provider as if they were `LatLongType` values.

Some providers of vehicle-location information might use GPS receivers on the vehicles, and for those providers it is meaningful to define an additional property, the number of GPS satellites contributing to the reading. Figure 3 defines a provider kind for these GPS-based providers.

```
Provider kind VINToGPSLocation:
    Type of provided values: LatLongType
    Activation parameters:
        vehicleID: VINType
    Properties:
        radiusOfErrorInMeters: float
        freshnessInSeconds: int
        satellites: int
```

Figure 3. Definition of provider kind VINToGPSLocation

`VINToGPSLocation` is also a subkind of `VINToLocation`, since its properties include all the `VINToLocation` properties. Any query for a provider of kind `VINToLocation` can be satisfied by a provider registered as having kind `VINToGPSLocation`. (The query will not refer to the `satellites` property, since that property is not defined for the provider kind in the query, `VINToLocation`.)

We can go one step further, and define a provider kind for GPS-based providers of three-dimensional location, as shown in Figure 4. `VINToLocation`, `VINTo3DLocation`, and `VINToGPSLocation` all qualify to be direct parents of `VINToGPS3DLocation`. Indeed, we allow a kind to have more than one direct parent. If we define `VINTo3DLocation` and `VINToGPSLocation` to be direct parents of `VINToGPS3Dlocation`, we obtain the subkind hierarchy shown in Figure 5. This hierarchy indicates that a query for a provider of kind `VINToLocation` could be satisfied by a provider registered as having kind `VINToGPS3DLocation`, `VINTo3DLocation`, `VINToGPSLocation`, or `VINToLocation`; a query for a provider of kind `VINToGPSLocation` could be satisfied by a provider registered as having kind `VINToGPS3DLocation` or `VINToGPSLocation`; and a query for a provider of kind `VINToGPS3DLocation` can be satisfied only by a provider registered as having kind `VINToGPS3DLocation`.

3.1.2. Bottom-up definition of superkinds. Traditionally, subtype hierarchies are built from the top down; that is, the definition of a type names its direct parents, which must have been defined earlier. In contrast, the definition of a new provider kind in Context Weaver allows both a set of direct parents and a set of direct children to be named. Thus the new provider kind can be installed as a superkind of some existing kind, as a subkind of some existing kind, or wedged between two existing kinds as the subtype of the first and the supertype of the second, provided that no circularity results.

Just as top-down growth of a hierarchy allows for specialization, the bottom-up growth of a hierarchy allows for *generalization*. Such generalization allows the vocabulary of provider queries to evolve without disruption as new provider kinds are devised. We give two examples.

First, suppose there is a standard type `TelematicsData` that has been extended independently by company X to a type `XTelematicsData` and by company Y to a type `YTelematicsData`. Each company markets a device that reports a value of its own extended telematics-data type, given a VIN. These devices correspond to provider kinds `VINToXTelematicsData` and `VINToYTelematicsData` defined by the two companies. We are managing a fleet that had been using company X's device to obtain standard `TelematicsData` values (by treating `XTelematicsData` values as `TelematicsData` values, ignoring company X's extensions); however, we have now added vehicles with company Y's devices to the fleet. So that we can write a query that will find all providers of `TelematicsData` values, regardless of which devices they use, we define a new provider kind, `VINToTelematicsData`, as a superkind of `VINToXTelematicsData` and `VINToYTelematicsData`.

A second use of bottom-up superkind definition involves activation parameters rather than provided values. Suppose we have some data providers, of kind `VINToLocation`, that provide the location of a vehicle given its VIN, and other data providers, of kind `PlateToLocation`, that provide the location of a vehicle given its license-plate number. Suppose further that we have both the VIN and license-plate number of all vehicles of interest. Rather than issue one query for `VINToLocation` data providers and, if that fails, a second query for `PlateToLocation` data providers, we can define a new provider kind `VINAndPlateToLocation`, which takes both a VIN and a license-plate number as activation parameters. Since `VINAndPlateToLocation` has activation parameters corresponding to those of both `VINToLocation` and `PlateToLocation`, `VINAndPlateToLocation` can be defined as a superkind of both those provider kinds. Then we can issue a single query for `VINAndPlateToLocation` data providers, which will be satisfied by both `VINToLocation` data providers and `PlateToLocation` data providers.

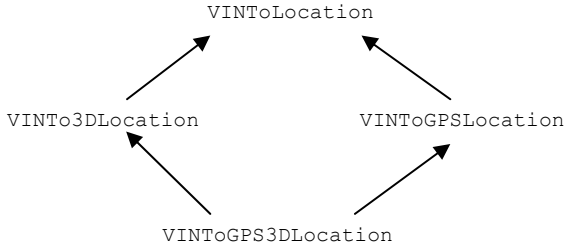

Figure 5. A multiple-inheritance subkind hierarchy showing VINTo3DLocation, VINToGPS3DLocation, and VINToGPS-Location as subkinds of VINToLocation

3.2. Provider descriptors

Every data provider has a *provider descriptor* that conveys the identity of the provider and some information about its state. This information may include static information about the nature and capabilities of the data provider as well as dynamic information; the dynamic information may include provider's current value, as well as information about the quality of information and quality of service currently being provided. A provider query is, essentially, a test that a given provider descriptor either

Provider kind `VINToGPS3DLocation`:
  Type of provided values: `LatLongElevType`
  Activation parameters:
    `vehicleID: VINType`
  Properties:
    `radiusOfErrorInMeters: float`
    `freshnessInSeconds: int`
    `satellites: int`

Figure 4. Definition of provider kind VIN-ToGPS3DLocation

```
<provider id="VTL003" kind="VINToLocation">
  <streamProperties>
    <property name="freshnessInSeconds">
      <data type="#long">60</data>
    </property>
    <property name="radiusOfErrorInMeters">
      <data type="#int">50</data>
    </property>
  </streamProperties>
  <value>
    <data type=
     "http://example.org/types#LatLongType">
        <lat>38.8976</lat>
        <long>-77.0366</long>
    </data>
  </value>
</provider>
```

Figure 6. A provider descriptor for a VINToLocation data provider

passes or fails. Sometimes, this test can be performed without fully materializing the provider descriptor.

A provider descriptor includes:

- a unique identifier for the data provider
- the name of its provider kind
- values for the properties defined for providers of that kind
- the provider's current value

If a provider kind *s* is a subkind of a provider kind *k*, a descriptor for a provider of kind *s* includes at least the properties found in a descriptor for a provider of kind *k*. In Context Weaver, a provider descriptor is represented in XML, as illustrated in Figure 6.

3.3. Provider queries

A provider query has four elements:

- the name of a provider kind, indicating that a provider of that kind or one of its subkinds is desired
- values for the activation parameters associated with that provider kind
- a predicate, possibly referring to the values of stream properties associated with the provider kind, to be applied to the property values in a given provider descriptor, yielding *true* if the descriptor should be considered to satisfy the query, and *false* otherwise
- a *selection mechanism* for determining which provider descriptors, among those determined to satisfy the query, should be returned in the query result

Since Context Weaver provider descriptors are XML documents, the predicates in Context Weaver provider queries are boolean-valued XQuery [4] expressions applied to provider descriptors. For example, to test whether a provider descriptor like that in Figure 6 represents a provider with a `radiusOfErrorInMeters` property less than 75 and a value with a latitude greater than 38, we could write the following XQuery expression:

```
/provider/streamProperties/property
   [@name="radiusOfErrorInMeters"] lt 75 and
/provider/value/data/lat gt 38.0
```

Presently, the selection mechanism is a two-valued field: The value *ALL* indicates that a list should be returned containing every provider descriptor satisfying the query, and the value *ANY_ONE* indicates that at most one provider descriptor, chosen arbitrarily by the query engine from among those satisfying the query, should be returned. We envision a more powerful selection mechanism, consisting of two parts:

- an integer-valued expression, possibly referring to the properties and current value of a particular stream, to be applied to a given provider descriptor, yielding a *provider-descriptor score*
- a criterion indicating how to select the descriptors to be returned, such as returning the *n* top-scoring descriptors, returning all descriptors with a score greater than or equal to *x*, returning the first *n* descriptors found, or returning all descriptors found within a given timeout interval

(A similar mechanism has been proposed [17] as an extension to the Service Location Protocol.)

3.3.1. Predicates versus activation parameters. The predicate and the activation parameters play distinct roles. The predicate tests whether the properties and value of a data provider satisfy certain conditions, but does not necessarily constrain any property to hold one specific value. The activation parameters supply specific values needed to establish and initialize a connection to a data provider.

Sometimes, the same information must be supplied redundantly as an activation parameter and in a predicate. Consider, for example, a query for providers of IBM stock prices. Some providers of this data might be general stock-quote services, which require a stock symbol to be

Provider kind `PriceBySymbol`:
  Type of provided values: `USDollars`
  Activation parameters:
    `symbolParameter: string`
  Properties:
    `symbolProperty: string`
    `tickerDelayInMinutes: int`

Provider kind `IBMPrice`:
  Type of provided values: `USDollars`
  Activation parameters: (none)
  Properties:
    `symbolProperty: string`
    `tickerDelayInMinutes: int`

Figure 7. Definition of provider kind PriceBySymbol and its subkind IBMPrice

passed as an activation parameter and supply a price for a stock identified by that symbol; other providers might be dedicated specifically to providing the price of IBM stock. These two varieties of providers might belong, respectively, to the provider kind `PriceBySymbol` and to its subkind `IBMPrice`, defined in Figure 7. We write a query for the provider kind `PriceBySymbol`, so that the query can be satisfied by a provider of either kind. However, the provider kind `PriceBySymbol` would also be matched by providers belonging to other subkinds of `PriceBySymbol`, such as `IntelPrice` and `MicrosoftPrice`. To filter out data providers of these other subkinds, we write a query that not only specifies a value of `"IBM"` for `symbolParameter` (as required for providers of kind `PriceBySymbol`) but also specifies the following predicate:

```
/provider/streamProperties/property
    [@name="symbolProperty"] eq "IBM"
```

In the case of general stock-quote services, the value `symbolParameter` is used to obtain the price of the desired stock. The predicate filters out the undesired data providers by testing the value of `symbolProperty`.

Some existing descriptive naming schemes test attributes only for equality with specific values. With such schemes, there is no need to distinguish between parameters and properties: The string "`symbol=IBM`" can be understood both as a specification of the value that is to be used for `symbol` (when activating a data provider requiring a specific value) and as a test to be performed on the value of the property `symbol` (when filtering provider descriptors that contain a `symbol` property). However, by restricting a query to be, in essence, a conjunction of equalities, such a scheme precludes queries for, say, a stock-price provider with a ticker delay *less than* 20 minutes, or a location lying within *any* of a set of polygons.

3.3.2. Semantics of a provider query. We define the semantics of a provider query operationally: A provider query specifying a provider kind *pk*, activation parameters $ap_1,\ldots,ap_n$, predicate *p*, and selection mechanism *sm* is applied *as if* by doing each of the following in turn:

- attempting to activate every data provider registered as belonging to some subkind of *pk*, using the activation-parameter values $ap_1,\ldots,ap_n$;
- for each successfully activated provider, constructing a provider descriptor appropriate for kind *pk* with the properties and current value of that provider;
- applying the predicate *p* to each provider descriptor and including all those for which the result is *true* in a set of candidates; and
- applying the selection mechanism *sm* to select a result set from the set of candidates.

In practice, this effect can often be achieved more efficiently: If the predicate does not refer to dynamic properties of a data provider, or to its value, the predicate can be applied to an *approximate descriptor*, containing only static properties, created without actually activating the provider it describes. (This approach is reminiscent of the approximate caches of [14].) If the predicate refers to dynamic properties, but not to the value, it is necessary to activate the provider, but not to retrieve its current value. If the selection mechanism establishes an upper bound *b* on the number of provider descriptors to be returned, but allows those *b* descriptors to be chosen arbitrarily from among those descriptors satisfying the predicate, the query processing can be stopped after *b* descriptors have been obtained. Traditional indexing and query-optimization techniques can be applied to static properties to avoid the retrieval of provider descriptors that cannot possibly satisfy the predicate in a query.

Allowing a predicate to refer to a data provider's value is potentially costly: In the general case, resolving a query entails activating every provider that has a suitable provider kind, and requesting its current value. However, this feature is also very powerful. For example, we can write a query that is matched by all vehicles whose current telematics data indicate that they are located in a specified geographic area, or by all temperature sensors currently sensing temperatures in a dangerous range. Fortunately, a provider-query resolver can be implemented so that the cost of value-based queries is borne only by those queries whose predicates explicitly refer to a provider's value. Rather than guessing whether the cost is worth the benefit, our approach is to allow value-based queries while alerting application developers to the potential cost, and allowing developers to make their own choices.

4. Synthesis of data providers

In Section 3, we made no attempt to formalize the semantics of a provider kind. However, by identifying certain specifiable aspects of a provider kind's semantics, such as units of measurement or the use of a sliding average, we enable Context Weaver to analyze relationships among various provider kinds, and to satisfy a provider query by synthesizing a new data provider.

Consider two provider kinds that take a VIN as an activation parameter and provide the velocity of the corresponding vehicle, one in miles per hour and the other in kilometers per hour. Neither of these provider kinds can be a subkind of the other, because their semantics are incompatible. Nonetheless, their semantics are closely related: A query for the velocity of vehicle *v* in miles per hour can be satisfied by discovering a data provider for the velocity of vehicle *v* in kilometers per hour, and synthesizing a new data provider whose current value is obtained by multiplying the result of the discovered data provider by 0.62. More generally, it is possible to satisfy

any query of the form *x* in miles, given *y,* by discovering a data provider for *x* in kilometers, given *y,* and synthesizing a new data provider that multiplies the result of the discovered data provider by 0.62.

Two difficulties arise. First, there are too many pairs of provider kinds—distance of a vehicle from Chicago in miles given its VIN versus distance of vehicle from Chicago in kilometers given its VIN, altitude of an aircraft in miles given its tail number versus altitude of an aircraft in kilometers given its tail number, *ad infinitum*—to exploit relationships on an *ad hoc* basis. Second, the scheme we have presented so far has no entity whose role is to multiply arbitrary values by 0.62. A data provider is characterized, in part, by the semantics of the value it provides; the semantics of a component that takes an input and multiplies it by 0.62 depends on the semantics of the input.

To address these difficulties, we enhance our approach as follows:

- We introduce the notion of a *stream transformer*, whose role is to transform a stream of input-data values in a methodical way to produce a stream of output-data values.
- We specify certain aspects of data-provider semantics, such as units of measurement, in provider-kind definitions and provider queries.
- We introduce *synthesis rules* asserting that queries matching particular patterns can be satisfied by applying particular stream transformations to appropriate input streams.

The net effect of these enhancements is to enable a provider of the kind requested by a provider query to be synthesized by applying an appropriate stream transformer to another, closely related, data provider, based on a catalog of synthesis rules. The closely related data provider may be discovered, or it may itself be synthesized. The following subsections elaborate on this approach.

4.1. Stream transformers

A *stream transformer* is a mechanism that reads one or more streams of input values and generates a stream of output values determined by the input streams. Stream transformers can be classified as value-based or stream-based. A *value-based* stream transformer generates one output value for each input value (or each set of contemporaneous input values, one from each of several input streams); the output value is determined by the input value (or values). An example is a stream transformer that emits the value 0.62*v* every time a value *v* arrives. A *stream-based* stream transformer generates output values based on the history of input values up to a certain point and the passage of time; it might not generate an output value for each input value, and might generate output values that do not correspond to any input value. An example is a stream transformer that emits a value every second, equal to the average of the values that arrived in the past minute.

4.2. Specifiable aspects of semantics

We stated in Section 3.1 that because a human's view of the world cannot be formalized, we cannot define the semantics of a provider kind completely. We can, however, specify certain important aspects of provider-kind semantics. Our approach is to decompose the semantics of a provider kind into specifiable parts and an unspecifiable part. The specifiable parts are specified in provider-kind definitions using a small, closed, precisely defined vocabulary. The unspecifiable part corresponds to a token whose meaning, like the provider-kind name in Section 3, is understood by humans.

One specifiable aspect of provider-kind semantics is the *dimension* of the provided data. (We use the term *dimension* in the sense in which it is used in physics: Mass, length, and time, among others, are dimensions; and centimeters, inches, and light years are some of the units appropriate for the dimension of length.) In the case of a provider kind for vehicle location, given a VIN, the fact that the provided data has the dimension *location* is specifiable; the fact that the provided location is that of the vehicle with the given VIN, rather than that of the vehicle's owner, is not.

A second specifiable aspect is *representation*. For any given dimension, there is a small number of well-known representations. In the case of physical dimensions, the representation has two components: the unit of measurement and the numeric format. Thus, for the dimension of *length*, representations include a number of centimeters represented in IEEE 754 single-precision floating-point format and a number of inches represented in decimal format. For the dimension of *time* (a point on a timeline), representation would encompass various date/time formats (such as "1985-05-15 1:30 pm" and "15 May 1985 13:30") and the binary 64-bit representation of the number of milliseconds elapsed since the start of the year 1970.

Additional specifiable aspects of semantics concern the relationship of values in a stream to each other, and to the passage of time. Such aspects are used in describing the streams that result from applying stream-based stream transformers. Examples are a specification that a stream is a time series of smoothed averages, or a specification that values satisfying a specified condition have been filtered out of the stream. Figure 8 shows what a provider-kind definition with specifiable semantics might look like; the token `VehicleDistanceFromChicago` captures the unspecified semantics. In place of the type of the provided values we saw in the provider-kind definitions of Section 3, we now specify the dimension, unit, and format of the provided values.

4.3. Synthesis rules

The application of stream transformers is driven by synthesis rules. A synthesis rule consists of an *output provider-query pattern*, the name of a stream transformer, and one or more *input provider-query patterns*. A provider-query pattern is a provider query in which the name of a provider kind is replaced by a provider-kind definition, and then certain entities within the query are replaced by *match variables*. Only match variables appearing in the output provider-query pattern may appear in an input provider-query pattern. A synthesis rule asserts that, if a query Q is an instance of the output provider-query pattern, then a data provider satisfying Q can be synthesized by applying the named stream transformer to input sources that satisfy corresponding instances of the input provider-query patterns (i.e., instances in which the same values are substituted for the match variables).

A query for a provider of the provider kind `VINToDistanceFromChicago` of Figure 8 is an instance of the provider-query pattern of Figure 9, in which match variable *`fmt`* corresponds to `float`, match variable *`paramDecls`* to

```
vehicleID: VINType
```

(a one-element list of activation-parameter declarations), match variable *`sem`* to the token `VehicleDistanceFromChicago`, match variable *`propList`* to

```
radiusOfErrorInMeters: float
freshnessInSeconds: int
```

(a two-element list of property declarations), match variable *`paramVals`* to a list containing one VIN, and match variable *`pred`* to a predicate potentially testing `radiusOfErrorInMeters` and `freshnessInSeconds`. An input query pattern `SomethingInKilometers` could be defined identically, except that the units would be specified as kilometers rather than miles. One synthesis rule would assert that, for all appropriatedly typed match-variable values *u*, *v*, *w*, *x*, *y*, and *z*, a query `SomethingInMiles`<*u*,*v*,*w*,*x*,*y*,*z*> (i.e., an instance of the output provider-query pattern `SomethingInMiles` using the values *u*, *v*, *w*, *x*, *y*, and *z* for the match variables) can be synthesized by obtaining a provider satisfying the query `SomethingInKilometers`<*u*,*v*,*w*,*x*,*y*,*z*> and applying to its output a stream transformation that multiplies each value in the stream by 0.62.

5. Related work

Two limitations characterize previous approaches to descriptive naming. Some approaches lack the expressiveness of our XQuery-based approach, effectively restricting the conditions that can be tested to a conjunction of equalities. Some approaches do not support a hierarchical classification of the descriptively named entities akin to our provider-kind hierarchy. Some approaches suffer from both of these limitations.

Schemes in which queries are equivalent to a conjunction of equalities are deficient in two respects. First, they are incapable of expressing constraints based on ordering comparisons (e.g., that a printer's speed should be *at least* 10 pages per minute). Second, they are incapable of expressing disjunctions (e.g., that a mobile device's location should be in *either* polygon *A* or polygon *B*).

Provider kind `VINToDistanceFromChicago`:
- Dimension of provided values: `distance`
- Representation of provided values:
  - Unit: `miles`
  - Format: `float`
- Activation parameters:
  - `vehicleID: VINType`
- Semantics: `VehicleDistanceFromChicago`
- Properties:
  - `radiusOfErrorInMeters: float`
  - `freshnessInSeconds: int`

Figure 8. Definition of provider kind VINToDistanceFromChicago

Nonhierarchical classifications of descriptively named entities do not evolve gracefully as new kinds of entities are introduced. A new entity distinct in any small way from previously categorized entities must be placed in its own new category; there is no way to express the common features among categories, such as the generalization mechanism described in Section 3.1.2. Since there is no way to specify a category consisting of a set of closely related subcategories, a query meant to cover such a category must be split up into a number of narrower queries.

The Lightweight Directory Access Protocol (LDAPv3) [15] allows attributes of a directory entry to be tested using an arbitrary boolean search filter. However, the hierarchy of an LDAP directory does not reflect superkind-subkind relationships: There is no straightforward way to cause a query for an entry of type *t* to be satisfied by an entry whose type is any subtype of *t*.

In the Service Location Protocol (SLPv2) [9], queries include an LDAPv3 search filter. Every SLP service belongs to some *concrete service type*, and concrete service types may be grouped into *abstract service types*. However, only a two-level hierarchy is supported.

In the Ninja project's Service Discovery Service (SDS) [7], a service has an XML service description, analogous to our provider descriptors. However, there is no unifying framework for the various forms of service description. A query takes the form of a service description with some subset of the elements removed, and is equivalent to a conjunction of equalities between the values in the query and the corresponding values in the service description.

The Intentional Naming System [1] and its follow-on Twine [2] take an approach similar to that of SDS: Both queries and resource descriptions take the form of *attribute-value trees*. A query matches a resource description if and only if each path from the root of the query tree has a corresponding path in the resource description, so a query is, in essence, a conjunction of equality tests. While [1] contemplates the addition of ordering comparisons to queries, [2] exploits the fact that a path in a query matches a path in a resource description only if a hashing function would map both paths to the same hash code.

Query pattern
`SomethingInMiles`
`<fmt, paramDecls, sem, propList, paramVals, pred>`:

Provider kind:
Dimension of provided values: `distance`
Representation of provided values:
Unit: `miles`
Format: *`fmt`*
Activation parameters:
[*`paramDecls`*]
Semantics: *`sem`*
Properties:
[*`propList`*]

Activation parameter values: [*`paramVals`*]

Predicate: *`pred`*

Selection mechanism: *ANY_ONE*

Figure 9. A query pattern

In the *directed diffusion* paradigm [11][10], queries are called *interests*. In [11], an interest is expressed as a set of attribute-value pairs, interpreted as a conjunction of equality tests. Every interest includes a `type` attribute whose value is an event code naming the subject of the query. Event codes roughly correspond to provider-kind names, but with no inherent relationships among the notions they denote. A variant on this scheme, using attribute-comparator-value triples instead of attribute-value pairs, is presented in [10]; an interest is still equivalent to a conjunction of simple tests, but the simple tests may include ordering comparisons as well as equalities.

The Jini [15] Lookup Service discovers Java objects representing services. A service object is described by a *service item* that includes a unique *service identifier* and zero or more *attribute sets*. A query takes the form of a *service template* consisting of an optional service identifier, zero or more Java types, and zero or more *attribute-set templates*. A service template matches a service item if the service identifier in the service template is absent or identical to the service identifier in the service item, the service object in the service item is an instance of each of the Java types in the service template, and each attribute-set template in the service template matches an attribute set in the service item. The hierarchy of Java service-object subclasses can play a role analogous to our hierarchy of subkinds, but the attribute-set template specifies a conjunction of exact matches with specified values.

Universal Description, Discovery, and Integration (UDDI) [3] is a framework for issuing queries to discover web services. The UDDI registry was originally conceived of as a "yellow pages" directory, in which services are looked up by first locating a business in a given industry and then examining the services offered by that business. Businesses are categorized by industry according to a hierarchical scheme, but this scheme has no semantic significance in the processing of queries. There is no mechanism for looking up services based on their semantic properties.

In contrast to these approaches that are less flexible than ours, ontology-based systems aspire to provide greater flexibility. The OWL Web Ontology Language [13], an enhancement of the Resource Description Framework (RDF) [12], is an example of a notation for defining ontologies and annotating web pages to relate them to concepts defined in these ontologies. The goal of typical ontology-based systems is to support unstructured queries, in particular natural-language queries, and to apply

common-sense reasoning to deduce facts that are not explicitly represented. In addition, by defining relationships between terms in different vocabularies, an ontology provides a bridge between the vocabulary of a query and the vocabulary of a provider descriptor, allowing providers of "thermometer reading" to be discovered in response to a query for "temperature." The goals of ontology-based systems are ambitious, but their promises are yet unproven. Development of ontologies is labor-intensive, so few exist yet, and it is not clear that resources will exist in the long run to maintain them. Our hierarchy of provider kinds can be viewed as a kind of primordial ontology, with less ambitious and therefore more attainable goals. We seek to classify only data providers rather than arbitrary knowledge, and we do so in a highly constrained manner.

6. Summary

Context Weaver discovers data providers that generate value streams satisfying application-specified queries. These queries *describe* the desired properties of value streams rather than identifying particular data providers. Descriptive naming of resources allows a system such as Context Weaver to select the best available resource dynamically, isolates client applications from dependence on one particular resource, allows resources to be added to or removed from the system without modifying client applications, and makes an application portable to other environments with different sets of resources.

Context Weaver provider queries are open-ended enough to be generally applicable to arbitrary domains, and to sorts of data providers not yet conceived of. At the same time, provider queries are precise and unambiguous. The structure of the query is amenable to consistency checks to ensure that a query refers to only those attributes that are meaningful for the kind of data provider it describes. It is possible to perform general tests on these attributes, including range tests and disjunctions.

Data providers registered with Context Weaver are classified according to a multiple-inheritance hierarchy of provider kinds. New provider kinds can be inserted in this hierarchy not only below, but also above specified existing provider kinds, facilitating the introduction of a new provider kind that generalizes previously existing provider kinds. The current state of a registered data provider is described by an XML provider descriptor with content that depends on its provider kind. A provider query specifies the name of a provider kind, a set of activation-parameter values meaningful for that provider kind, an XQuery predicate applicable to descriptors for providers of that kind, and a selection mechanism specifying how data providers are to be selected from among those that are eligible. The descriptor includes the current value of a data provider, enabling queries for all data providers currently providing values that satisfy a particular condition.

Successful discovery of data providers can be facilitated by dynamically synthesizing data providers of the kind specified in a query. Such synthesis applies generic stream transformations to other, closely related data providers. To write synthesis rules specifying that the application of a given stream transformation to the output of a provider of one given kind yields a provider of another given kind, we must formalize certain aspects of a provider kind's semantics, such as the dimension and representation of the provided data.

Most of Context Weaver's requirements for descriptive naming are shared by other systems that obtain services from a variety of resources with fluctuating characteristics, such as mobile resources. Therefore, a naming system based on hierarchies of resource kinds, and arbitrary predicates over attributes meaningful for a given resource kind, would be appropriate for such systems.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99), December 12-15, 1999, Kiawah Island Resort, South Carolina, published as Operating Systems Review 33, No. 5 (December 1999), 186–201

[2] Magdalena Balazinska, Hari Balakrishnan, and David Karger. INS/Twine: a scalable peer-to-peer architecture for intentional resource discovery. International Conference on Pervasive Computing (Pervasive 2002), Zurich, Switzerland, August 26–28, 2002, 195–210

[3] Tom Bellwood, Luc Clément, Claus von Riegen, eds. UDDI version 3.0.1. UDDI Spec Technical Committee Specification, October 14, 2003 <URL: http://uddi.org/pubs/uddi_v3.htm>

[4] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Working Draft, May 2, 2003 <URL: http://www.w3.org/TR/xquery/>

[5] Mic Bowman, Saumya K. Debray, and Larry L. Peterson. Reasoning about naming systems. ACM Transactions on Programming Languages and Systems 15, No. 5 (November 1993), 795–825

[6] Norman H. Cohen, Apratim Purakayastha, Luke Wong, and Danny L. Yeh. iQueue: a pervasive data-composition framework. 3rd International Conference on Mobile Data Management, Singapore, January 8–11, 2002, 146–153

[7] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and

Networking (MobiCom '99), Seattle, Washington, August 15–19, 1999, 24–35

[8] David C. Fallside, ed. XML Schema Part 0: Primer. W3C Recommendation, May 2, 2001 <URL: http://www.w3.org/TR/xmlschema-0/>

[9] E. Guttman, C. Perkins, J. Veizades, M. Day. Service Location Protocol, Version 2. IETF RFC 2608, June 1999 <URL: http://www.ietf.org/rfc/rfc2608.txt>

[10] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP 2001), Banff, Alberta, October 21–24, 2001, 146–159

[11] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCom 2000), Boston, Massachusetts, August 6–11, 2000, 56–67

[12] Frank Manola and Eric Miller, eds. RDF Primer. W3C Working Draft, October 10, 2003 <URL: http://www.w3.org/TR/rdf-primer/>

[13] Deborah L. McGuinness and Frank van Harmelen, eds. OWL Web Ontology Language overview. W3C Candidate Recommendation, August 18, 2003 <URL: http://www.w3.org/TR/owl-features/ >

[14] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, May 21–24, 2001, 355–366

[15] Sun Microsystems. Jini Technology Core Platform Specification. Version 2.0, June 2003 <URL: http://wwws.sun.com/software/jini/specs/>

[16] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). IETF RFC 2251, December 1997 <URL: http://www.ietf.org/rfc/rfc2251.txt >

[17] W. Zhao, H. Schulzrinne, E. Guttman, C. Bisdikian, and W. Jerome. Select and sort extensions for the Service Location Protocol (SLP). IETF Network Working Group RFC 3421, November 2002, <URL: http://www.ietf.org/rfc/rfc3421.txt>