A Structural Computing Model for Dynamic Service-Based Systems

Peter King, Marc Nanard, Jocelyne Nanard, Gustavo Rossi

A Structural Computing Model for Dynamic Service-Based Systems

Peter King[1], Marc Nanard[2], Jocelyne Nanard[2], and Gustavo Rossi[3]

[1] Department of Computer Science; University of Manitoba,
Winnipeg; MB; R3T 2N2 Canada
prking@cs.Umanitoba.ca

[2]LIRMM, CNRS/Univ. Montpellier, 161 rue Ada,
34392 Montpellier cedex 5, France
{mnanard,jnanard}@lirmm.fr

[3] LIFIA, Fac. Cs. Exactas- Universidad Nacional de La Plata,
(1900) La Plata, Buenos Aires, Argentina
gustavo@sol.info.unlp.edu.ar

Abstract. Traditional techniques for Programming in the Large, especially Object-Oriented approaches, have been used for a considerable time and with great success in the implementation of service-based information systems. However, the systems for which these techniques have been used are *static*, in that the user-services and the data available to users are fixed by the system, with a strict separation between system and user. Our interest lies in currently emerging *dynamic* systems, where both the data and the services available to users are freely extensible by the users and the strict distinction between system and user no longer exists. We describe why traditional object-oriented approaches are not suitable for modelling such dynamic systems. We discuss a new architectural model, the *Information Unit Hypermedia Model*, *IUHM*, which we have designed for modelling and implementing such dynamic systems. IUHM is based upon the application of structural computing to a hypermedia structure, which thereby operates as a service-based architecture. We discuss the details of this model, and illustrate its features by describing some aspects of a large-scale system which was built by using this architecture.

1 Introduction

An important current trend in system design and development is the consideration of dynamic systems, particularly dynamic service-based systems. In this paper, we present a new architectural model for modelling and implementing such systems. Specifically, we are interested in dynamic systems where users of the system are free to create new data and new data types together with new services for the interrogation and manipulation of such data. Furthermore, the addition of new types and services must not require any explicit changes, upgrades or reorganization in existing parts of

the system; indeed existing types and services should be able to make automatic use of such additional components[1].

Indeed, the existence of these two forms of user extension, new data and new services, serves to characterize the class of system of interest to us. The dynamic nature of both data and functionality within such a system has very considerable effect on the viability of implementation approaches. Whereas traditional techniques for programming in the large, particularly object-oriented approaches, provide substantial support for low- and medium-scale programming, these approaches do not lend themselves as readily to a number of the specific issues arising in such large-scale dynamic systems. The work we describe herein shows how *structural computing* [23] techniques, based on graphical description of relationships between system components, are particularly applicable to the management of large-scale extensibility and tailorability. These techniques provide a means to describe formally the structure of the system and to depict properties of items within the system. Moreover, properties similar to those that have proved useful in object-oriented approaches, such as inheritance, polymorphism, and delegation, are readily described by these structural techniques. Thus, our overall approach to implementing such dynamic systems is a joint one, in which we use traditional object-oriented programming for programming the individual system components "in the small", and use the structural programming based approach to be described in this article to provide an implementation model for the other aspects of a dynamic system.

The systems which we are considering in this article are not *auto-adaptive*, but depend exclusively on user interactions for their extensibility. The systems we have in mind are large and may involve many users.

The architectural model which we describe in this paper is a unified, reflexive one, in which all entities (data, metadata, service, ontology, etc.) are represented in a uniform fashion; each entity is encapsulated as an *Information Unit* (IU), and relationships between entities are denoted by an explicit graph structure, built as a linked network of IUs. This linked network may be viewed as a hypermedia structure, with manipulations taking the form of the application of structural computing to this hypermedia structure, which thereby operates as a service-based architecture. This approach enables us to apply well-known meta-level programming techniques in order to reason on the system structure (the meta-level), just as we reason on the system data (the base level). Furthermore, the IU maintains a distinction between *structure* and *semantics* in the manipulation of an entity. As we explain in what follows, each IU contains a number of links (pointers to other IUs) and in particular, an IU has a *type* link and a *role* link, which are both dynamic and which correspond, respectively, to the structure and the semantics of the entity represented in the information unit in question.

The infrastructure discussed in this paper describes, naturally, a conceptual rather than a physical architecture, that is to say the infrastructure says nothing about the physical locations of the actual software elements. Nevertheless, the approach has been used in practice, in the implementation of the OPALES system, designed and implemented for INA, (the National Institute for Audiovisual Archives in Paris).

---

1 We use the term tailorability to describe this aspect of the system.

Opales services implementation contains some 80,000 lines of Java code, and the same amount of C++ code [4], [20]. OPALES provides a portal to a set of diverse open digital library services, and is designed for the cooperative manipulation of shared multimedia documents among multiple users and user-groups. In particular, such documents may be enriched by multiple *annotation* and *indexation* structures through private and public workspaces. The techniques we describe are intended for dynamic systems such as OPALES; however, they also work well where there is a limited degree of extensibility – where, for example, new upwards compatible, functionality may be added in a system-controlled fashion.

The remainder of this paper is organized as follows. In section 2 we further describe the general context of our work, and we discuss the design rationale for the architecture to be described. Section 3 briefly describes the *Information Unit Hypermedia Model*, our infrastructure for modelling dynamic systems. Subsequent sections go into further detail of how this infrastructure meets the requirements described in section 2. Thus section 4 discusses how the IUHM model may be used for modelling and implementing a dynamic system, We introduce and describe in detail the concepts of *role* and *type*, and discuss how these concepts lead to a simple resolution of the question of *interoperability* between system modules. Section 4 also discusses the notion of *reflexivity*, and describes in detail the *information unit* which is at the core of our infrastructure. Section 4 also explains the dynamic mechanism used to run the *pattern matching* in the implementation of our model. In section 5, we briefly discuss a number of important aspects of related work, and section 6 concludes.

2 General Context

In this section we consider the general context on which our work is based. In particular we discuss the essential differences between what we term *static* and *dynamic* systems, for it is these differences which provide the motivation for the architectural design which we will discuss in later sections.

2.1 Static Systems -- Description

By the term *static* system we refer to a system in which there is a fixed number of pre-defined services available to users of the system, that is the set of allowable user operations and the available data (and data types) are both pre-determined. In such a system, there is a clear distinction between the system developers, and the system users. Users are permitted to access particular services which access and/or modify data in well-defined ways. The use of such systems is widespread, and such systems include banking systems, travel reservation systems, university on-line registration systems, and so forth. In such systems data security is of primary importance, and in addition, therefore, to careful user authentication, such systems expressly exclude any

facility whereby a general user could create a new service, since such a service could access data in a non-authorized fashion.

It should be noted that the distinction between user and developer is more usually a distinction between classes or levels of user: thus one may in particular have a class of super-user, who have responsibility for system maintenance, including updating existing services and the creation of new services. Generally speaking, making such new services available to other general users takes the form of a new upwards-compatible system release.

2.2 Static Systems -- Implementation

Such static systems lend themselves well to traditional *programming in the large* implementation approaches [5]. In particular, the services provided in such a system are fixed, and are usually classified into a number of distinct categories. Within each category, the available services are to a large extent hierarchical in nature, and thus the traditional object-oriented approach is an appropriate one.

2.3 Dynamic Systems -- Description

By way of contrast, a *dynamic* service-based system is one in which developers are free to add new classes of data and new services to access such data at any time, such as in [9], without the need to suspend, to reorganize or to re-release an entire system. Moreover, end-users may also restructure the system architecture to tailor the system to their own needs, and may create new services as compositions of existing services; such user level adaptation may imply quite deep component restructuring. Indeed, user operations of this sort are the intent of a dynamic system. Such large-scale changes imply that provision for automatic reorganization of computing within the system is one of the primary requirements of the system architecture.

Dynamic systems are less common, and thus we provide some examples of user activity by describing some typical user-induced system extensions. We take our illustrations from the OPALES digital video library system. We give a more detailed description of how our model handles such systems in a later section.

2.3.1 Developer-based extension

OPALES provides several tools for indexing and retrieving data, including queries based on descriptors, on keywords, on text similarity, or conceptual graphs and so on. Let us suppose we also want to support the Conceptual Vectors [13] querying technique. The data type and its associated set of tools are first implemented in the small in Java, say, in an IUHM [21] compliant manner. This new type of data and tools are then added into OPALES simply by setting, in a formal manner, the relationships between this new data-type and existing types by means of links between IUs. In this manner, any existing service concerned with indexing tools or data has

automatic access to the new components. For example, the general querying service, which has the ability to combine expressions given according to different formalisms so as to build queries, will then automatically support also the Conceptual Vector technique, without any further updating. Similarly, annotating and indexing tools will gain access to the associated new editor.

2.3.2 End-user based structuring

OPALES provides private and sharable workspaces to its users. A user can build and organize a workspace and dedicate it to a specific *workgroup*. For instance, suppose that the user is an ethnologist, and wishes to annotate a video as a member of this interest group. The role of annotations created on the behalf on a given group is called the *viewpoint* of this group. The user may define work-rules within the workspace and restrict the data created by users of this group to be handled only by a set of specific tools, according to specific rules. Such a restriction is achieved by simply specifying the appropriate relationships between the data-type defining the interest group in question and the other items of the system. Any service in the system then automatically, configures accordingly, thus providing for the extensions, restriction and reconfigurations appropriate to this user group.

These examples taken from the OPALES system are by no means exclusive, and are cited to illustrate the types of user service which might be found in a dynamic system and which our architecture is designed to support.

2.4 Dynamic Systems -- Implementation

The evolutionary nature of service and data creation within such a dynamic system means that a traditional object-oriented approach will not be sufficient, for a number of reasons.

- The creation of services is unpredictable, and there is no *a priori* reason to suppose that the set of services comprising the system at any stage will form a set of hierarchies, such as one typically finds in a static system.
- Moreover, in the case of a system which is dynamically evolving, and in which dynamic evolution is of the essence, the functional view which one has of each individual component does not lead to an overall view of the system as a whole.
- Indeed, users may create quite diverse services, so that the very concept of "the system as a whole" as a discernible item is lacking.
- Further, the creation of new services must be implemented in an evolutionary manner, without the need to modify the remainder of the system, or the need to resort to discrete versioning.

In the following sections we discuss an architectural model designed to implement such dynamic systems, and therefore to meet the various points just listed.

3 The Information Unit Hypermedia Model

The Information Unit Hypermedia Model, IUHM, is fully described in [21], and here we briefly review the model to the extent needed for the purposes of this paper. We observe that in IUHM, system construction corresponds to the specification of a network, and that the primary idea of IUHM is to provide a graph-based description of relationships between *Information Units* which encapsulate any entity (data, metadata, services), so that structural computing techniques can be applied. The type-based hypermedia structure we have chosen for IUHM induces a generalized typing mechanism on objects which is far richer than the classical class inheritance graph of object-oriented classes. Changing a link in the structure has an impact on the actual type of any object. As we discuss in the following paragraph, the originality of the IUHM Model is that each tool may set its own type matching rules, enabling a late binding which relies on the structure of the actual IU graph.

3.1 Design Rationale

The important notion of *type matching* in the context of programming in the small is well-known. Notions such as classes, polymorphism, inheritance and so on have proven their efficacy in object-oriented programming. Programming in the large with dynamic binding of services and data, and composition of services requires a distinct paradigm which is suited to the specification of the rules which apply when data are assigned to services and the specification of how services cooperate. This section informally introduces the fundamental notions on which IUHM is based; these notions are developed in detail in the next paragraphs.

We introduce two new notions: *surroundings* and *affinity*.

- The *surroundings* of an item characterizes the relationships between that item and the others in the system. In contrast to data types, surroundings is not local to an item but is affected by structural changes which occur around an item. Surroundings characterizes not simply the data but all the relationships between one IU and others, and thus the surroundings of an IU potentially includes other IUs which are quite distant in the graph structure. The notion of surroundings is significant in that it provides a means to trigger or inhibit actions on data from other arbitrary items without knowledge of concerned items in question.
- As in the social world, *affinity* depends upon surroundings. The affinity of a service refers to the kind of surroundings that must have the items it is willing to process. A service can define its affinity, and the affinity rules determine which properties of the surroundings are appropriate in a particular instance. The notion of affinity thus introduces a generalization of type matching, which encompasses but which is far richer and has a higher expressive power than the type matching to be found in programming languages.

3.2 Information Units

IUHM has its origins in a hypertext model and is the result of a long evolution and enrichment of our work on typed links hypertext systems [19]. IUHM represents information in the form of a hypertext with typed links and typed nodes, in which nodes *encapsulate* data within a surroundings which is the hypertext network itself, and on which structural computing is used to compute actual affinities. An original aspect of IUHM, and one which demonstrates a fundamental difference from object-oriented approaches, is that services and data are fully unified, that is all nodes encapsulate data; that data in question may in particular be code, depending only on its surroundings. We refer to this node as an *Information Unit, IU*. An IU is connected to other IUs by *links*, which express different types of properties of the surroundings.

IUHM introduces the notion of *role* and makes an explicit distinction between the notion of role and the notion of *type*. The *type* provides information needed to handle the data at low level, whereas *roles* are the high-level actions in which that data is involved. Provided that a given set of types share a given interface (say they inherit from a given type), several IUs of distinct types belonging to this set may share the same role; that is, the same high level actions are possible on it regardless of the underlying low structures. In the IUHM model therefore, each information unit, IU, has two required links, the *type* and the *role*. Thus, every IU is related to at least two other IUs, which represent its type and its role. Fig. 1 illustrates how typed links are used to specify the surroundings of an information unit. More precisely, the type of a UI *a* say is a second IU *b* which encapsulates code capable of handling the data structures of *a*. In this sense, the IUHM *type* is similar to the type notion of programming languages. We will return to this point in section 4.2. The role of an IU *a* is a third IU *c* which encapsulates {something} which deals with a semantics assigned to *a*. An IU may have several roles, thus enabling organizations based upon the semantic level to be set.

Beyond these two mandatory link types, several other links are useful in IUHM, and can be set by the system designer. In the implementation of OPALES[2], for example, considerable use was made of the *owner* link, but this like is not meaningful for all applications. The *inherits* link type has a strong semantics for representing Class like structures. The *relative to* link is a general-purpose link which helps define relationships between IUs. For example, a piece of code, a service *d*, might have a *relative to* link to an IU *e* whose *type* is *affinities*. In this instance, the data describing the affinities of the service *d* would be in the UI *e*, and that the code for handling these descriptions would be in the IU *affinities*. The hypertext structure is dynamic; changing a link (with respect to certain given constraints) may change the

[2] As an aside, we point out that in the OPALES system, as presented in [Nottingham], the hypertext implementation was built on UI descriptors. These descriptors separated data content and links, and gave a special statute to four link types, thereby enabling faster structural computing of affinities. However, this implementation using additional links was based on practical efficiency and is specific to OPALES, rather than an aspect of the IUHM model.

surroundings of an IU, and thereby cause other items to enter or leave the affinity of other services.

A complete presentation of all the possible useful link types is beyond the scope of this paper. We point out, however, that the link mechanism may be used to derive notions found in other programming paradigms, particularly object-oriented paradigms. Notions such as simple or multiple inheritance, delegation, and so forth, may be thought of as sets of relationships, and these relationships may in turn be represented in IUHM by the use of typed links between appropriate IUs. Indeed, since the relationships in question are explicit, it becomes possible in the IUHM model to mix various techniques as required, in contrast to the situation normally found in traditional programming paradigms in which the use of a single technique is frequently enforced. Thus, one may use the type-role mechanism to depict specific inheritance rules and to selectively provide the code to compute inheritance in a given surroundings as needed.

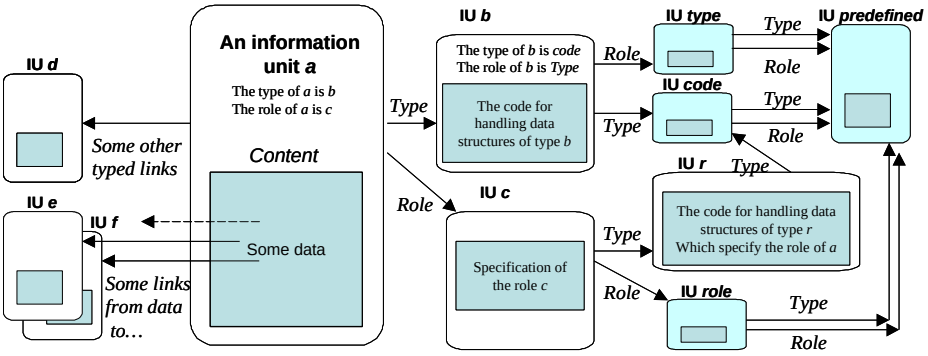

Fig. 1. Using type and role links to specify information units.

4 Architecting a Service-Based System with IUHM

In this section, we discuss how a dynamic service-based system may be modelled and implemented using the IUHM approach. It is important to realise that the complete specification of the behaviour of a system is not simply the content of IU nodes (code and data) but also includes the hypertext network induced by the links which depict relationships between IUs. As a consequence, structural reorganization of the system can in principle be achieved at low cost by editing these links[3]. In practice, this

[3] By way of illustration, in OPALES, surroundings were used to model the concept of *workgroup*. Specifically, in OPALES surroundings are used to represent the fact that certain data is within the concern of some workgroup, and has been validated by the group moderator. Surroundings are further used to set the affinities of the services associated with the workgroup in order to specify which data the group is willing to operate on. Linking a data to the concern of a group, or moving the owner link to, say, the group moderator,

approach requires there to be in place a mechanism, the *IUHM Functional Core*, to dynamically handle structural computing on the IU hypertext network. The IUHM functional core provides the primary mechanisms required in order to run an application described by an IUHM network[4]. To be as simple as possible, the functional core takes advantage of the reflexivity of the IUHM description. This section discusses in greater detail the notions of type, role, surroundings and affinities of IUs and describes the dynamic management of these items in the IUHM core.

4.1 Reflexivity in Type and Role Descriptions

As mentioned in section 3.2 any IU *a* has a type, which is an IU *b* containing the code necessary for handling the content part of the IU *a*, and an analogous remark applies to roles. Reflexivity implies all IUs throughout the system have links to a type IU and a role IU. This type-role network is terminated by a set of *primitive* types and roles, which are directly implemented in the system core. Primitive types and roles are nonetheless represented in the hypertext network, making use of the predefined node called *predefined* (see Fig. 1). In a similar fashion all the primitive notions are represented by predefined nodes, and this includes the nodes *empty* and *undefined*, whose type and roles are themselves *predefined*. This approach ensures that the graph description is consistent with respect to link types: there is no dangling links, rather links pointing to the *undefined* node and there are no missing links, rather links to the *empty* node. This reflexive technique is both quite simple and powerful, and enables replacement and substitution of system components to be implemented by link replacement.

4.2 Why Types, Roles and Affinities?

The distinction between types and roles places emphasis on two distinct and separate aspects of the manipulation of items within the system. The type manages the technical, implementation aspects; the role determines what user-level semantics are attached to the item.

By way of an example from the digital library domain, let us consider a XML file *a*. Technically the document is simply an XML file which would have, in a classical system, the mime type *a.xml*. In IUHM, the IU *a* would have, naturally, a type link to the XML parser which is to be used in the system. The IU *a* also has a role link to a UI *b*, which might indicate, for example, that the IU *a* is an *annotation* of a segment of a movie *c* whose type is MPEG2. The role of this annotation may in turn express the

---

changes the data surroundings, and thereby associates the specific tools dedicated to this user group with the data in question.

[4] In terms of the Dexter Hypertext Reference Model [8], most of the functional core is embedded in the *run-time* layer of the hypertext engine, and the *storage* layer consists mainly of a IU server. The *within-component* layer consists for the most part of services within system components, although a service may be far more complex than, say, a simple component presenter.

viewpoint of an *interest group*, *d*, of ethnologists, and this group may have bound to its description an IU whose role is to set the *work rules* for its members. Further, the IU which contains information about Mrs Smith, say the *group moderator*, may point to the *owner* of this set of work rules, and so on (see Fig. 2). Thus one sees that the rich semantics described by the surroundings of an IU by means of a network of role links goes far beyond the traditional notion of type.

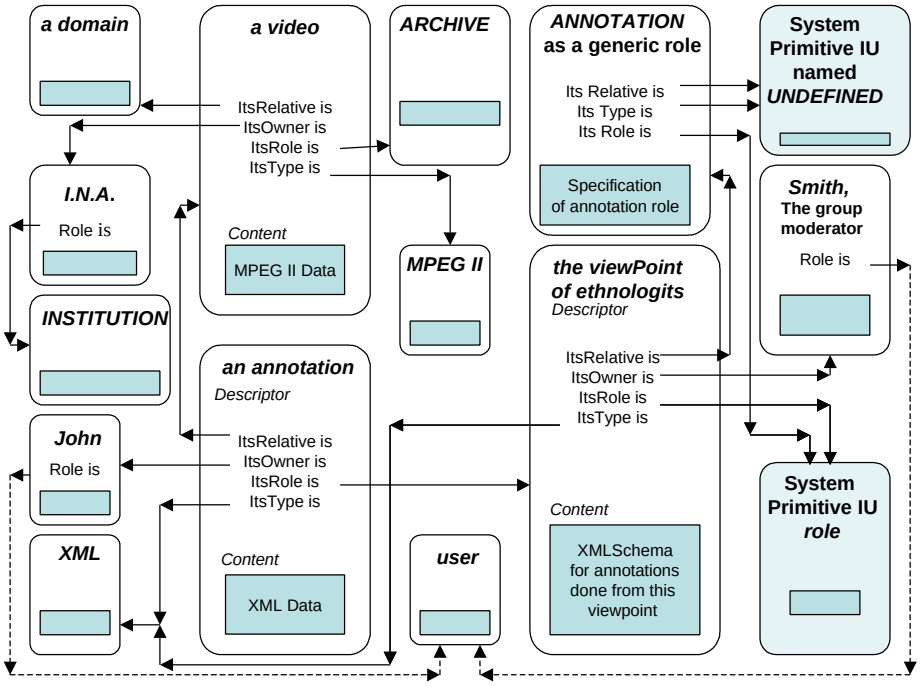

Fig. 2. Surroundings of an IU represented as a hypertext with typed links.

Furthermore, the services which can operate on a given data in a large scale dynamic system are not selected simply on the basis of the data type, which provides information at too low a level, but rather according to the surroundings of the object, which provides the appropriate semantic level. Each service can determine what part of a surroundings is significant, that is, its affinity.

Thus, continuing the previous example, the code which handles annotations is used in various services which have affinities for this code, for instance in a compound service which displays the video segment which is annotated. The annotation service provides the user with a general interface which operates on any annotation (we say that is *belongs* to the annotation role) regardless of the actual type of the annotation. Thus, both an unstructured plain-text annotation and an annotation in the form of a conceptual graph can be handled by this code, because the low level data are processed by its type IU (text or conceptual graphs in this instance, but potentially any annotation type, including XML, conceptual vectors, text, audio etc) whilst the higher level is processed by the code associated with the role.

4.3 IUHM and reuse

The separation of type and role makes possible easier reuse of parts of the system components, since aspects represented by low level types and by higher level roles are clearly separated. Observe that a role is itself an IU and has therefore a type, which contains the code to handle data structures denoting the role semantics. Naturally, both type and role can be hierarchically organized with a *inherit* link. It should be noted that low level system code supporting interoperability of types and roles must be provided, IUHM does not provide any syntactic means for interoperability checking, whereas it can easily support dynamic (run time) checking.

It may be observed that data of different types may share the same role(s) without any need to adapt the role implementation. When several types share the same programming interface, various roles can be built upon this interface. In IUHM there is no need to (re-) implement an interface; rather linking an IU to a compatible role plugs this role into the underlying type. Conversely, new roles can be added by taking advantage of the type-level code, provided two conditions are satisfied:

- a role is constructed using the interface provided by the type,
- the contents of the interface are unchanged when the new type is introduced.

Fig. 3 illustrates the abstraction of a family of IUs with compatible types and roles.

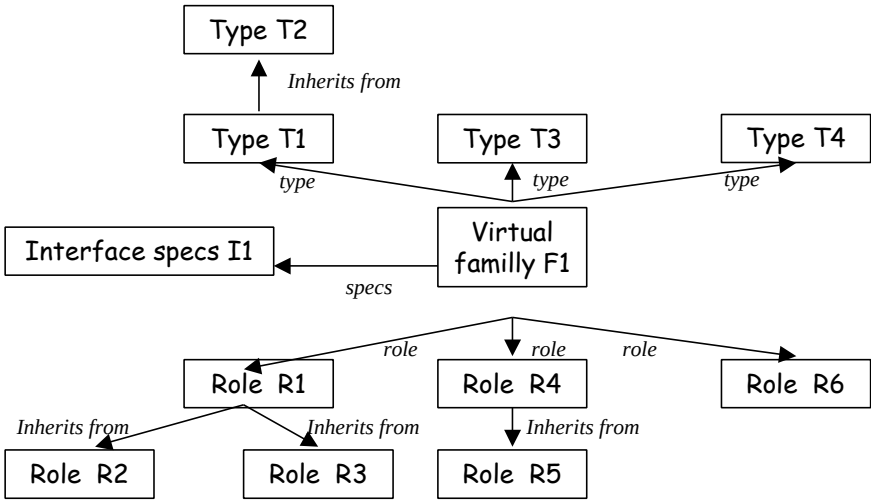

Fig. 3. Reuse of types, roles and interfaces.

The separation of type and role means that the role may be arbitrarily extended by adding new user-level operations which are either defined in terms of the defined type-role interface or which already exist at the role level. Similarly, new compatible types may take advantage of role aspects without reprogramming.

4.4 Affinities and Dynamic Aspects

IUMH manages sets of services, that is applications which cooperate and which interact with users in a given context. We make the assumption that interoperability between low and medium scale services is provided by classical techniques. IUHM only deals with the large-scale cooperation of services. IUHM helps in specifying and exploiting the rules of cooperation between services. Such rules are defined in terms of affinities.

A service is an IU *s* which has a link to an IU *af* whose role is *affinity* . This means that the IU *af* contains specifications which describe the affinities of *s*. We have not mentioned the type of *af* simply because the type of *af* contains the code which is capable of handling the low level description of the affinities of *s*, whereas the role *affinity* enforces the high-level methods provided by this role to comply with the *affinity* semantics predefined in the system. The *AffinityMatch (x)* method delivers a Boolean which indicates whether or not a given IU *x* has surroundings compatible with the affinity rules of the service. Affinities are usually described in terms of graph pattern matching on the hypertext network.

The IUHM functional core registers services and handles the set of affinities of the registered services to dispatch IUs to the appropriate services. Affinities may, arbitrarily, be quite simple, such as unitary direct links such as *type = xml*, or complex, such as *videos annotated by some of the concern of a group moderated by Ms. Smith*. Since affinity handling is managed by the general core mechanisms, there is no predefined syntax or techniques to denote affinities. The predefined core-implemented type for affinities deals only with direct combinations of types and roles, whereas one can bind more specific affinities handlers to a service just by placing a link between the affinities *af* and the type *taf* which computes these specific affinities. The essential point is that the affinities of a service must be able to answer the question as to whether an affinity is or in not interested in managing a given IU.

Another important method provided by an affinity role is *Share*. If a service responds *true* to *Share*, other services whose affinity matches the IU can share the data. For instance, a data inspector service shares the data it inspects with the other services, whilst an editor, on the other hand, may request exclusive access, to avoid contention during editing operations.

In order to deal with conflicts of interest between services, an order is defined on services at loading time. The loading of services is handled by a primary role *service loader*. By default the associated type is predefined (built-in) and would load as service any UI bound to a service role. Alternatively, one may override the default and define a private strategy for loading services by providing a link to a specific code for this type. In this fashion, the IUHM core is extremely simple; when bootstrapping, it looks for *service-loader* IUs in the description and loads these IUs, otherwise it looks for and loads services. Since, by construction, such IUs match the built in service-loader affinity, they are dispatched to the resident service loader core which installs and registers them, collecting their affinities. Initiating these non-resident service loaders induces the loading of other service, in the order and with the strategy specified by these loaders, ordering them as dictated by their affinity rules.

In summary, the notion of affinity provides a very simple reflexive mechanism within the IUHM core to define arbitrarily and to change dynamically the strategies which are used in the system. Since affinities are computed at run time, changes in the hypertext link structure may induce major changes in the system behaviour.

4.5 Affinities and Generic Service Structure

Because of the reflexive nature of IUHM, there is no difference between data and service, both are IUs. As a consequence, one service may be regarded as the data of another service. In this way service affinities can be used to denote subsets of related services (see Fig. 4).

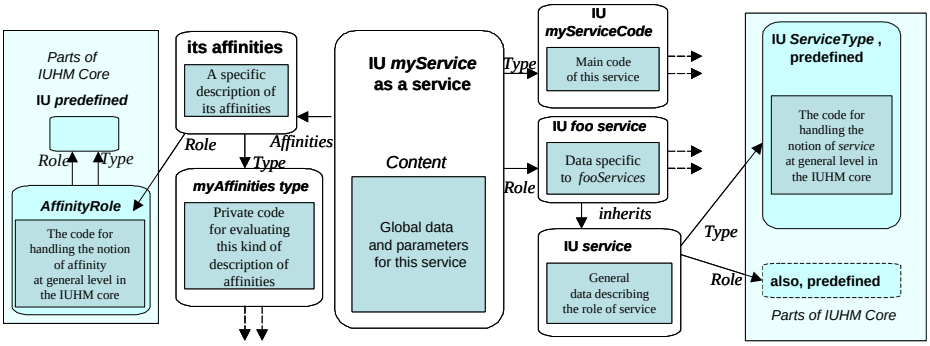

Fig. 4. Surroundings of a service which belongs to the *fooservice* category and which has specific affinities.

This is typically the case of the service loader behaviour, as shown in the previous paragraph, but more interesting features rely on this property. One may easily compose generic services by combining virtual services which are defined solely in terms of affinity. A compound service asks the IUHM core to call a service operating on given data, by passing as parameter the actual data and simply an affinity specifying the kind of service required. In this way the core looks for the services which match the requested affinity and from among these services, selects that which has affinities with the data to de handled. The example described in section 2.3 works in this manner. To add the *Conceptual vectors search engine* into Opales, one just needs to place a role link between this search engine and the role *search engine*. In this way, whenever the generic querying mechanism asks the core which services are in its affinity, it will receive this search engine also. Furthermore, when a compound query contains an IU whose type is *Conceptual vectors*, the generic querying mechanism would simply asks for a *search engine* to open this UI and would receive the *Conceptual vectors search engine* as the target of the core call. It may be observed that this mechanism has similarities with the polymorphism to be found in Object Programming.

5 Discussion and Related Work

In this section we discuss a number of original aspects of our IUHM Structural Computing Model for building dynamic systems. We do so within the context of the MIS Conference, and therefore we focuses our discussion on issues related to meta informatics, and we discuss how our ideas can contribute new concepts for meta development. We organize our discussion around four main points:

- object-oriented development of applications versus structural computing modelling of applications ;
- openness and service-based architecture ;
- open hypermedia architecture and structural computing ;
- scalability, interoperability, reflexivity, flexibility, and adaptation.

5.1 Object-Oriented Development versus Structural Computing

The main idea introduced in this paper is that it is possible to represent an application architecture explicitly by a *computing* structure compliant with the IUHM model. In this structure, any element of a system is represented by an information unit which has a type and a role. Although some similarities can be found between our approach and object-oriented programming, there are two major differences as follows:

- in object-oriented approaches, an object has a class, and methods belong to the class even when the effects of polymorphism and inheritance mean that the method is to be found be elsewhere in the inheritance tree,
- in IUHM an IU
  - has a surroundings which, by nature, depends upon other IUs and thus may change dynamically,
  - is processed by some service which is dynamically selected by the IUHM functional core from among the services which are registered at this instant in the current context, the choice being based upon the affinities declared by this set of services.

Thus the various elements which are responsible for the assignment of an IU to a service are extremely dynamic; the service loader is responsible for the set of active services in a given context, the affinities are responsible for matching data to services, the surroundings are representative of the general over structure of the system.

The main interest of using a hypertext structure to denote these relationships is to offer a very flexible technique to separate clearly the concerns of data, algorithms, and the concerns of system structure (the HTX structure).

Our approach can be thought as an extension of the ideas of Adaptive Object Models-AOM- [2]. Adaptive Object Models provide a way to free the designer from the obligation of creating dozens of classes when it is not necessary. An AOM is basically an instance-based model in which some instances play the role of classes (similar to types in IUHM) and others play the role of base objects. AOMs use the Type Object pattern [10] and the Strategy pattern [7] to provide a way to add behaviors dynamically.

The main difference between AOMs and the IUHM is that while the former is still based on a "traditional" separation between classes and instances, the latter introduces the idea of roles to separate clearly the basic operations of an object (specified in its type) from the semantics of that object from the user's points of view (specified in the role).

In [28] it is shown how to implement roles using a conventional object- oriented language. The Role Object pattern use decorators [7] to "extend" a base class with roles.

The difference between types (or classes) and roles has been widely discussed in the literature. In [12] the authors propose to include roles as first-class citizens in class-based languages. In this proposal, roles permit an object to behave differently when playing different roles.

The OORAM (object-oriented role analysis and modelling) software engineering method [26] proposes to use the concept of roles from the early stages of the software life cycle. While we are not focusing on analysis and modelling, many of the ideas in OORAM can be applied while building IU networks.

Finally, in [28], [29] it is described how to use roles to describe and design composite patterns and object-oriented application framework. The authors introduce the concept of Role Diagrams and show that different class-based implementations can be derived from these diagrams. In this case roles are viewed as higher-level abstractions that allow simplified descriptions of complex object interactions.

5.2 Openness and Service-Based Architecture

From the considerable literature on the subject, it is clear that the construction of open systems has been a topic of great interest for some time. Various techniques have been proposed to deal with different abstraction levels, from hardware levels with techniques such as plug-and-play devices to, more recently, business levels. Furthermore a major constraint is to be able to deploy networked applications. The current trend is to design service-based architectures which enable to separate concerns of services offered through the application and concerns of components involved to offer the various services [32]. Jini network technology [9] is an open software architecture that enables developers to create network-centric services -- whether implemented in hardware or software -- that are highly adaptive to change. Jini technology can be used to build adaptive networks that are scalable, evolvable and flexible, as typically required in dynamic computing environments. Jiny is oriented towards development. The growing movement around web services, the new step in the evolution of the World Wide Web infrastructure, aims at allowing programmable elements to be placed on Web sites where others can access distributed behaviors through published description of services (WSDL) [33]. However, these descriptions do not appear sufficient to elaborate strategic development for business applications. The UDDI registries [30] are used to promote and discover these distributed Web Services, by including explicit description of business models. But from a design point of view, one can observe that there has been an evolution from designing application by decomposition [5], [31], [1], to designing application by

composition [3], [25], or by flow description [14]. One should observe that standards are emerging for describing services offered by distributed components et clearer interface to “plug” them into applications but there is not yet clear support for explicitly modeling both technical conditions of behaviour and semantic conditions of use. Furthermore, models of composition of web services are still as yet the object of reflexion [34]. Whereas IUHM is still in its infancy, it offers both a technical architectural and executable infrastructure for integrating open services, data and metadata and a way to explicit as a separate hypermedia network syntactic and semantic constraints (through types, roles and so forth), thereby providing explicit modelling of application structure and behaviour. Openness and dynamic mechanisms have also been discussed in [21]. In so far as there exists an explicit structure, there is possibility of structural computing for various purposes; we use the term *meta-computing* for this concept.

5.3 Open Hypermedia Architecture and Structural Computing

These computing fields are very representative of growing efforts in a particular domain, hypermedia, to use generic architecture [35] in order to separate the concerns of application modelling and underlying techniques used to manipulate the hypermedia structures of the application [27]. However, as we described in [21], we feel that an approach which adopts distinct models for describing hypermedia structures on the one hand and services on the other are not relevant. The use of such an approach in our view complicates the management of openness and interoperability while maintaining homogeneous semantics [18].

5.4 Scalability, Interoperability, Reflexivity, Flexibility, Adaptation

A primary requirement of architecture capable of describing a dynamically evolving system is that the architecture should embody a simple resolution of the problems of *interoperability* between modules. The term interoperability refers to commonality of access means for services in all domains, and is distinguished from, say, the provision of middleware components specifically related to particular domains, such as one finds in RPC or CORBA. A number of approaches to interoperability are to be found in the literature, including object-oriented approaches [22], 15], [16], layered approaches [17], and aspect oriented programming [11], [6]. While our approach has elements in common with several of these, we have not found any of these existing approaches entirely adequate for our needs. These various approaches appear to be more concerned with applying these notions to implementation, whereas our perspective embodied in IUHM is that an IUHM compliant structure makes it possible to both model and support execution of the application.

We have focused on the notion of interoperability, which is one of the most important quality criteria for software. We observe that IUHM is fully reflexive, thus facilitating adaptation and offering flexibility. Owing to its reflexivity, IUHM also support scalability of modelling and development.

6 Conclusion

The IUHM technology is the consequence of a three-year maturation of the Opales project, in which we had to cope with the continuous evolution and enrichment of the system, and its adaptation to the evolution of user needs. The first version of Opales which has been initially developed using classical programming techniques made us conscious of the need for a flexible architecture for user configurable service based systems. Our long experience in hypermedia systems suggested to us that we should take advantage of typed-links hypertext structure and of structural programming to support the specification, the development, and the evolution of the system in a unified manner. IUHM is the result of the experience gained in this long development. In this paper we have gone beyond the OPALES experience, and have extracted the key elements which may be utilized to depict and organize large-scale service based applications in a generic manner.

Representing relationships between the components of the application structure as a typed-links hypertext graph provides a simple and flexible approach to the description of system composition and of application architecture evolution. IUHM provides a means to handle late binding between any entities in the system, relying on surroundings and affinities, both of which reflect dynamic aspects of the system. IUHM sets a paradigm both for the description and for the dynamic behaviour of the system. We observe that the reflexive architecture of IUHM adds a great deal of flexibility in the design, which enables any of its own mechanisms to be overridden in accordance with the designer's choice. Even the service affinity determination code or the service loader code themselves are handled as services and thereby can be overridden at will, as by editing links in the IUHM description.

Many other techniques, of course, are available to design and implement large scale service-based applications. The major difference between such techniques and IUHM relies on the orderly separation of three major aspects of a system, its technical aspect (types), its functional code (roles), and the relationships between services, data and any notions in the system (IUHM graph). This separation is the key to code re-use and sharing and enables the flexible reorganization of the overall architecture by simply changing the IUHM description.

7 References

[1] Architecture Board MDA Drafting Team. (2001). Model Driven Architecture a technical perspective. Document Number ab/2001-02-01.

[2] AOM. See http://www.adaptiveobjectmodel.com.

[3] Atkinson C., Bayer J., & Muthig D. Component-based product line development. *Software product lines: Experience and research directions*. Edited by Patrick Donohoe. Kluwer Press. ISBN 0-7923-7940-3, 2000.

[4] Betaille, H., Nanard, J., & Nanard, M. OPALES: An Environment for Sharing Knowledge between Experts Working on Multimedia Archives. In *Proc. Conf. Museums and the Web*, Seattle, 2001, 145-154.

[5] Bredemeyer, D. & Malan, R. Software architecture, central concerns, key decisions, 2002, http://www.bredemeyer.com/pdf-files/ArchitectureDefinition.PDF

[6] Constantinides, C.A., Bader, A., Elrad, T.H., Fayed, M. E., & Netinand, P. Designing an Aspect-Oriented Framework in an Object Oriented Environment. *ACM Computing Surveys*. March 2000.

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*, Addison Wesley, Reading, 1995.

[8] Halasz, F. & Schwartz, M. The Dexter Hypertext Reference Model , *NIST Hypertext Standardisation Workshop*, Gaithersburg, 1990, also in *CACM*, Vol. 37 (2), (version without specification in Z), 1994, 30-39.

[9] JINI, see http://www.jini.org

[10] Johnson, R. & Woolf, B. The Type Object Pattern. http://www.ksc.com/article3.htm

[11] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. Aspect-oriented programming. In ECOOP'97, Object-Oriented Programming, 11th European Conference, LNCS 1241, pages 220--242, 1997.

[12] Kristensen, B.K., Osterbye, K. Roles: Conceptual Abstraction Theory and Practical Language Issues. *TAPOS 2(3)*: 143-160, 1996.

[13] Lafourcade, M. Guessing Hierarchies and Symbols for Word Meanings through Hyperonyms and Conceptual Vectors. *In Proc. of OOIS 2002 Workshop*, Montpellier, France, September 2002, Springer, LNCS 2426, 84-93.

[14] Leymann F. (2001). Web Services Flow Language. IBM Software Group. *WSFL 1.0*.

[15] Liu, L. & Pu, C. The distributed interoperable object model and its application to large-scale interoperable database systems. In *Proc. ACM Int'l. Conf. on Information and Knowledge Management*, ACM Press, 1995.

[16] Manola, F. & Heiler, S. An Approach To Interoperable Object Models, In *Proc. of the International Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.

[17] Melnik, S. & Decker, S. A Layered Approach to Information Modeling and Interoperability on the Web. In *Proc. ECDL'00 Workshop on the Semantic Web*, Lisbon, Portugal, Sept 2000.

[18] Millard, D.E. & Davis, H.C. Navigating spaces: the semantics of cross domain interoperability, *Proc. 2nd Int. Workshop on Structural Computing*, Springer-Verlag, LNCS 1903, 2000.

[19] Nanard, J. & Nanard, M. Using types to incorporate knowledge in hypertext. In *Proc. ACM Conf. Hypertext'91*, ACM Press, 1991, 329-344.

[20] Nanard, M. & Nanard, J. Cumulating and Sharing End-Users Knowledge to Improve Video Indexing in a Video Digital Library. In *Proc. ACM / IEEE Joint Conf. on Digital Libraries*, ACM Press, 2001.

[21] Nanard, J., Nanard, M., & King, P. IUHM, a Hypermedia-Based Model for Integrating Open Services, Data and Metadata. In *Proc. ACM Conf. Hypertext 2003*, ACM Press, 2003.

[22] Nguyen, T. Towards Document Type Evolution - An Object-Oriented Approach, *In Proc. of AusWeb'02 Conference*, 2002.

[23] Nürnberg, P.J., Leggett, J.J., & Schneider, E.R. As we should have thought. In *Proc. ACM Conf. Hypertext'97*, ACM Press, 1997, 96-101.

[24] Nürnberg, P.J. & Schraefel, M.C. Relationships among structural computing and other fields. *J. of Network and Computing Application*, special issue on Structural Computing, October 2002.

[25] Oellermann, W.L. Architecting Web Services. A Press. ISBN 1893115585, 2001.

[26] Reenskaug, T. *Working with Objects*. Prentice Hall, 1996.Uniform Description Discovery and Identification: http://www.uddi.org

[27] Reich, S., Wiil, U.K., Nürnberg, P.J., Davis, H.C., Gronbæk, K., Anderson, K.M., Millard, D.E., & Haake, J.M. Addressing interoperability in open hypermedia: the design of the open hypermedia protocol. *The New Review of Hypermedia and Multimedia*, 1999, 207-248.

[28] Riehle, D. Composite Design Patterns. In *Proc. OOPSLA'97*, ACM Press, 218-228, 1997.

[29] Riehle, D. & Gross, T. Role Model Based Framework Design and Integration. In *Proc. OOPSLA 98*, ACM Press, 117-133, 1998.

[30] Uniform Description Discovery and Identification: http://www.uddi.org

[31] UML, see http://www.omg.org/technology/documents/formal/uml.htm.

[32] Van Zyl, J. A perspective on service-based architecture: the evolutionary concept that assists technology providers in dealing with a changing environment. In *Proc. SAICSIT 2002*, 2002.

[33] Web services: see http://www.w3.org/ws

[34] Web Services Choreography working Group: see: http://www.w3.org/ws/choreography group

[35] Wiil, U.K. Toward a proposal for a standard component-based open hypermedia system storage interface. In *Proc. OHS6 and SC2*, LNCS 1903, Springer Verlag, 2000.