Dynamic Links for Mobile Connected Context-Sensitive Systems

Philip Gray and Meurig Sage

Department of Computing Science, University of Glasgow, UK
{pdg, meurig}@dcs.gla.ac.uk

Abstract The current generation of mobile context-aware applications must respond to a complex collection of changes in the state of the system and in its usage environment. We argue that dynamic links, as used in user interface software for many years, can be extended to support the change-sensitivity necessary for such systems. We describe an implementation of dynamic links in the Paraglide Anaesthetist's Clinical Assistant, a mobile context-aware system to help anaesthetists perform pre- and post-operative patient assessment. In particular, our implementation treats dynamic links as first class objects. They can be stored in XML documents and transmitted around a network. This allows our system to find and understand new sources of data at run-time.

1 Introduction

Early interactive systems just had to deal with mapping input from input devices onto application operations and system changes to output devices. However, current interactive systems exhibit a much higher degree of potential sensitivity to change in surrounding environment. In addition to changes in input they must also handle changes to sensors and incoming information from distributed information sources.

The modelling and implementation of this change-sensitivity is generally ad hoc. Attention has been given to the forms of change that are usefully exploited, to the methods of storage and communication of such changes in distributed systems, but little attention to the general mechanisms for mediating application-oriented links between change in source data and its desired consequential effect.

We propose a general software mechanism to support a variety of related types of change sensitivity. Given their genericity of structure and the dynamic environments in which they have to operate, we have also designed them to be highly configurable and able to respond to changes in their run-time environment.

In Section 2 we provide a general introduction to the notion of dynamic links and discuss their relationship to distributed context-sensitive interactive information systems. Section 3 describes the setting in which our work was carried out, the Paraglide Project, identifying the domain-related challenges that we have tried to meet via our generic configurable link structure. Section 4 presents our model and its implementation in the Paraglide Clinical Assistant. Finally Section 5 offers some conclusions and directions for further work.

2 Dynamic Links

2.1 What Is a Dynamic Link?

The notion of dynamic link is based on the concept of constraints as value dependencies. That is, some data element e is constrained by condition c if the value of e depends on the value of c. Constraint-satisfaction systems and related constraint specification languages have proved useful for a variety of applications in which value dependencies are volatile and subject to change. A number of successful user interface development environments, for example, have been implemented using constraints to specify the interactive behaviour of graphical elements ([2],[7]).

A dynamic link, in our sense, is a reified constraint. That is, it is a value dependency represented by an object in the run-time system itself that defines a relationship that may result in changes to the state of the link destination based on changes to the link source over the lifetime of the link.

2.2 Dynamic Link Structures

In the domain of user interface software, constraint-based mechanisms have been used for at least twenty years, although in their early incarnations the constraints were not always instantiated as links. Smalltalk's MVC provides a notify/update mechanism for creating constraints between view and model components. However, the mechanism is not intended to be visible; it implements constraints as implicit links. It is possible, but not necessary, to create specialised model components that interpose between the source data and the dependent view. Such models can be viewed as dynamic links. Other approaches have made the links explicit and hence configurable [6].

The Iconographer and Representer systems treat the link as a central configurable element, with special visual programming tools for the configuration ([4], [5]). Little recent work has revisited this issue and we are now confronted with user interface components with complex interactive structures with only poorly configurable interfaces between linked components.

Similarly, dynamic links in hypermedia systems offer the potential to make the usually fixed document associations dynamically configurable, so that they reflect different potential views onto the document or so that they can change to accommodate changes in the remote resources to which the links can point [3].

Modern distributed systems architectures provide mechanisms for the implementation of distributed link structures (i.e., those in which source or destination of the link resides in a remote environment). For example, Elvin [8] and JMS[1] offer facilities for establishing subscription-based notifications and data delivery from remote servers. However, while this supplies enabling technology for the link, it is not sufficient to create the link itself, which often requires access to application-oriented data and operations.

[1] JMS is a messaging standard defined for Java by Sun Microsystems (http://java.sun.com/). We are using the iBus//MessageServer implementation of JMS, produced by SoftWired Inc. (http://www.softwired-inc.com).

2.3 Dynamic Links in Context-Sensitive Interactive Systems

Our concern in this paper is not with constraints or even dynamic links in general, but in their application to distributed interactive systems, particularly those in which client services are mediated via small, mobile context-aware devices such as PDAs and wearable devices. In this domain, links can serve a number of roles, particularly relating local data elements to

- other local data
- data from remote services
- data from external sensors.

By generalising over these different forms of link we can

- hide from the link ends the nature of the link, improving reusability via information hiding and
- centralise relevant domain knowledge for use across different link sources and destinations.

Link effects may vary according to the aspect of the target data that is affected. Thus, the linked source may cause a change in the value of the target (the most common relationship). However, it might also cause a change in the likelihood of certain values being appropriate.

We can distinguish between link update effects. The link may actually cause a change to the value of the destination object or simply notify the destination that an appropriate change has taken place in the link source and let the destination object take appropriate action.

Actual link behaviour is also variable. In some cases, the link is governed by a set of constraints, as in the case of constraints among graphical elements or between multiple views onto the same data. In other cases, complex domain knowledge may be needed to resolve the link relationship. Some links cannot be resolved without the involvement of a human agent, resulting in user-assisted links. Finally, some links depend on contextual information for their resolution; that is, they behave differently depending upon the context in which they are resolved.

Links may have to perform additional work to establish relationships and to maintain them. Thus, if a link has a remote component as a source or target, then it may have to communicate with that remote component, perhaps via middleware, to create a communication channel for transfer of data. New sources and types of data that are relevant as link sources may become available during the lifetime of a context-sensitive system; it is important, therefore, that such a system be able to discover new resources and either configure its links or create new ones to handle these resources [1].

As shall be described later in this paper, taking this high-level conceptual view of links offers potential advantages in the flexibility of software structures to support them. We are unaware of any systems, apart from the one described here, that provides this level of generality in approach.

3 Paraglide – An Anaesthetist's Clinical Assistant

3.1 Overview

The work reported here has taken place in the context of the Paraglide project which is developing a mobile, wireless context-sensitive system for pre- and post-operative assessments by anaesthetists. The Paraglide system consists of a set of clinical assistants that hold information about current cases requiring assessment, along with associated data. Clinicians use a clinical assistant to collect additional data to record their assessments and to develop plans of drugs and techniques that will be used during the operation.

Clinical assistants communicate with a set of remote services, opportunistically requesting information from these services. This information falls into two general categories: *data* and *task* oriented. The first category covers relevant medical data, such as records of previous anaesthetic records and current laboratory test results. The anaesthetist can view this data and decide whether to incorporate it into their current anaesthetic record. The second category of *task* oriented data covers information that the system can use to help the anaesthetist. Much anaesthetic work is very routine at least for any given anaesthetist. For instance, in some cases only a small set of drugs and techniques are appropriate. The type of operation, along with a few key aspects of the patient's medical record, determine the technique that will be used. The Paraglide system can offer plan templates based on the anaesthetist's previous work that match the current patient's history and the surgical procedure.

3.2 Types of Context & Change Sensitivity in the Domain

There are two basic forms of context- and change-sensitivity that the Paraglide system must be able to handle:

- changes to local data on the clinical assistant
- changes to the accessibility of case-relevant data from Paraglide servers (i.e., the generation of requests to such servers and the subsequent arrival of responses to these requests)[2].

Paraglide clinical assistants operate in a wireless environment with intermittent connections. Additionally, they can operate outside the normal clinical environment (e.g., in the anaesthetist's home) where the nature of the connection may be very different, without access to sensitive data that must remain inside the hospital's LAN. Thus the system must also be sensitive to changes in the connection status.

Paraglide users perform tasks that require changes in the organisation of the activity depending upon the context. Thus, a clinician operating on an emergency case will have very different demands on data than one dealing with general or day surgery. Our system must be responsive to these changes in context.

[2] Sensor-derived data can be modelled as a form of either local or remote data, depending upon how it is captured and communicated.

3.3 Scenarios and Use Cases

A Paraglide clinical assistant usually runs on a small handheld computer to allow anaesthetists to access and enter data on the move. Such systems have a number of input problems. For instance, data entry is often slow and cumbersome. In general the aim is therefore to allow users to select data rather than enter it. The system must therefore be able to predict sensible values for this to work.

When a document, such as a set of blood results, arrives, a title summary is presented in the relevant document interactor. The anaesthetist can open the document in a reader panel and view its contents. They can then choose to paste the details into their current anaesthetic record, or to delete it if not relevant.

Certain elements within the system can be automatically generated. For instance, when the anaesthetist enters the height and weight of the patient, the system can generate the “body mass index” which is used to determine if a patient suffers from obesity.

Mutual dependencies exist between a great deal of the data within the system. These frequently can only be expressed as predictions rather than actual definite changes. For instance, if a patient has a given complaint such as asthma we can predict that they are likely to be on one (possibly more) of a limited set of asthmatic drugs.

Predictions can also depend on remote data. A user could select the operating surgeon from a set of surgeons. This prediction set will change if the staff list changes and a new list is sent out. Anaesthetic plans come from remote servers; the choice of plans is affected by data on the handheld assistant (i.e., the patient’s medical record) and on a remote server (i.e., the set of plans available in a database). Matching can take place externally and then the predicted set of plans can be updated.

There are therefore three dimensions of change that we must consider: predictions vs values, distributed vs local data, and implicit vs explicit update (see figure 1). In general, we wish to handle prediction updates implicitly. The user does not want or need to be informed every time the system changes its set of predictions. They need to find out when they attempt to set a value on which a prediction is based. It is also useful to highlight a field if a prediction changes such that the system believes the new value is not valid. The system can do this in a visible manner without interfering in the current activity of the user. In general, we wish to handle value updates explicitly so that the user knows what is in their system. For instance, when a set of blood results arrives the anaesthetist will look at them. The anaesthetic report acts both as an aide-memoire and a legal record. It would therefore be very unhelpful if there was data in the record that the anaesthetist had not explicitly looked at and affirmed as true. Also there is a need to examine where data came from (to see what caused a value update). There can, however, be times when value updates should be implicit; for instance the calculation of “body mass index”. These implicit updates generally depend on local data changes. It is important to note that these local changes may have been propagated by distributed updates. For instance, the height and weight could come from a document; once those values have been updated, the body mass index will be calculated.

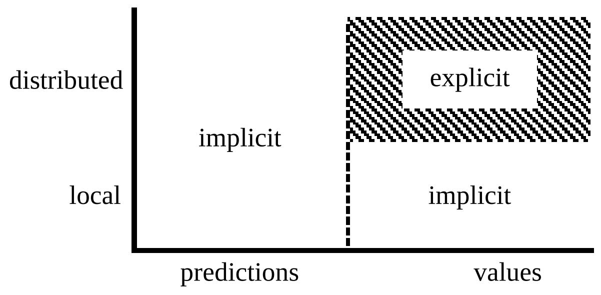

Fig. 1. Dimensions of Change Sensitivity

4 The Paraglide Dynamic Link Architecture

4.1 Overview of the Architecture

A Paraglide Clinical Assistant consists of four components: a set of interactors (user interface), a set of resources managers, a library of documents and a communications subsystem (a broker). The interactors (or widgets) are used to communicate with users and the broker mediates communication with other Paraglide Services. The relationship among the components is shown in Figure 2.

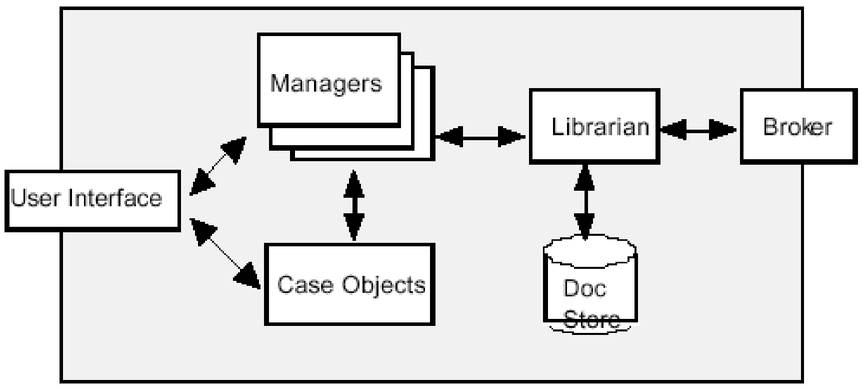

Fig. 2. The Paraglide Clinical Assistant Architecture

Many links depend on domain-related data, e.g., the link between a surgeon's specialty and the surgical procedure for a given case. Such information is held and/or mediated via resource managers. Each manager is responsible for supplying domain-related information to the system as a whole and also for creating links associated with its domain.

Managers may record history about a given topic. For instance, the scene manager can record navigation paths used by the anaesthetist to suggest possible future routes.

They can also preload and provide access to lists of data. For instance, the worker manager can load a set of surgeon records at start up.

Resource managers maintain data used within the system. There are four different types: the scene manager maintains the collection of scenes and handles and records navigation; the case manager produces new cases and provides access to them; the technical environment manager maintains data about the system such as battery power and network connectivity; and domain knowledge managers maintain data about drugs, procedures, staff etc. The case manager produces case objects that represent an object graph maintaining all data about a given case within the system.

We must therefore handle two sorts of links: those that maintain consistency between different elements of a case object; and those that import document data. Our fundamental approach is to try to unify the link framework, so that we can cope with change-sensitivity in a principled way. Therefore all links are viewed as associations that relate a source object to a destination object with respect to an aspect, or operation, via a link function:

```
link = <source, destination, operation, link_function>
```

A link goes from a given source to a given definition, applying some form of link function to transform the data from the source, and then performs some operation on that destination. For instance, a link could go from a document to the blood test results set, extracting all blood results from the document and transforming them into Java objects, before adding them to the blood test results collection.

Because links depend on potentially variable relationships and because they must be created at run-time, we also include link specification as an explicit element in the architecture. A link specification is an object that holds the information necessary to create a link of a specified type, defined in terms of the types of its arguments.

```
linkspec = <source_type, destination_type,

                operation_type, function_type>
```

As we shall see, it is occasionally useful to have multiple sources and frequently useful to have multiple destinations. For example, a pre-operative examination document (a link source) might have information relevant to patient medical history, current medications and other clinical issues, all of which are contained in different parts of the anaesthetic record and that appear in different parts of the user interface (viz., several link destinations). We can therefore think of a link as a *set* of sources, and a *set* of tuples of destination, operation and function.

Furthermore, to enable configurability, link specifications are written in XML. They can therefore be stored in documents and transferred around a network. New links types can be added without the need for recoding. They can, in fact, even be added while the system is running, thus enabling *dynamic* reconfigurability. For instance, if a new document type were to be added in a hospital, an update could be sent out to allow all Personal Clinical Assistants to interpret it, without any need to disrupt the users of the system.

4.2 Link Sources

There are two sorts of link source, document and value sources. It is important to note that these sources are not simple documents or values but sources that will *provide* new documents or values over the duration of the program.

4.2.1 Document Sources

In the Paraglide system, information is transmitted between services as documents, i.e., structured text. These documents may contain information relevant to a number of local links. This information must be extracted from incoming documents in order to resolve the links dependent on that document.

Managers talk to brokers (through the librarian) in order to request documents. These requests are made through the following interface. A manager generates a topic that says what documents to get. This topic specifies which service to go to for the data, and what to say to the service. For instance, it may specify an SQL query to extract data from a database service.

```
public void addDocumentRequest(PgTopic topic,

                               DocumentListener l);
```

The manager provides a document listener that is used to consume relevant documents. The document listener interface contains one method `handleDoc` which consumes a document and performs some action with it. The document is also stored in the document library until released by the consumer.

```
public void handleDoc(Document d);
```

This interface is inspired by, and built on top of the Java Messaging Service (JMS). iBus//MessageServer, the JMS implementation we are using, guarantees document delivery under conditions of intermittent connectivity, supporting the use of wireless connected mobile systems.

It is significant to note that JMS uses a JavaBeans style model for interaction. We provide a listener which is a callback function defining what to do with the given input. This JavaBeans model is the same one used by Java user interface components. This mapping makes it possible to unify information gathered from remote and local sources.

To use a source we must generate a specific request from a given service. We can specify a document source in XML in two parts: a *from* attribute specifying the service source and a *request* attribute specifying the query to be made of the remote service. For instance, the following source specification queries the *LabResults* service for all blood results for a given patient.

```
<PgDocSource

  from="LabResults"

  request="select BloodResults where

           subject={/case/pgSubject/hospitalNumber}"/>
```

Note that the above example highlights two important aspects of link specifications Firstly, we can use a simple path-based syntax for referring to the elements that make up the link, including documents and the attributes of local values (further details of both context paths and document descriptors will be given below). Secondly, the specification can refer to current data within the system, i.e. references that will be resolved at run-time. Here we specify the hospital number of the patient in the current case.

4.2.2 Value Sources

The other link source is a local value within the data structure. For instance, we may have a link between a client and her specialty.

Paraglide uses a general JavaBeans model for all data structures. All mutable objects are *active values*, i.e., we should be able to listen for and react to changes in a value. Paraglide distinguishes two sorts of active value: *Attributes* and *Collections*. Attributes contain simple object values that satisfy the Java Beans PropertyChangeListener interface. We can add a listener to hear about changes to the value.

```
public void addChangeListener(PropertyChangeListener
l);
```

The second form is DynamicCollections. Consider a dynamic list of items. We may not want to hear about the whole change, but only be notified about incremental changes to the collection. We can therefore add listeners to hear about changes such as additions and deletions from a list.

```
public void addListDataListener(ListDataListener l);
```

Based on these two source types we have two forms of source specification. Value sources specify a particular attribute; collection sources specify a collection and an operation (we can do something when an item is either added to or deleted from a collection). Again note that we use context paths to specify a route to a given object in the current case data structure.

```
<PgValSource from="/case/client"/>

<PgCollSource op="add" from="/case/regularMedication"/>
```

A simple model might assume only a single source for any given link. In fact, it is sometimes useful to have multiple sources. For instance, the “body mass index” calculation depends on two sources: the height and weight attributes.[3]

4.3 Link Operations

Once we have a source we require an operation to perform with that source result. There are two general types of operation:

- updates – operations which explicitly update some data structure,
- notifications - operations which notify a data structure about a set of updates.

[3] In practice, we have found only a few instances where the use of multiple sources was really necessary. None of these involve document sources.

4.3.1 Updates

The simplest of the two forms of operation is update: an operation can explicitly update some data. For instance, when the surgeon changes we update the specialty, by setting the value. Again we have a distinction between collection and attribute destinations. With an attribute destination we can set it with a given value. In contrast, with a collection destination we can reset the collection, add or delete one or more items to or from the collection.

4.3.2 Notification

An operation can notify some destination object about a set of changes. Recall that when an anaesthetist receives a new document, such as a set of blood results, it is not immediately integrated into the system. The anaesthetist has the opportunity to review the document, and decide whether it is indeed accurate and relevant. The anaesthetist can then either accept the contents and paste them into the anaesthetic record or reject them, thereby deleting the document.

This form of activity is supported by notification. A notification operation contains a summary function that specifies how to summarise the document. It also contains a set of sub-operations that specify what to do if the notification is accepted. Each of these sub-operations will extract some data from the source. We can generate a summary based on these extracted elements.

For instance, we can have a message that notifies the relevant data structure with a given summary and an operation that adds a new element for each blood investigation to the system.

```
<PgNotify to=...

   summary="Blood Results {/PgBloods/@datimPublished}">

    <PgOp op="add" mode="collection"

          to="/case/bloodInvestigations">

     ...

    </PgOp>

</PgNotify>
```

Note that the summary is a combination of static text and extractions. Here the source is a document so the extraction rule is an XML query. In contrast, if the source were a local value then the query would be a context path query.

What do we do with a notification? We send a *NotifyEvent* to the destination. This provides a summary, and two methods *accept* and *reject*. The summary method provides a *PgSummary* with a title summary (which can, for instance, be viewed in the relevant document list) and a list of child summaries that summarise the data that is extracted from the source for each child operation. The *accept* method accepts all the notification. If some of the child operations fail to work, we throw an exception summarising all the failures. The *reject* method rejects the notification. If this were a notification in response to a document, it would delete the document from the library document store.

```
public interface NotifyEvent {

  public PgSummary summary();

  public void accept() throws Exception;

  public void reject();      }
```

In practice so far we have used notification updates only with document sources. We have generally opted for explicit updates only with distributed data. However, the mechanism is available here if necessary.

While we can have one operation associated with a link function it is more helpful to have multiple operations, both for ease of specification (we can write the source only once) and efficiency (we can add only one listener that does several things, perhaps eliminating some of the common work).

4.4 Link Destinations

4.4.1 Value Destinations

With a value destination we are changing the actual value, such as setting the surgical specialty. As outlined in the previous section there are several possible operations that can be performed on the destination. If the operation is a notify operation then we will simply notify the destination. In this case it must be notifiable object (i.e. able to accept a *NotifyEvent*). Otherwise if the destination is a collection then we will have a collection operation, if an attribute the operation will simply set the value.

```
<PgOp mode="value" op="set" to="/case/specialty">
```

4.4.2 Predictions

Changes to the probability of values in the destination object have been implemented via a *predictions* component. A prediction identifies three subsets of the value-set for a destination object:

- a default
- a likely subset
- the entire value-set.

Although primitive, this provides a potentially useful way of offering alternatives to the user, especially where there is a large set of enumerated alternatives.

More sophisticated prediction models, and more sophisticated ways of utilising the predictive information, are possible. For instance, one can give probability values to destination alternatives. Also, the source of the prediction can be identified where several prediction-changing links are active on a single property.

A prediction update will generally change only the likely subset. We could imagine situations in which the system could attempt to forbid a particular value. For instance, an expert system might predict that a patient could not be on two drugs simultaneously. However, such an approach is very heavy-handed and assumes the system is always accurate. We have chosen a lightweight approach in which updates affect the likely subset by adding or deleting suggestions from it. The operations used here are therefore collection operations.

The default is then simply the most likely element in the likely subset.

```
<PgOp mode="prediction" op="add"

      to="/case/regularMedication">
```

4.5 Link Functions

We may wish to perform some arbitrary transformation on the data before applying the operation. To do this we use link functions. A link function is a function that performs a simple *apply* transform to a piece of data.

```
public interface PgLinkFunction {

 public Object followLink(Object context,Object obj);}
```

It takes a context object (described below) and a value and generates a new value based on this input. For reasons of efficiency, it is important that the *link-function* is a *pure* function. That is, it transforms the data without any side-effecting updates. If applied at any given time in the program to the same value it should therefore return the same result. Given these conditions we can precompile link functions in advance so that the difficult work is done at start-up, not each time the *followLink* function is called.

We support several types of link function, based on the nature of the link. These include property queries, xml queries, maps and ranges, constructors, and predefined functions. A link function can also be a composition of these functions.

4.5.1 XML Queries

The first two forms of link function are both types of query that extract data from the argument value. As we have seen so far there are two sorts of data that we may wish to query: local data and incoming documents.

An extract query is the first type of query. It contains two parts: an actual query and a result type. The result type can be either *collection* or *value*. An arbitrary *XMLQuery* will generally return a set of results. However, sometimes we only wish the first result form a query that we know will return a result. In this case we can use the *value* result type to return only one (i.e., the first) result.

An *XMLQuery* is based on the developing *XQL* query standard[4]. In our initial work we have only used and implemented a subset of this query mechanism. The format of a query is based on a UNIX path structure, consisting of a set of entity names separated by backslashes, e.g., “/PgBloods/PgBlood”. The query can start at the root “/” or within the current context “./”.

[4] http://www.w3c.org/

What is the current context? Remember that we're applying this query to a value. This value may be a document or it may be the result of some earlier query. This happens often in Constructor link-functions, detailed below. The context parameter in the link-function argument provides access to the root document that the source provided. We can precompile all query link functions for a given document source. We can preprocess a document when it arrives, extracting all necessary data, allowing us to parse a document once. This is particularly useful if several link-functions extract the same data.

A query can return one of four results: an attribute or a set of attributes (e.g. return the docId attribute of PgBloods entity is /PgBloods/@docId); the character data residing under an entity (eg return the text string child of /Name/-); an entity (e.g. return the PgBloods entity and attributes /PgBloods); or a whole document subtree (e.g., return the whole PgBloods entity, attributes and children).

The following extract function extracts all the PgBlood subtrees and returns a collection of them.

```
<Extract type = "collection"

         query="/PgBloods/PgBlood#">
```

4.5.2 Property Queries

A property query extracts a value from a local data structure. The JavaBeans model introduced the notion that all values in a bean have a String property name through which they can be accessed. A property query is based on this idea. For instance, the following property link function extracts the specialty field from its argument.

```
<PgProperty value="specialty"/>
```

A property query is in fact a ContextPath. We can apply a context path to a Context object to yield a result. Every object within the Java case object implements this Context interface.

```
public class Context {

 public Object find(ContextPath p);

}
```

A ContextPath is in fact more complex than a simple field name. It does in fact have similarity to an XMLQuery, involving a descent path which is a set of field names separated by a backslash e.g., ./subject/age. The descent path can begin either at the root or at the current value. Context paths into collections require some extra handling. We can specify either the location of an item in a list (e.g. ./1) or a query on the items within a collection (e.g. ./findings/[type='weight']).

4.5.3 Maps

One very useful type of function is a map from keys to values. These occur commonly in prediction links. For instance, consider the case where the procedure prediction depends on the specialty of the surgeon. We might have a map from specialty names to workers. Every time we change the specialty value we look up the new value in the map and generate a new set of likely values.

It would, however, be very tedious to have to specify all of these maps by hand. For instance, we have an XML data file containing data on the list of surgeons. Each entry in the file contains a specialty field. We would like to be able to generate the map from this file. This requirement becomes even more important with medical history data with a number of different dependencies. There are links between *issues* such as asthma, *drugs*, *findings* and *measurements*, and sometimes *operations*. For instance, a coronary bypass operation implies one of a set of serious heart conditions and likely drugs.

Our link functions allow maps to be generated from data files. We specify a file type, which provides access to the data; a *root* query to apply to the file and a set of *from* and *to* queries. Each of these queries is an XMLQuery. The *root* query extracts a set of document objects. The *from* query then extracts the map keys from each object. The *to* query extracts a result type. We may have one or more *from* and *to* queries. For instance, the following link function says: "extract from the PgWorkers xml source, the Worker entities; then generate a map where the keys are the specialty values of the worker entities, and the values are lists of worker entities themselves".

```
<MapExtract file="PgWorkers"

            root="PgWorkers/Worker"

            from="./@specialty" to="./">
```

The use of pure functions is particularly important here. We can precompile the link-function, reading in the data once and then applying all following transformations to the results. This means that when a change actually occurs we perform a simple hash-map lookup, which is cheap to perform.

4.5.4 Ranges

Ranges are very similar to maps. Given a value, we calculate which of a set of non-overlapping ranges it lies within and generate a set of likely results. For instance, if a patients "body mass index" is greater than a given value, they are likely to be obese. We can specify these using ranges. The measurement entity has two important attributes *minValue* and *maxValue*. We can therefore generate a list of ranges from measurement ranges to issues. We can generate a list of ranges and then simply perform a binary search.

```
<MapExtract file="PgIssues"

            root="PgIssues/Issue"

            from="./Measurement"

            to="./"

            op="range">
```

4.5.5 Constructors

A *PgConstructor* object converts data extracted from an XML file to a Java object, taking as parameters the target object's class name and additional parameters as necessary. These additional parameters specify Extract queries. If we're generating the object from an XML source then these will be XML Extract queries. Each of these queries generates one parameter. These queries may have children, i.e., we may have a query that extracts a collection of values, and applies a further constructor to that result.

For instance, consider the following constructor. It generates a *PgInvestigationBlood* object. It extracts the date attribute as its first parameter and then generates a collection of blood results, one for each result value for its second parameter.

```
<PgConstructor ref="PgInvestigationBlood">

<Extract type="value" query="./@date" />

   <Extract type="collection" query="./Result/@value" >

     <PgConstructor ref="PgInvestigationBloodResult"/>

   </Extract>

</PgConstructor>
```

For this all to work, we need a static Java method which generates instances; this resides in the data manager.

```
public static Object getInstance(String name,

                                 Object[] params);
```

4.5.6 Predefined Functions

Sometimes the set of functions defined above is not enough. In this case we can call preprogrammed Java link functions. This is most common for arithmetic calculations. For instance, we can calculate the age based on the date of birth. We define a calcAge method in Java and then call it.

```
<PgPredefined value="calcAge"/>
```

The data manager also provides access to such predefined methods.

```
public static PgLinkFunction getLinkFunction(String
name);
```

5 Conclusions and Future Work

The generic link structure described in this paper is still in its infancy. We have implemented a restricted prototype for the Paraglide system and intend to test it in field trial-based setting in which the links will reflect the anticipated information needs of clinicians.

There remain a number of features that require further development, the most important of which are:

- bidirectional links
- mechanisms to identify and cope with pathological links (e.g., circular link sets)
- tools for specifying links and link functions
- more sophisticated predictions, including predictions from multiple sources.

We also envisage our link architecture offering additional functionality to the application, such as context-sensitive help.

Acknowledgements

This work was supported by EPSRC under the Healthcare Informatics Initiative (Grant GR/M53059). We wish to thank our colleagues on the Paraglide project (Chris Johnson, Gavin Kenny, Martin Gardner and Kevin Cheng) for their feedback on the ideas expressed in this paper and for their contribution to the design of Paraglide system. We also thank the referees for their helpful comments.

References

1. G. Abowd and E. Mynatt. Charting Past, Present and Future Research in Ubiquitous Computing. Transactions on Computer Human Interaction 7,1 (March 2000), 29-58.
2. A. Borning. Thinglab - A Constraint-Oriented Simulation Laboratory. Ph.D. thesis, Stanford University, 1979.
3. L. A. Carr, D. DeRoure, W. Hall and G. Hill. The Distributed Link Service: A Tool or Publishers, Authors and Readers. Proc. 4th International World Wide Web Conference. Pp. 647-656.
4. P.D. Gray and S. Draper. A Unified Concept of Style and its Place in User Interface Design. Proc HCI '96. Springer-Verlag. pp. 49 -62.
5. P. D. Gray, "Correspondence between specification and run-time architecture in a design support tool," in Bulding Interactive Systems: Architectures and Tools, P. D. Gray and R. Took, Eds.: Springer-Verlag, 1992, pp. 133-150.
6. R. D. Hill, "The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications," Proc. CHI '92, 1992.
7. B.A. Myers., et.al. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. IEEE Computer 23, 11 (Nov. 1990), 71-85.
8. Bill Segall, David Arnold, Julian Boot, Michael Henderson and Ted Phelps, Content Based Routing with Elvin4, (To appear) Proceedings AUUG2K, Canberra, Australia, June 2000.

Discussion

J. Hohle: It sounds like you are talking about consistency issues. Have you investigated reason or truth maintenance systems, where you can model information dependencies?
P. Gray / M. Sage: We are trying to build a lightweight system. There are heavy systems that will do this? Our system needs to work on handheld devices. The power of these links is the fact is that they can be connected to local and remote services. Decision support systems are far too heavy for handheld devices. Perhaps you could ìplug inî queries to a remote server which could do further processing.

C. Yellowlees: How do you resolve the desire to employ these high-powered back-end systems with the constraint that the users in this application domain do not have persistent connections to the network.
P. Gray / M. Sage: The resource manager may be adapted to perform predictions locally when no network is available, and query a more powerful prediction tool on the network if it detects that such a resource is available.

F. Paterno: In your application, do you really need a mobile device? Or would it be sufficient to just have a computer in the patientís room connected by a LAN?
P. Gray / M. Sage: The hospital that we are working with is very large, so it is more useful, and cheaper, to have them on their handheld device (which the anesthetists already have and already carry). It is essential for this kind of system, given the number of places where the doctos move (including by the bedside) to always have access to the information, so this is a cheaper, more practical solution.

J. Roth: Are your handheld devices really mobile? Are they actually notebooks or something like a Palm?
P. Gray / M. Sage: We are currently using tablets and are moving to a Compaq IPAQ.

J. Roth: How is the design influenced by using mobile computers?
P. Gray / M. Sage: The information from services in a hospital will be consistent, but the doctor can use the information anywhere in the hospital.

K. Schneider: Given the goal of a dynamic lightweight application, how are you ensuring that the data is reliable and trustworthy? How successful has the application been? Are the doctors able to trust the info? Are they using it? What about the system changing over time?
P. Gray / M. Sage: The full trial has only been running for a month. So far the doctors seem to trust the information. The doctors are using our system and then printing a paper backup later. Also, it is important to note that right now their data is often not very timely (paper records transcribed by a secretary, and not delivered in a timely fashion). It is important that any changes in the system over time are visible to doctors.