Journal of Web Engineering, Vol. 5, No.3 (2006) 216-245

AN ENGINEERING PERSPECTIVE ON STRUCTURAL COMPUTING: DEVELOPING STRUCTURE SERVICES FOR THE WEB

MICHAIL VAITIS

Department of Geography, University of the Aegean

University Hill, GR-811 00 Mytilene, Greece

MANOLIS TZAGARAKIS

Research-Academic Computer Technology Institute

N. Kazantzaki str., University of Patras Campus

GR-265 00 Patras, Greece

GEORGE GKOTSIS

Department of Computer Engineering and Informatics, University of Patras

GR-265 00 Patras, Greece

(vaitis@aegean.gr, gkotsis@ceid.upatras.gr, tzagara@cti.gr)

Received April 24, 2005
Revised May 18, 2006

The emergence of Component-Based Open Hypermedia Systems aims at releasing hypermedia and web applications from the monocracy of link as an information structuring primitive. Instead, an open set of structure servers – each one providing abstractions and semantics relevant to a specific data-organization domain – are employed by an open set of client applications. Nonetheless, the lack of an engineering framework guiding the development and deployment process of structure servers has a part in their limited exploitation. In this paper, we analyze the characteristics of structure servers from an engineering approach, and we propose a software methodology and a set of potential tools in order to direct their development. In addition, we present how this methodology is supported by the *Callimachus* CB-OHS, emphasizing on the tools enabling rapid prototyping of new structure servers.

Key words: Structural computing, service-oriented architecture, hypermedia engineering, rapid prototyping, design patterns
Communicated by: S. Christodoulou

1. Introduction

During the past decade, many researchers have pointed out certain inadequacies concerning the data structuring abstractions used in both web and hypermedia applications [2, 16, 22, 31, 45]. This "structure crisis" mainly originates from the nature and implementation of the notion of link. In the web, links are limited in functionality since they just denote starting-points for unidirectional jumps. Also, they are characterized as "second-class" entities, since they are embedded into the content data – HTML file. In hypermedia systems, although links and anchors are first class entities, they are employed for incarnating all information structuring situations (such as information association, argumentation support, data classification, etc.). Unfortunately, any closed set of abstractions cannot be guaranteed to be useful in a practical sense for all possible data-organization applications [30]. The described

situation raised both convenience and efficiency problems when hypermedia systems are utilized, and lack of standards and interoperability inadequacies.

A significant amount of research and development efforts aiming to overcome the above issues has resulted in releasing structure abstractions and semantics from both data and core system functionality. Structure has been promoted to a first class entity, being provided to third-party client applications on demand, through specific software components and communication protocols. The new generation of *Component-Based Open Hypermedia Systems* (CB-OHS) has emerged, consisting of an underlying set of infrastructure services that support the development and operation of an open set of components (called *structure servers*), providing structure services for specific application domains. The theoretical and practical aspects of this promotion of structure from implicit relationship among data-items to a first-class entity constitute the subject of the newly-established field of *structural computing* [31].

CB-OHS are among the forerunners of a trend for service-oriented computing (SOC); the computing paradigm that utilizes services as fundamental elements for developing applications [36] and relies on a layered service-oriented architecture (SOA). A SOA combines the ability to invoke remote objects and functions (called "services") with tools for dynamic service discovery, placing emphasis on interoperability issues [1]. As both hypermedia applications and the class of web applications categorized as *informational* [14] are content-intensive, the employment of structure services during their development (following the SOC paradigm) would improve efficiency and convenience. Unfortunately, today's developers of hypermedia and web applications do not exploit the facilities offered by CB-OHS [32]. We argue that one of the reasons for this situation is the lack of both an adequate software engineering framework for CB-OHS construction and utilization, and the appropriate tools to support it. This results in ad-hoc development methodologies which produce systems missing certain essential characteristics. The development of a structure server is a complicated task to be repeated from scratch whenever a new structure abstraction has to be supported [55]. In this paper, we analyze structure servers from an engineering point of view, and propose a software methodology involving all aspects of their life cycle. We anticipate that the research and development work in the field of structural computing will be strengthened and that we will be able to draw the attention of researchers of relevant fields to the issue.

The rest of the paper is organized in five more sections. In section 2 we present the field of structural computing and describe the functionality and internal organization of related systems. In section 3 we propose a software methodology aiming to steer all the development and utilization processes of CB-OHSs, while in section 4 we concentrate on the appropriate tools for supporting this framework. In section 5 we describe how this methodology is supported by the *Callimachus* CB-OHS [46, 48], emphasizing on the tools enabling rapid prototyping of new structure servers, while section 6 discusses a number of other structural computing environments. Finally, section 7 concludes the paper and presents future research and development directions.

2. Structural Computing

Traditionally, hypermedia systems were developed to offer linking capabilities among data items in order to produce non-linear information spaces, where the user is able to navigate. Such a system is conceptually composed of three parts: (i) the storage, managing both data items and links among them, (ii) the link engine, offering the linking functionality, and (iii) the application, presenting the informa-

tion space and supporting navigation.[1] Early hypermedia systems are monolithic, since all parts are tightly embedded in a single program (Figure 1a).[2]

During the last two decades, hypermedia systems have been evolved towards the "opening" of their functionality and architecture. The Open Hypermedia movement [35] was originated from the approach to offer linking functionality to any third-party application, properly customized to become hypertext-aware (Figure 1b). Hyperbase systems were developed to abstract the interface between the linking mechanism and the storage (Figure 1c), providing transaction management, access control and other database functionality.

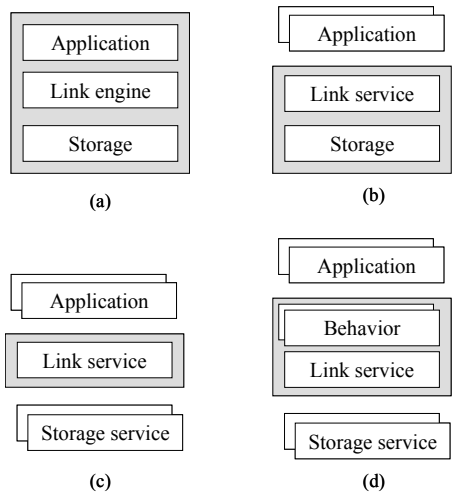

Figure 1: Evolution of Hypermedia Systems

Additionally, new paradigms of information organization patterns have emerged, escaping from Bush's association of information to more elaborate activities, such as hyperfiction authoring and reading, information analysis and classification or argumentation support. Primarily, these structuring tasks were supported by explicit services (behavior), customizing the basic link service of the system (Figure 1d). The *hypertext domain* research field has emerged, focusing on the identification of the different ways the human mind perceives structure in different problem domains. At the same time, *hypermedia system* research is focused on designing and building the computational foundations to support people working with structure, concentrating especially on openness issues.

It has become apparent that the abstractions provided by systems supporting information navigation (mainly, node-anchor-link constructs and follow-link behavior) cannot address issues in new domains (e.g., spatial, taxonomic, argumentation support, etc.) in a *convenient* and *efficient* way [30]. These domains require structure abstractions markedly different from those used to support navigational hypermedia, thus manifesting a gap between hypermedia domain and system research.

Let us take an example; Let us consider the information classification (or taxonomic) domain [37, 38]. Taxonomic reasoning deals with the comparison and classification of highly similar pieces of in-

[1] *Both storage and link engine are integrated into the "storage layer" of the Dexter Hypertext Reference Model [15].*

[2] *Figure 1 is based on a similar figure in [31].*

formation (species, artifacts, etc.) into sets of related items. So, the basic activities that need to be supported are essentially set operations. Three primary components are included in the classification hierarchy: specimen, taxon and taxonomy. A specimen has arbitrary content and attributes, and represents an element of the given data. A taxon has no content but rather arbitrary attributes, which form a constructed descendant. It can have three sides: supertaxa, subtaxa and specimens. A subtaxa can contain an arbitrary number of elements, while a supertaxa or a specimen consists of exactly one element. Finally, a taxonomy contains a hierarchy of specimen and taxa. Although it is possible to implement such an application domain based on the node, anchor and link entities, and simulate set operations with link definitions and traversals, the resulting system would be neither convenient for the user nor efficient executing the operations.

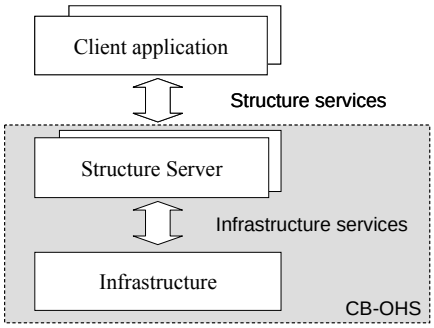

Figure 2: CB-OHS architecture

The need to deliver the tailored support required by different application domains has given birth to CB-OHS (Figure 2). Each structure server supports structure abstractions concerning a well-defined data organization problem, through the provision of structure services to third-party client applications. Structure servers are components of the framework of a CB-OHS, acting as clients to infrastructure services (including generic structure storage, naming, versioning, etc.).

CB-OHSs are the incarnation of a new approach to solving data organization problems, called *structural computing*. This new discipline, driven by the philosophy that structure is more important than data ("primacy of structure over data"), is aiming to shape the theoretical and practical foundations upon which structure (being a "first-class" entity) will eventually become ubiquitous to all computing environments [30]. Structural computing provides general structure-oriented models and services that are able to be adapted to domain-specific abstractions, thus narrowing the gap between hypermedia domain and system research. The provision of dedicated structure services for each domain results in a more convenient and efficient utilization of its abstractions, improving in turn the performance, quality and cost-effectiveness of client applications. This fact is acknowledged as the most important benefit of CB-OHSs, since contemporary systems struggle with the close coupling of the navigational model to their infrastructure.

Other manifestations of structural computing environments are multiple open service systems; that is, systems supporting arbitrary middleware services that can be divided into infrastructure and application services [54]. Structure services are considered as components of the context of SOC, where service reuse and composition constitute a fundamental activity for application development [8, 36]. Therefore, structural computing focuses on the developer's side, aiming to provide tools and services

to assist structure servers and client systems development (belonging to the "B-level" or "C-level" of work).[3]

In Figure 3, the conceptual internal architecture of a CB-OHS is presented. The various entities of both middleware and infrastructure layers are described below, along with the appropriate protocols and interfaces.

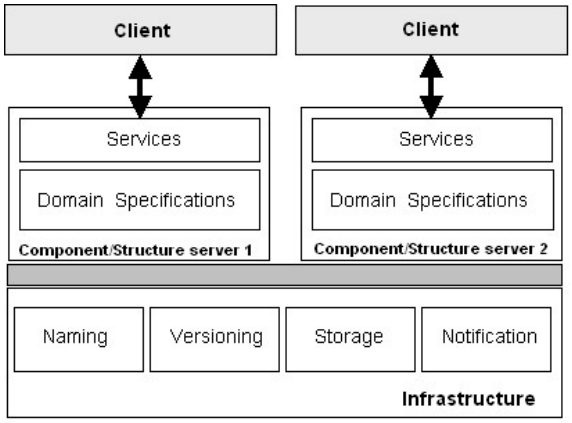

Figure 3: Internal architecture of CB-OHS

Component/Structure server: Instantiates the domain specific abstractions, providing specific structure services to client applications. They are semi-autonomous components, since they rely on the infrastructure services for common functionality. They establish a well-defined interface for communication with client applications.

Domain specifications: Specify the abstractions of the domain in terms of both structure entities (patterns/constructs) and behavior semantics. The structure entities particularize the primitive structure entities supported by the infrastructure, while the behavior semantics model the computational aspects of the domain. These computations can be divided into *internal* operations used mainly for consistency reasons (e.g., to affirm conditions and constraints or to interpret abstractions in a specific manner), and *external* operations invoked by client applications.

Services: Implement the domain-specific external operations. These services are available to client applications through the use of a specific interface (API).

Infrastructure: Includes the fundamental functionality that is available to all structure servers. Persistent storage, naming, event notification control and versioning constitute an essential core of common services. A well-defined protocol is provided for the communication among structure servers and infrastructure services.

Storage: Provides persistent storage services for structure and domain specifications. The storage protocol manipulates primitive (domain neutral) structure entities. It is the responsibility of the structure server to transform (or cast) them to the domain-specific structure entities. Domain specifications

[3] *According to Douglas Engelbart [11], there are three types of work that can be performed in an organization. The A-level is the work of the organization itself. The B-level is the work that develops tools to improve the ability of people performing A-level work, while C-level is the work that develops tools to augment the ability of people performing B-level work.*

are managed separately in a *repository*, in order to support reusability and extensibility of structure patterns among structure servers.

Client application: It is any third-party program that requests structure functionality from one or more structure servers. To utilize structure services, clients may be either custom-build applications or extensions to existing applications. In the latter case, either direct extensions are made or wrapper programs are separately developed.

The presented internal architecture of CB-OHS implies the necessity of a methodology for declaring domain specifications. Although different methodologies are incorporated by different structural computing systems, the resulting benefits are underlined:

- Better understanding of the domain. In conventional systems, domain foundations are hard-coded and informally described.
- The domain specifications could be the framework of a structure server. It may be possible to automatically configure a structure server by setting or modifying structure specifications.
- Exploitation of common structures among different domains, thus enhancing reusability and interoperability.
- Narrowing the gap between hypermedia domain and system research, by providing a common framework to express structural abstractions.

As stated above, the transition from early monolithic hypermedia systems to Open Hypermedia Systems and recently to CB-OHSs has been driven by the vision to provide open structure-oriented functionalities to every concerning application, in a convenient and efficient way. During the last five years, a number of structural computing environments have emerged (e.g., [3, 23, 46, 56]). The layered architecture of CB-OHSs aims to improve the work of both client application engineers and structure server developers, enabling them to utilize high-level abstractions offered by the lower layers. Nonetheless, the lack of an engineering framework guiding the development process counteracts with most of the anticipated benefits of structural computing.

3. Structural Engineering

A hypermedia or web application is differentiated from conventional software products in a number of characteristics, including navigability, provision of search mechanisms, appropriate content organization, aesthetic and cognitive aspects. As pointed out in [24], hypermedia applications "uses associative relationships among information contained within multiple media data for the purpose of facilitating access to, and manipulation of, the information encapsulated by the data", while in [10] a Web Hypermedia Application is defined as "the structuring of an information space in concepts of nodes (chunks of information), links (relations among nodes), anchors, access structures and the delivery of this structure over the Web". In addition, portal-oriented web applications have emerged, providing a single point of access to distributed, heterogeneous sources of information and services [51]. The above applications imply a number of specific activities during their development, such as content acquisition and structuring, navigational and aesthetics design, and multimedia synchronization. The fields of Hypermedia Engineering and Web Engineering have emerged, aiming to provide a systematic (scientific and practical), disciplined, quantifiable approach to the development, operation and maintenance of hypermedia/web applications [14, 24].

The development process of a hypermedia or web application includes a *design phase*, during which issues such as application architecture, content scope, structure, depth, granularity, presentation metaphor, viewpoints and access mechanisms are considered [24]. A number of design models have been proposed in the literature to assist this particular phase (e.g., HDM [13], RMM [18] and OOHDM [43]). What is common in all the aforementioned applications, development processes and design models is the implied support of the navigational domain that results in the utilization of constructs such as node, anchor and link. Since structural computing perceives the navigational domain as just an instance (perhaps the most popular and significant) of an open set of structure services, web applications have the potential to exemplify customized structural abstractions, according to their needs. Consequently, some modifications should be carried out in their development process.

We introduce *structural engineering* as the framework referring to a systematic and disciplined approach to the development, operation and maintenance of structure servers and their usage in the creation of web applications.[4] We argue that the design phase of web applications should follow or comprise a *structure assessment phase*. The purpose of this phase is to analyze the structure abstractions of the application and identify the structure services that have to be used. During the implementation phase, the developer should locate the appropriate structure servers and exploit their protocols. In case of an unsuccessful result, there is an opportunity for the establishment of a new hypermedia domain.

For the definition of a structural engineering framework, the special characteristics of the structure service components should be identified and analyzed. This is the purpose of the next subsection.

3.1 Engineering characteristics of structure servers

So far, [5] is the only work that has addressed a number of engineering requirements for structure servers. We extend that work in order to incorporate recent trends in the field of structural computing, as well as to harmonize CB-OHSs with the web universe.

Structural completeness: The structure services provided should completely solve the structure-oriented problems caused by the application domain.

Specifications evolution: The decision for the construction of a new structure server should be made only when the application needs could not be satisfied by existing services. This requires a deep study of the application domain, so the specifications might not be changed during the development of the structure server.

Size: Structure servers are considered small to medium software projects, since they constitute components in the framework of a CB-OHS.

Distribution and Heterogeneity: Structure servers should operate in the distributed environment of the web, consisting of different hardware and software platforms.

Reusability and Extensibility: Structure services at a fine granularity level constitute building blocks that can be extended or reused during the development of other, more complicated ones.

---

[4] *In the following pages we concentrate on web applications, because of the establishment of the web as the de facto computation and communication infrastructure. However, the essence of our work is also applicable to web-unaware hypermedia applications.*

Life time: The duration of a structure server is long, presuming that the decision for its development is carefully determined. As the software implementation technologies continually evolve, structure servers may need to migrate to different platforms from time to time, while providing a constant interface to third-party applications.

Robustness, Performance, Scalability and Availability: These properties are essential for the proper web-oriented function of structure servers.

Introspection capabilities: Structure servers should be able to communicate their behavior to other applications (i.e., their service interfaces and descriptions, location, and access control details).

Interoperability: Structure servers that provide functionality for the same hypermedia domain should be able to interoperate. An additional requirement is the existence of supporting mechanisms for the transformation of structure abstractions between different hypermedia domains. In this way, structures may be shared among structure servers.

Web integration: Structure servers should be employed by web application developers in a convenient and cost-effective manner. Thus, their interfaces should conform to well-defined and widely accepted standards.

3.2 Life cycle of structure servers

Although the size of a structure server is usually small, a disciplined development methodology is needed, since the demanding characteristics should be met, as presented in the previous subsection. Based on previous experience in the field of structural computing [46, 48, 49], and the conventional software process phases of Specifications, Development (i.e. analysis, design, coding, and integration), Testing, and Evolution/Maintenance [39, 40], we are proposing a life cycle for structure servers (Figure 4), while in the following paragraphs we are describing each one of its phases.

Scenario description

We incorporate the scenario-based specification for Open Hypermedia Systems [35], providing some essential modifications. All functionality proposed to be part of a given structure server should be justified through one or more scenarios of its use; that is, when a proposal that some given structure abstraction should be implemented arises, a scenario based on actual or foreseen use should be mapped out. This policy facilitates discussions among application designers and developers as to better specify the desired structure functionality and avoid "reinventing the wheel". The description of a scenario could include the following paragraphs:

- Goals (name of each goal, plus a description of it),
- Characters (the different kind of users of the services, plus a description for each one),
- Data (some examples of data items that may be associated together with the structure abstractions),
- Requirements for third-party applications (requests),
- Structure configuration (description and constraints among the structure elements),
- Behavior description (operations and propagation of them, synchronization among elements),
- Infrastructure requirements (storage, naming, etc.).

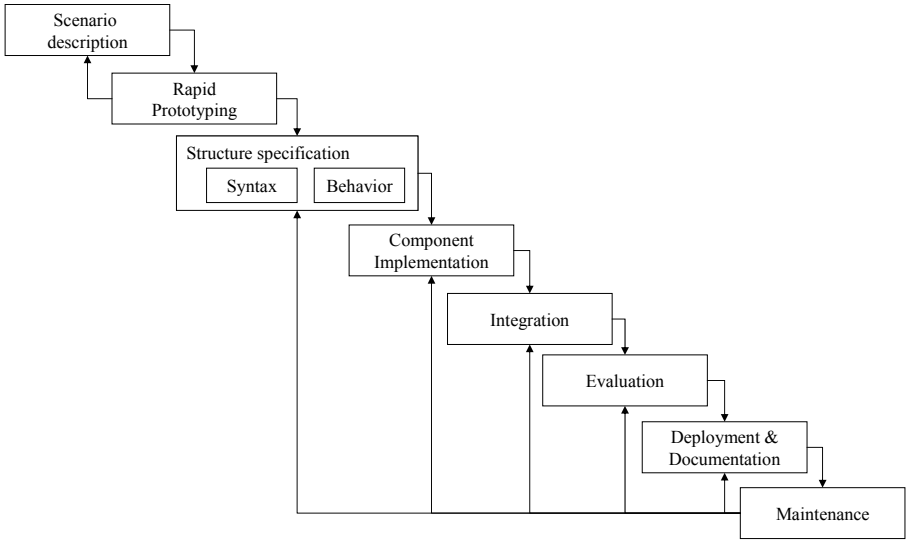

Figure 4: Life cycle of structure servers

Rapid Prototyping

The output of this phase is a prototype structure server that simulates the intended functionality, while not being fully operational. In this way, the evaluation of the scenario is possible and the developer of the application has the opportunity to test the effectiveness and correctness of the desired services. Accrued ambiguities or misunderstandings are clarified and modifications to the scenario paragraphs are made. This testing and backtracking cycle eventually leads to a complete and correct scenario specification.

Syntax specification of structure

In this phase, the *structure elements* of the domain are specified. The purpose of the structure elements are to make concrete the desired structure abstractions and distinguish them from the data abstractions where they usually reside in traditional software applications. In this way, structure is elevated to a first class entity, enabling users, designers and developers to discuss and reason about it. Moreover, during this phase, the properties of the structural elements and the relationships with other structural elements or with data items are determined. In addition, a thorough analysis of already developed components should be carried out to detect relevant elements that may be reused. Methodologies that have been used for the syntax specification of structure include UML (in Construct [56] and Themis [3] structural computing environments), XML (in the Callimachus CB-OHS [46]) or proprietary formalizations (like FOHM [28] or EAD [34]).

Behavior specification of structure

Behavior embodies the computational aspects (or semantics) of a domain; that is, how the structure elements act in order to accomplish the data organization operations of the domain. Computations may be classified in two categories, depending on the calling entities: (i) Services, which are available to clients through the server protocol, and (ii) internal operations, which are used by the structure server internally (e.g., for consistency checking), or when communicating with the infrastructure. In early

structural computing systems, behavior is considered as an "add-on", developed on top of the structure elements. To support the software development task, some structural computing environments provide tools for the design and automated production of the skeleton of the elements' methods – like Construct [56], or *Callimachus* (subsection 5.4). Recent research efforts consider structure, behavior and data in a unified manner, as different views of the same "whole" [23, 33, 49]. For example, in Themis 2.0 [23] the notion of *type* is introduced, enclosing functions, variables and constraints expressing the semantics of a domain, while the structure elements are defined in a *template*. The association of a type with different templates enables the implementation of polymorphic behaviors, served by different structure servers.

Component implementation

Based on the outcomes of the previous phases, the structure server is created during component implementation. Activities that should be carried out are the customization of the primitive (domain neutral) structure objects of the infrastructure, the implementation of the internal operations, the realization of the protocol for communication with the client application, the exploitation of the infrastructure services, and the possible reuse, extension or customization of already existing components. A quite essential task is the implementation of functions that cast the neutral structure objects stored by the infrastructure to the specific structure elements concerning the application domain. In [46], two internal layers are identified in a structure server: The *Abstraction Factory Layer* (AFL), which is responsible for instantiating un-typed structural entities to domain specific abstractions, and the *Abstraction Utilization Layer* (AUL), in which the domain specific abstractions — once created — may be used by clients requesting structuring functionality; see also subsection 5.4.

Integration

During the integration phase, the structure server becomes an active component embedded in the framework of the CB-OHS. Activities to be carried out depend on the concrete implementation-specific characteristics of the CB-OHS, and may include integration within web servers, binding of port numbers, arrangement of authoritative and security issues, etc.

Evaluation

The evaluation activities should primarily ensure properties such as functionality, performance, compatibility, reliability, and usability.

Deployment and Documentation

The deployment activity turns the structure server in operational mode, so that client applications may use its services. A certain prerequisite for this activity is a client's ability to discover and locate required services. The documentation activity includes both the registration of the structure server properties in the dedicated directory services, and the configuration of its introspection capabilities. The aforementioned properties denote all characteristics that distinguish one structure server from the other, including the name of the domain that is served, the protocol used to receive requests and reply, its availability, etc. Both maintaining these characteristics in a special repository within the infrastructure, and selecting an appropriate representation mechanism would ease the development of client applications, since client-side code could be generated automatically. In [20], a Hypermedia Service De-

scription Language is proposed, and provides a wide range of information (such as host and port numbers, interfaces definitions, comments, etc.) that clients can exploit.

Maintenance

As mentioned before, a careful analysis of the provided services will minimize the need for functionality evolution of a structure server. Maintenance is mainly engaged in *corrective*, *adaptive* and *perfective* tasks, aiming to guarantee that structure services are according to the requirements.

4. Tools and Supporting Services

Tools that facilitate the entire development process of structure servers are still limited. In some cases, structure servers are either incorporated tightly into the infrastructure or act as completely autonomous applications. In the following pages we are describing the importance of a number of tools for supporting the development and deployment process of structure servers in the context of CB-OHS. We are distinguishing two categories of tools: *Theoretical tools*, aiming to support structure-oriented problem analysis, and *development tools*, attempting to assist developers who work with a structural computing environment.

4.1 Theoretical tools

As already mentioned, CB-OHSs are an incarnation of structural computing, which suggests a specific view to problems dealing with organization of data. Being not only a technological approach but also a philosophy and school of thought, structural computing requires new ways that will help: (i) examine its own foundations, and (ii) analyze real-world data organization problems in a proper manner. Theoretical tools are needed primarily by analysts and, to a lesser degree, by developers to address two important research fronts, as analyzed below.

1. Structural completeness of services: Since structural computing (and thus CB-OHSs) asserts that it provides a framework able to cover any need when people working with structure, processes are required to examine to what degree structural services cover or solve structural problems. This coverage determines their *structural completeness*.
2. Methodologies for structural analysis and decidability: Structure servers support domains in a very abstract way, meaning that the abstractions provided solve a family of problems. However, real-life data organization problems are rather concrete. Currently, there are no systematic approaches to reduce a real-life organizational problem to a hypermedia domain (or more concisely, to formally determine to which hypermedia domain a given organizational problem belongs). For example, let us consider that a hierarchical security model supports users and groups, which in turn may consist of other groups. Is this problem a special application of the taxonomic domain, and why? Is there a need for the development of new structure services? As structural computing has yet to answer these questions in a systematic manner, methods for structural problem analysis need to be established (i.e., methods being able to compare structure abstractions and semantics to determine their differences) and methods to decide whether a particular organizational problem belongs to a hypermedia domain or not. An organizational problem is *decidable* if there is an effective and systematic procedure (i.e., comprised of finite steps) that solves the problem within the structural computing framework.

4.2 Development tools

Besides the theoretical tools needed by analysts, actual development tools are needed by developers, so to fully exploit the infrastructure services of a CB-OHS. In the following paragraphs the main characteristics of a number of tools of this category are presented:

1. Tools for structure syntax specification: The development of structure servers would be substantially facilitated by the establishment of a specification formalism for structure abstractions. Such formalism should be open to extensions, model-neutral and provide a common ground for cooperation. Although initial attempts of such formalisms exist (e.g., [3, 46, 56]), they do not cover all domain-specific aspects and have not yet been excessively deployed in order for their shortcomings to appear.

2. Tools for structure behavior specification. While tools for structure syntax specification are aiming primarily at syntactical aspects of structure, tools for structure behavior specification are targeting the dynamic and computational aspects (semantics). Based on the structure syntax definition, tools for behavior specification would allow controlling the life-span of structural abstractions, their interaction and their reaction to messages. An initial work in this direction is presented in [49].

3. Tools for structure services discovery. The great amount of potential services provided by CB-OHSs, raises issues about components' usage, location, status and availability [21]. While naming can solve the problem of locating structure servers [47], it can do so only if their names are known. When names of structure servers are unknown, locating them in contemporary CB-OHSs is rather impossible. The latter characterizes the situation of developers that get in touch with CB-OHSs for the first time. In such cases, the existence of supporting services is essential, aiming the publication of structure server's properties and deployment details. The effort of attempting to locate available structure servers without prior knowledge of their existence or their name is referred to as *discovering of structure servers*. Discovering services can take the form of browsers that examine the available structure servers within CB-OHSs and report on their properties. Furthermore, a special service can be included in the infrastructure of CB-OHSs providing the descriptions of APIs that allow the integration of discovery mechanisms into third-party applications. Irrespective of the form these tools may take, special protocols are needed to facilitate discovery. For example, using web services to represent the provided services (in particular, WSDL) would allow the automatic generation of client-side protocol stubs. Furthermore, this approach would also allow the runtime binding of protocol classes into clients, as it is the case on the web.

5. Developing Structure Servers using the *Callimachus* CB-OHS

In this section we outline how the presented software methodology is currently being used in the *Callimachus* CB-OHS [46] in order to develop support for hypertext domains. First, we give a brief description of *Callimachus,* emphasizing the technologies that have been used for its construction. Following this, we focus on the rapid prototyping support for the development of new structure servers.

5.1 Callimachus software components and technologies

Callimachus follows a component-based architecture as depicted in Figure 5. Its primary architectural elements are *client applications*, *structure servers* and *infrastructure*. Client applications can be either

native or third-party applications, such the MS Office Suite and Emacs, or even web servers and web applications. Client applications (clients for short) request services from structure servers using a well defined protocol.[5] The on-the-wire messages sent between clients and servers are encoded using XML and transferred using HTTP tunneling.[6] The adoption of this technique has been imposed mainly by the need to overcome the access restrictions to non WWW services enforced by firewalls. Figure 6 shows a sample message requesting the opening of a node from a navigational structure server in *Callimachus*. HTTP is used as a transport protocol to tunnel client requests. The `Content-Type` parameter specifies the protocol that is being used.

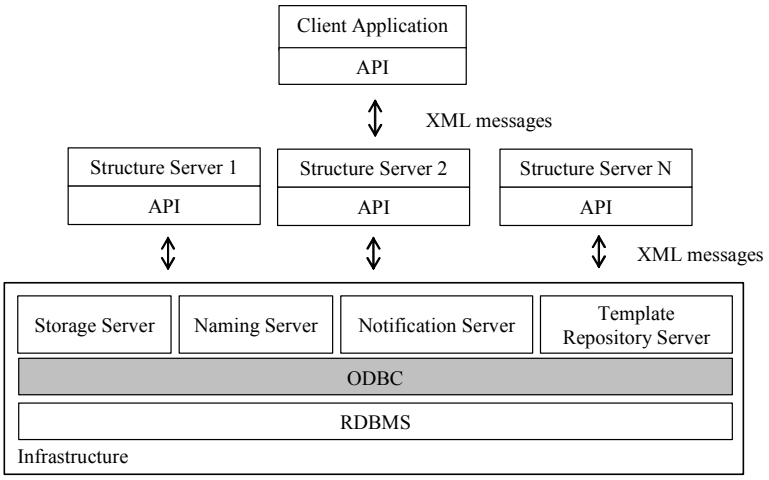

Figure 5: The *Callimachus* architecture

All client-side aspects of the protocol come in the form of a library that implements an API (this library is an essential software module on the client side). Different structure servers require different protocols to communicate with client applications, while this communication is stateless. The construction of the client-side API takes place during the development of the structure server.

The *Callimachus* structure servers have the form of TCP/IP daemons that listen at a specific port for incoming requests. A new thread is spawned for each request. Internally, a structure server has a layered architecture, composed of two core layers: the *server shell* and the *domain model* (Figure 7). The server shell layer deals with the aspects of the structure server as an interface to external entities (clients). The domain model layer deals with the aspects of the structure server as a hypermedia-aware component; that is, it implements and provides the specific structure abstractions of the hypertext domain. All structure abstractions are specializations of a primitive entity called the *Abstract Structural Element (ASE)* [46, 48]. This layered architecture facilitates the rapid development of new structure servers, described in the following paragraphs.

The infrastructure provides those services that are common to all structure servers: storage, naming, notification, and template repository services. Of particular interest to developers is the *template repository service* that maintains a repository for the *templates* of the structure servers. A template is

[5] *With the term protocol we refer to the syntax and semantics of the "on-the-wire" messages exchanged between client applications and structure servers.*

[6] *HTTP tunneling is also known as HTTP encapsulation.*

the formal specification of the structure abstractions of the domain that is served by a structure server. Keeping templates organized in the repository, enables reusability and extensibility of structure abstractions among structure servers (see subsection 5.4).

```
POST /executeOperation HTTP/1.1
Content-Type: NavProtocol v1.2
Content-Length: 540
User-Agent: Callimachus MS-Office plugin v2.4

<?xml version="1.0"?>
<!DOCTYPE np.xml>
<NavProt version=1.2>
    <NPMessageHeader>
           <Host>150.140.18.219</ Host >
           <Agent>Callimachus MS-Office plugin v2.1</Agent >
           <SessionID>0x562AAA2222</SessionID>
           <Operarion>OpenNode</Operation>
           <Request Time>2/3/2003 11:08:52</ Request Time>
     </NPMessageHeader>
     <NPMessageBody>
            <NPOpenNodeRequest>
                  <Node>
                     <NodeName>TestNode</NodeName>
                  </Node >
            </NPOpenNodeRequest>
      </NPMessageBody>
</NavProt>
```

Figure 6: A message (`openNode`) sent from a client to a navigational structure server

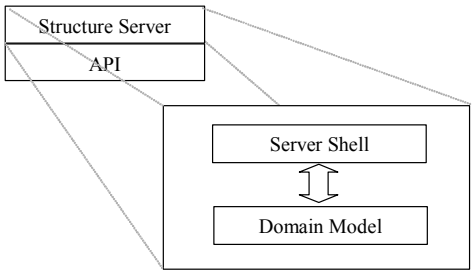

Figure 7: Internal layered architecture of structure servers

The communication between structure servers and the infrastructure follows the same technological design as for the interaction between clients and structure servers. Each infrastructure service is implemented as a TCP/IP daemon that listens to a particular port and requires an exclusive communication protocol. For each service, an API allows access of external entities to the available functionality. Such a separation of services has been adopted so that structure servers will be able to achieve fine-grained integration into structure servers. For example, some structure servers may not have the need for notification services, thus code dealing with notification issues should not be part of them. For permanent storage, a relational database management system is used, where information is stored by each one of the infrastructure services. For example, HRDs[7] of resources [47] are stored in a dedicated database schema and managed by the naming service. Infrastructure services communicate with the RDBMS using ODBC.

[7] *A Hypermedia Resource Descriptor (HRD) captures information regarding how to access a hypermedia resource.*

5.2 Rapid prototyping support

In this subsection, we analyze the rapid prototyping phase of the development of a structure server in *Callimachus*, emphasizing on the involved tasks and tools. As mentioned in section 3, the objective of this phase is to illuminate and verify the scenario describing the application domain. Figure 8 illustrates the tasks performed during the prototyping phase.

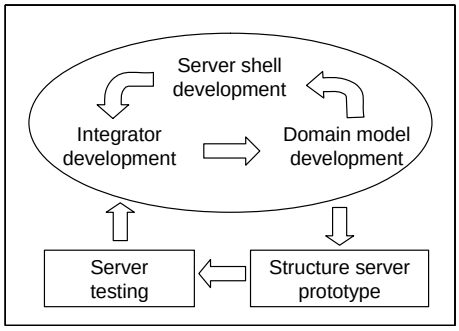

Figure 8: Rapid prototyping of structure servers in *Callimachus*

In Callimachus, a throw-away prototyping approach is avoided being used; instead, an evolutionary prototyping approach has been adopted [25]. This means that the objective is to deliver a working structure server to end-users, rather than throwing the prototype away, and use the knowledge gained to develop the server from scratch.[8] However, the working server provides only the fulfillment of the scenario requirements; issues such as enterprise level robustness, performance, scalability and availability are not addressed.

Design and development is split into tasks, each one dealing with a particular aspect of the structure server. Three main tasks are carried out, each producing a prototype subsystem. The integration of developed subsystems results in a working structure server. The specification, design and implementation of each subsystem do not follow a particular process model, because of their tightly coupled nature and their "small" size as software artifacts. These tasks are described in detail below.

Server shell development: During server shell development, the structure server's interface is built. In this task, the emphasis is on the design of the exact procedure the structure services are invoked. More particularly, all aspects of the structure server when viewed as the receiver of client requests are addressed. Such aspects include listening to, parsing and validating incoming requests, as well as preparing and passing these requests to the domain model for execution.

Domain model development: During this task, the syntactic and behavioral aspects of the domain-specific abstractions (including their relationships) are designed and developed. The syntactic and behavioral specifications originate from the scenario and are defined in terms of the *Callimachus* Abstract Structural Element.

Integrator development: The aim of this task is the development of the necessary software modules that will enable integration of clients with the structure server. These modules come in the form of a client-side API. Specifically, a *wrapper container* and a *communicator* are developed [52] so that client applications are able to request structure services.

[8] *Within Callimachus, developers that build new services or tools based on a structure server are considered end-users as well.*

The prototyping phase starts with the development of an initial domain model prototype. Consequently, the server shell and the integrator prototypes are developed. After an initial cycle, each prototype is refined by constantly iterating through the tasks until an acceptable structure server prototype has been completed. The prototype structure server is tested by end-users aiming to assess its accordance to the scenario.

A number of tools aid developers during structure server design and creation in order to ensure rapid prototyping and in particular short iteration cycles between releases. These tools are crucial, since they attempt to address frequent challenges that hinder developers from delivering working prototypes quickly. These challenges include non-functional and functional aspects of structure servers:

Lack of high level domain abstractions: Callimachus's primary abstraction (the ASE), although powerful in expressing structural abstractions, proves difficult to work with, yielding to error prone structure servers that are difficult to maintain and evolve.

Incremental service (and operation) formalization: During prototyping, the set of the provided services (and operations that users can request) is initially unknown, with their name, behavior and parameters slowly emerging, as prototypes become available for testing. By having services emerging and evolving while development is progressing, the emphasis is on ways to easily integrate new or modify existing services, without requiring changes in functionally unrelated modules of the structure server (which cause major concerns to developers). In particular, the goal here is to achieve localization of the effects during the development of services.

Coping with concurrency issues: As mentioned earlier, in *Callimachus* a new thread is spawn for each incoming request, making a structure server a multi-threaded environment. In such environments, care needs to be taken, when developing access mechanisms to shared resources, such as structural elements. This increases the complexity of the design and imposes new concerns on developers. Currently, such issues are addressed in an *ad hoc* manner for each structure server, making the utilized solutions difficult to maintain and evolve.

Designing for multi-protocol support: Structure servers in *Callimachus* should be able to support multiple protocols for the same domain in order to allow interoperability, to some extent, with existing clients. This would allow, for example, a navigational structure server to support its "native" protocol, and other existing protocols, such as OHP [41]. Towards these directions, developers have to concentrate on the construction and integration of new protocols in a "plug-and-play" fashion.

Smooth evolution of protocol implementations: Although the design of multi-protocol support ensures easy integration of new protocols developed entirely from scratch, it does not address evolution of existing protocols. During protocol evolution, new methods might be added to existing protocol implementations; existing methods might change their signature or might even be associated with different operations at the domain model layer. Such tasks need to be carried out quickly to ensure short iteration cycles.

Designing advanced features: As exemplified by recent hypermedia applications [7, 44], history, logging, and undoing/redoing of user actions, along with supporting transactions, are considered important aspects of hypermedia frameworks. Nevertheless, few systems can actually demonstrate such qualities.

To ensure rapid prototyping and short iteration cycles among releases of the above subsystems, we base their development on design patterns and CASE tools, thus providing the foundation for enabling

extensibility and customization in a systematic way and attaining smooth evolution of structure servers. In the following paragraphs we describe how design patterns and CASE tools are used so that the server shell and the domain model subsystems can be developed. For the development of the integrator subsystem we follow a similar approach as for the server shell, so a detailed description is omitted.

5.3 Structure Server Shell development: The S^3 framework

As mentioned above, the server shell subsystem addresses the issues of the structure server as the interface to external entities; this means that it acts as the mediator between clients issuing requests and the domain model layer receiving and executing these requests.

During the development of the server shell of a new structure server, developers always repeat two inevitable steps: (a) *Parsing and validating incoming requests,* and (b) *preparing and passing the operations* specified by the requests to the domain model subsystem for execution, *and sending the reply* back to the requests' origin.

Parsing and validating incoming requests

During this step, developers need to address the structure server's protocol issues by creating the appropriate *protocol handler* [17]. After a connection with a client is accepted, all received requests for structure services need to be parsed in order to be checked for validity and prepared for execution. Validity checking includes the examination of the conformance of the requesting message to the syntax of the domain protocol specifications, as well as to the semantics of the domain model functions (i.e., the indicated operations along with the type of parameters supplied). Preparing a request for execution refers to the necessary actions dealing with determining the appropriate operation in the domain model that has to be executed. Since different structure servers require different protocols, this step is performed every time a new structure server is developed.

Preparing and passing operations

Once requests have been parsed and validated, they have to be delegated to the domain model subsystem to be executed. In this way, besides reducing development costs, the server shell software leads to fewer errors in structure servers and helps fulfill basic requirements. At this point, developers need to bridge the protocol handler with the domain model. In *Callimachus*, an essential design requirement for the domain model is to be decoupled from the server shell. This approach enables the independent variation of the two subsystems, resulting in a layered architecture in which many challenging tasks can be easily addressed by the developers, such as attainment of a systematic approach to the evolution and extension of domain-specific operations (and protocols), development of operations' logging and undoing, and support for operations' transaction management.

In order to reduce the aforementioned repeated development efforts, we identify and factor out tacit practices by making them explicit in a tool we call the *Structure Server Shell (S^3) framework*. In this way, beside development efforts reduction, the server shell software is less error prone and fulfills the laid requirements. The S^3 framework is a "*semi-finished application that can be specialized to produce custom applications*" [19]. Developers can specialize it and provide the necessary implementations to fill-in the variable parts, thus rapidly constructing server shell prototypes.

The design of the S[3] framework is based on specific design patterns [12] to fulfill the aforementioned requirements. In the next subsection we briefly comment on the design patterns used by the framework and report on how they are used by developers.

5.3.1 Design patterns for the S[3] framework

The design of the S[3] framework makes use of the *strategy* and *prototype* design patterns [12] as well as variations of the *active object* and *command processor* design patterns [9]. The former is used as the foundation to create protocol handlers, while the latter ones are used to passing the operations to the domain model subsystem.[9]

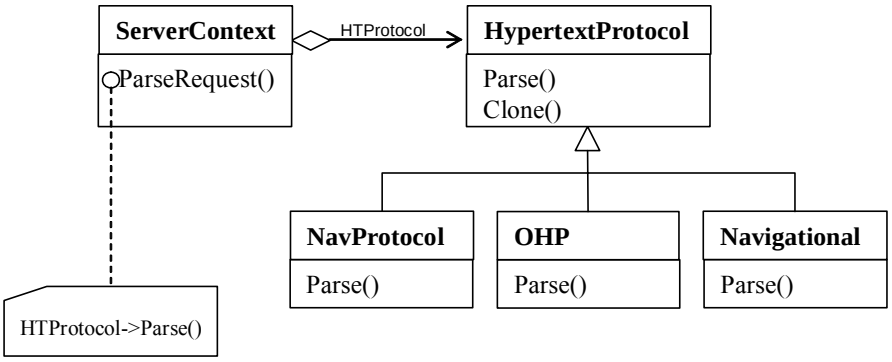

Figure 9: Use of the strategy design pattern

The use of the strategy design pattern is depicted in Figure 9. Within each structure server, the `ServerContext` class deals with all low level aspects of receiving a request from the TCP/IP socket, as well as parsing the HTTP headers of the tunneled request. The class also maintains a reference to an instantiation of the `HypertextProtocol`, an abstract class that is used to parse the received request and supports only the public virtual methods `Parse` and `Clone`. While the `Parse` method encapsulates the suitable algorithm for parsing and preparing incoming requests, the `Clone` method returns a copy of the `HypertextProtocol` instance, used in the context of the prototype design pattern. All protocols supported by a particular structure server, are derived from the `HypertextProtocol` class. Every derived class (that constitutes a protocol handler) implements the method `Parse`, where the appropriate code for parsing, validating and preparing the request is placed by the developer. The appropriate protocol is determined and instantiated during runtime based on HTTP's `Content-Type` parameter (see Figure 6). For this task, the *prototype* design pattern is utilized, determining how the appropriate available protocol implementations are declared and instantiated during runtime (see `hypertextProtocolFactory` in Figure 10). The hypertext protocol factory is part of the `ServerContext` class and is instantiated during initialization of the structure server. There is exactly one `hypertextProtocolFactory` for every structure server. Within the S[3] framework, adding support for new protocols is fairly trivial, allowing developers to focus only on parsing and preparing without spending time about how to integrate the new protocol into the structure server. During design time, developers have to create a class that resembles their protocol implementation (derived from the `HypertextProtocol` class) and to provide the implementation for the

[9] *The bridge pattern could be used as well. However, the strategy pattern has been chosen because of its emphasis on the behavior of individual objects, rather than their interface.*

`Parse` method. Furthermore, they have to register the new class in the factory's `registerProtocol` method that takes place in the factory's constructor. During runtime, correct deployment of the new protocol handler is ensured by the prototype design pattern, since the mechanism of how to determine which class to instantiate is independent of protocol handlers.

```
class hypertextProtocolFactory {
private:
    hash_map<const char *, hypertextProtocol*> htProtocolLibrary;
protected:
public:
    hypertextProtocolFactory (){
         registerProtocol( "NavProtocol", new navProtocol() );
           registerProtocol( "OHP", new OHP() );
           registerProtocol( "Navigational", new Navigational() );
      }
     ~hypertextProtocolFactory(){};

    //Registers a new protocol handler
    int registerProtocol(char *pName, hypertextProtocol *ht);

    // Searches the library and
    // returns the appropriate protocol handler. Calls the Clone
    // method of protocol handler objects
    hypertextProtocol *getProtocol (char *pName);

} // hypertextProtocolFactory
```

Figure 10: Use of the prototype design pattern for the registration and selection of the appropriate protocol at runtime

To address issues related to the task of passing operations to the domain model subsystem for execution, the S^3 framework utilizes a design inspired by the active object and command processor design patterns (Figure 11). The objective of our approach is to separate request invocation from request execution [9].

In the design pattern of Figure 11, all client requests (denoting operations on structure abstractions, such as `openNode` or `traverseLink` in case of a navigational structure server) are instantiated as separate objects. There exists one class for each operation available to clients (e.g., `openNode`, `createLink`, `traverseLink`, `createComposite`, etc.), elevating operations to first class entities, thus allowing them to be stored, scheduled and even undone. Such treatment of operations also allows the support of transactions. In S^3 framework, all available operations are derived from the `DomainOperation` class, an abstract class with two methods: `Execute` and `Undo` (implemented by the concrete derived classes). The `Execute` method of each concrete class executes the operation by calling the appropriate method of the class `HMDomain` that represents the interface to the domain model subsystem.[10] For example, the `openNode` class would call the `openNode` method of class `HMDomain`.

The appropriate concrete operation instances are created by the `HypertextProtocol` class, after having parsed and validated incoming client requests (e.g., see the `Operation` tag in Figure 6). The `HypertextProtocol` class decides which operation to instantiate in order to be flexible with respect to which method of `HMDomain` class to invoke. There might be cases where a matching

[10] *The* `HMDomain` *class can be thought of as an "API" to the domain model subsystem.*

method might not be available in the `HMDomain` class, so an equivalent method (or set of methods) in that class should be invoked. For example, a `getNode` operation [6] (that would be modeled as a separate class) has to invoke the available `openNode` method (i.e., an equivalent method) of the `HMDomain` class, when a `getNode` method is not available. Such choice is conveniently done at the `HypertextProtocol` class after parsing and during preparation of the client request.

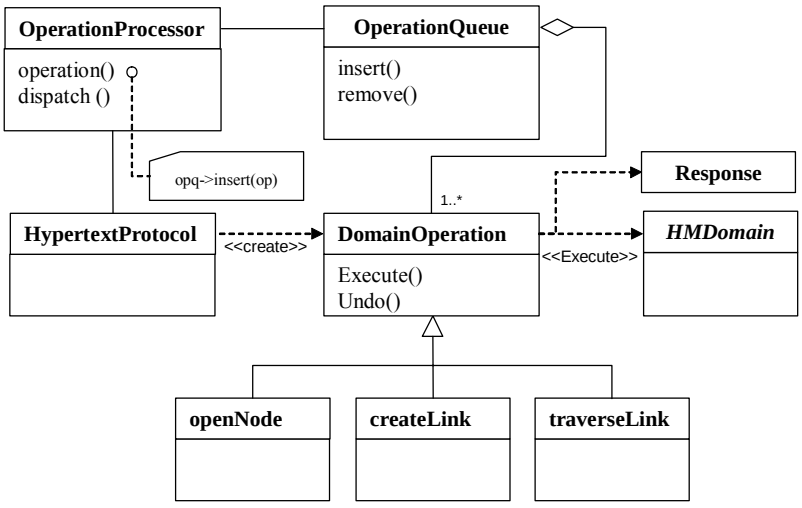

Figure 11: The design pattern for passing operations to the domain model subsystem

The `HypertextProtocol` class enqueues all operation instances by calling the `Operation` method of the `OperationProcessor` class. There is exactly one `OperationProcessor` instance for every structure server. Thus, an `OperationProcessor` constitutes a singleton [12]. The `OperationProcessor` class maintains the operation objects in the `OperationQueue`, and schedules their execution. The `OperationQueue` class may arrange the operations by priority and decide which operation is ready to be executed by calling the operation's `canExecute` method. Operations are dequeued and executed concurrently by calling the appropriate methods of the `HMDomain` class. Each operation executes in a separate thread of control. The output of each operation is available through a specific class (see `Response` class in Figure 11) that is used to sending replies back to clients.

With respect to operation execution, the S^3 framework applies a *thread pool model* [42]. More particularly, while a thread per request model is adopted at the connection stage (meaning that for each request a new handling thread is spawn), only a fixed number of dedicated threads is responsible for executing the operations. Figure 12 depicts the threading model in the S^3 framework. Each thread of the thread pool polls the operation queue for operations that are ready to be executed. A thread in the thread pool gets a new operation for execution by calling the queue's *remove* method. Thus, every operation has its own thread of control permitting concurrent execution of requests.

During structure server prototyping developers can systematically approach the problem of constant change in the domain operations, in the protocol specifications and in their bridging. New operations can be added during design time by extending the `DomainOperation` class and delegating execution to the appropriate domain specific interface method. Since identification and invocation of the operation is provided by the framework at run-time, developers can focus only on semantic aspects

of the operations. In addition, the framework provides the foundation for supporting a number of advanced (but necessary) capabilities, such as the undo/redo operations, as well as transaction management for all structure servers in a uniform manner, thereby reducing maintenance efforts.

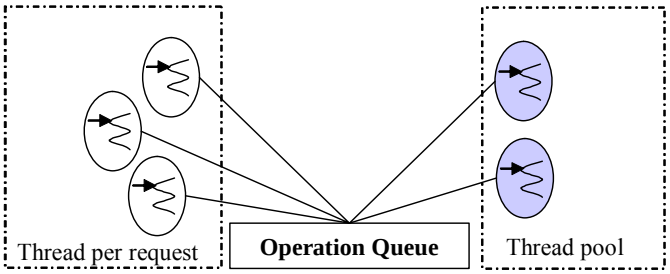

Figure 12: Concurrency models supported in the S^3 framework

Currently, the S^3 framework is available in the form of C++ classes that should be copied into a private workspace in order to be specialized.

5.4 Domain model development

While the S^3 framework addresses development issues related to the structure server's interface to external networked entities, the *Callimachus Template Editor* (*CTE*) tool focuses on the hypermedia-aware aspects of it. The objective of the CTE is to support short iteration cycles during prototyping of the domain model subsystem, by providing an environment that helps the design and implementation of domain-specific structure abstractions (Figure 13). In particular, CTE allows developers to:

- inspect and browse the template repositories of remote Callimachus instances (left pane of the screenshot of Figure 13);
- design, create and update the domain-specific abstractions by creating templates and structure types [46, 48];
- store the templates in a Callimachus instance template repository; and
- generate code that allows the handling of the designed abstractions at runtime. The generated code forms an essential part of the structure server's functional core.

Templates consist of structure types and each structure type capture the semantics of a domain specific abstraction. CTE displays all available templates grouped by the *Callimachus* instance to which they belong (Figure 13). Moreover, CTE allows locating templates across *Callimachus* instances by permitting querying of remote template repositories. Currently, querying is supported against a number of template attributes, such as *template name*, *creator* and *comments*. An initialization file holds the host and port information of the available template repository services. CTE allows the creation of new templates and structure types and for their browsing and update.

Since all structure types are derived and expressed in terms of the Abstract Structural Element, CTE provides the means to tailor ASE in order to "shape" particular structure abstractions. CTE addresses syntactical aspects of structure abstractions, leaving out behavioral characteristics; the latter are handled during the code generation step that follows. For each structure type, the *attributes* and its *endsets* are defined, while endsets can be configured even further. Figure 14 depicts the definition of

the *'Parts'* endset of the *'Composite'* structure type. The description of all properties of templates, structure types, attributes and endsets is out of the scope of this paper and can be found in [46, 48].

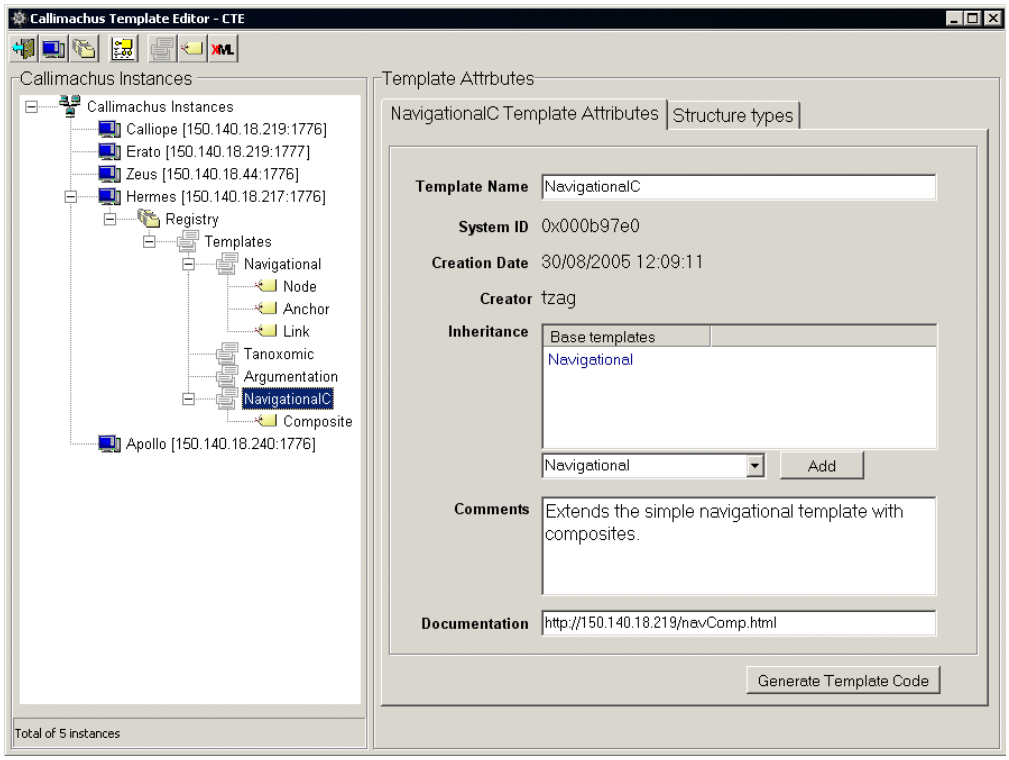

Figure 13: User-interface of the *Callimachus Template Editor*

Abstraction mechanisms are also supported by CTE. In particular, an inheritance mechanism is provided at the template and structure type level. A template can extend one or more other templates, while a structure type can extend only one other structure type. In this way, developers have the ability to build new abstractions based on existing ones, thus permitting reusability and ultimately shortening the development cycle.

Once a template has been defined, it is stored in the appropriate instance's template repository in an XML file (Figure 15). During start-up, structure servers load the templates from the template repository and consider their specifications in order to validate (i.e., allow or prevent) requested operations. For example, the addition of a link to the *Parts* endset of a *Composite* is prohibited, since only nodes may be members of it (Figure 14).

Apart from validating requested operations, templates are utilized for the runtime support of the hypertext domain. In particular, the specifications of the templates and their structure types are employed for the automatic generation of code, handling their runtime aspects. The generated code becomes part of the structure server's core, dealing with the actual domain-specific abstractions (as exemplified with the class `HMDomain` in Figure 11). In this way the required efforts to address the behavioral (computational) aspects of the domain are reduced; the gap between domain-specific and *Callimachus*-required abstractions is filled in.

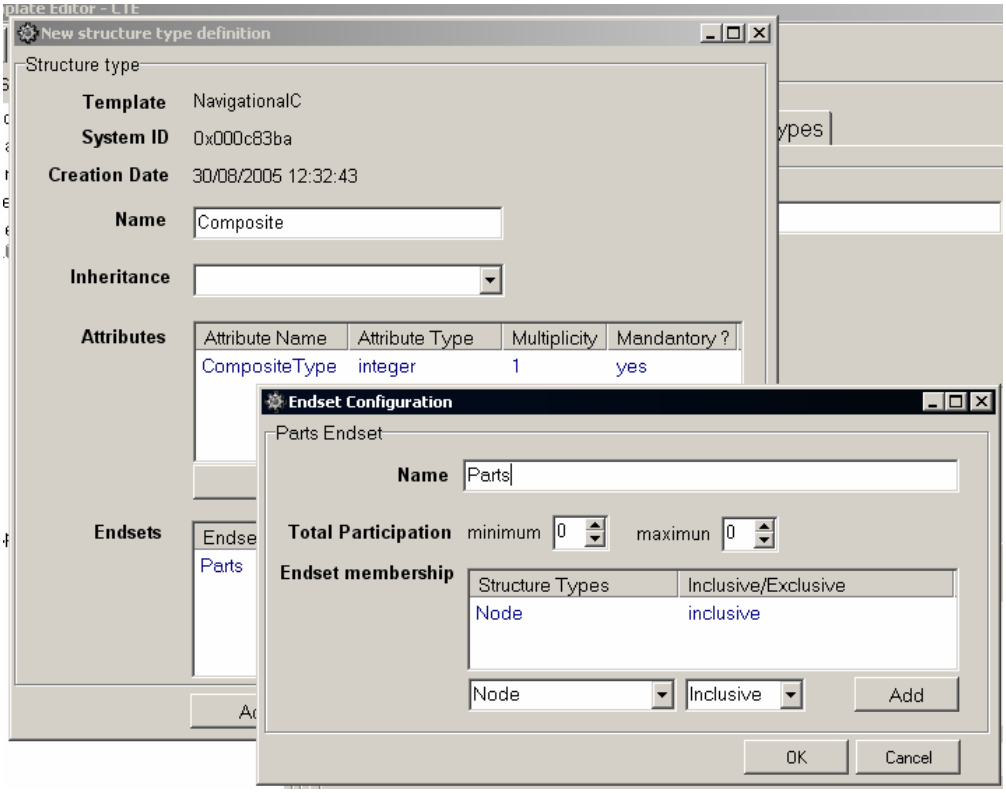

Figure 14: Definition and configuration of an endset (*Parts*) of a structure type (*Composite*)

CTE employs an object-oriented approach to provide the necessary constructs. The code generated reflects a layered process of delivering domain-specific functionality using the ASE. For each structure type, two classes are created, each one corresponding to one of the following two layers of domain-specific abstraction: (1) the *factory layer*, which is responsible for adapting the ASE to meet the constraints of the abstraction (syntactic aspects), and (2) the *utilization layer*, which is responsible for using the tailored ASE (behavioral aspects). In the generated code, the class of the utilization layer extends (via inheritance) the class at the factory layer. Figure 16 illustrates the classes generated for the *Composite* structure type (Figures 13 and 14). The left pane of Figure 16 shows the class corresponding to the utilization layer (high level abstraction with which developers work), while the right pane shows the class corresponding to the factory layer ("shaping" of the high level abstraction using lower level ones).

For both classes, CTE provides the implementation of some of their methods. More specifically, for the class at the factory layer, CTE provides the implementation of the constructor, where the actual configuration of the ASE takes place (see constructor `compositeASE` in Figure 16). For each endset of the structure type, CTE provides the implementations of methods for adding, updating, removing and accessing endset members (see `addParts` and `removeParts` methods in Figure 16), taking care of runtime type conversion.[11] CTE also provides methods for manipulating attributes that are ei-

[11] *A specific naming convention is used for these methods: the name of the method consists of the operation name and the name of the affected endset.*

ther application specific (e.g. compositeType in Figure 14) or inherent to the ASE (such as id, creator, creation date, type name). For the class at the utilization layer, CTE provides the implementation of methods for calling all endset-related methods of the class at the factory layer. Developers can also add any new methods required to express more behavioral aspects. It is the class at the utilization layer that gives developers the illusion of simplicity of working closely to the application domain, by hiding *Callimachus* related lower-level system concerns.

```
<template name="NavigationalC" resolvability="yes" accessibil-
ity="yes">
<uses>
     <template>
       <id>2</id>
     </template>
</uses>
<structure_type id="21" name="Composite" sidedness="1">
     <endset name="Parts">
       <s-set><member type="4" min="0" max="-1" /> </s-set>
     </endset>
</structure_type>
</template>
```

Figure 15: XML output of CTE for the 'composite' structure type

Whenever a template inherits from another template, code is generated for the inherited structure types. Currently, CTE creates code in C++ that developers may copy into their local workspace and tailor as needed.

6. Other Structural Computing Environments

In this section, a number of other existing approaches related to the structural computing field are briefly presented, emphasizing the supported development methodology and their integration with the web.

The *Fundamental Open Hypertext Model* (*FOHM*) [27, 28] offers a single generalized model to express structure types for hypertext domains. It is constructed from four core object types: *Data objects* (wrappers for any piece of data, held outside of the model), *Associations* (relationships among data objects or/and other associations), *References* (pointers to data objects and associations), and *Bindings* (connectors used to attach references to an association). *Context* and *behavior* objects may be attached to any part of a FOHM structure. The former are used to define in which context a FOHM part is visible, while the latter are opaque to the model and are interpreted by the client application. FOHM constitutes the underlying model for a number of structure servers (Real Time Streaming Protocol Server, Auld Leaky link server [26, 27]). The services are provided by autonomous applications, which do not follow the layered CB-OHS architecture. The API for the development of client applications is fully documented, while no other development tools are provided. The communication with client applications is realized with proprietary XML messages over HTTP.

```
class composite: public compositeASE
{
    private:
       int compositeType;
    protected:
    public:
     composite();
     composite(compositeASE &c);
     ~composite();

     Node *getParts(aseID id){
        nodeASE *na=_getParts(id);
       return(new Node(na) );
     };
     int getCompositeType();
     int setCompositeType(int ct);
     int addParts(Node *n);
     int removeParts(Node *n);
      …
  }
```

```
class compositeASE: public ASE
{
  private:
  protected:
   nodeASE *_getParts(aseID id);
   int _addParts(nodeASE *a){
          _addEndsetMember(PARTS,a);
   };
   int _removeParts(nodeASE *n);
   int _inParts(nodeASE *n);

  public:
   compositeASE(){
     setTypeName(COMPOSITE);
     createEndset(PARTS);
     addInclusive(PARTS, NODE);
     setMinParticipation(PARTS,1);
     setMaxParticipation(PARTS,-1);
   };
   ~compositeASE();
   int load();
   int save();
   aseID getID();
   char *getTypeName();
   char *getCreator();
    …
  }
```

Figure 16: Excerpts from generated classes for a structure type designed in CTE

The *Information Unit Hypermedia Model* (IUHM) [29] originated from OPALES, a hypermedia environment for exploring and indexing video archives. IUHM was designed to provide extensibility, openness, and interoperability. It adopts a single hypermedia model based on the navigational paradigm, whereas new services can be expressed and integrated. The fundamental construct of the model is called an *Information Unit* (IU); every system entity (such as data, metadata, service, user-group, ontology) is encapsulated as an Information Unit instance. The structure of an IU is divided into its *descriptor* and its *content*. The descriptor holds four links, named *type*, *role*, *relative* and *owner*, pointing to other IUs. The type link points to the structure syntax of an entity, while the role link points to its semantics (behavior). Relative links express arbitrary directed relationships between a pair of IUs. The owner link assigns the IU access rights. IUHM follows the layered CB-OHS architecture. The

infrastructure consists of an *IU server*, managing IU descriptors, and a number of other content management servers. A *functional core* at the middleware level provides a generic interface for creating, searching, accessing and updating IU descriptors and contents. Developers can make use of this functional core and a set of primitive and basic services, in order to compose their specific services (in turn, defining the properties of a new IU). New services should be registered and are made available dynamically.

The *Themis* structural computing environment [3, 4] consists of a framework interface, a generic structure server, and two extension subsystems, for the definition of structure templates and structure transformations, respectively. The primary structure abstraction is the *Element* abstract class, associated with an open set of attribute-value pairs. The value of an attribute may be another element instance. Element instances belong to either of two subclasses, *Atom* and *Collection*, where the second subclass aims to group together other elements. These simple conceptual constructs can be combined to support a variety of domain-specific structures, both tree-based and non-hierarchical. Application developers can extend the generic structure server by defining templates (using the template subsystem) for the needed domain-specific structures. The template subsystem provides also instantiation operations for structures, which can be manipulated through the framework interface. The transformations subsystem provides supporting operations that automatically transform structure instances from one template to another. This functionality is delivered through custom-developed plug-ins that are loaded on demand into the transformation subsystem. *Themis* utilization in real application development projects (e.g., the InfiniTe Information Integration Environment [4]) has resulted in reducing the amount of code required, along with raising the level of abstraction such that it is easier to be understood and maintained.

The *Construct* structural computing environment [55, 56] is designed to support the development and host of an open set of structure and infrastructure services. The development process consists of using a UML tool to specify the classes (in terms of both state and behavior) that make up a new service. The derived diagram is automatically transformed to an IDL specification, which in turn delivered to the *Construct Service Compiler* (CSC). CSC produces a set of files, including an XML DTD, a skeleton service, and a set of common service behaviors. The skeleton service consists of a set of classes specifying a set of methods having their bodies (semantic parts) empty. The developer has to fill the missing code and load the service to the Construct environment. Generated services are available not only to third-party clients, but also to the development environment itself. The current set of services includes navigational, metadata, taxonomic, spatial, cooperation, and data mining services, while some of them are also provided on the web [53].

All the above structural computing environments have been designed and developed with respect to the basic notion of structural computing, i.e. the primacy of structure over data. For the representation and manipulation of structure, each of them provides a single abstract entity and a set of operations and procedures for its customization to the specific structure syntax and semantics of the application domain. Structure server developers utilize a proprietary API provided by the infrastructure in order to accomplish their tasks, while application developers ought to be familiar with the different protocols and interfaces each structure server provides.

As is evident, a significant amount of work has been invested for the construction and provision of infrastructures to realize the structural computing vision. Issues like process models, development methodologies, integrated development environments, CASE tools, standard service protocols, web

integration, are only partially examined. Although there are still some open questions about the foundation of structural computing, we believe that the maturity of the aforementioned engineering aspects will broaden structure services utilization and acceptance.

7. Conclusions – Future Work

In this paper we have attempted to approach some software engineering issues on the newly-emerged field of structural computing. Despite the advantages in terms of convenience and efficiency that are attained by the isolation of the structure-aware aspects (and their encapsulation in structure services) from the data-aware ones in data-organization applications, structural computing is not widely accepted by the software development community. We believe that the lack of an engineering framework directing the design and implementation of structure servers, along with the immaturity of the tools supporting the "primacy of structure over data" philosophy of structural computing, make the utilization of CB-OHSs cumbersome; developers hesitate to incorporate structure services into their daily tasks.

Based on well established software engineering practices and on the experience gained from previous research work, we propose a software methodology for CB-OHSs, aiming to emphasize on their critical characteristics, rather than exploiting specific technologies. Towards the establishment of proper development tools for structure servers, we have identified the areas in which such tools are mostly needed, including both the theoretical and the practical aspects of structural computing. With regard to the theoretical aspects, new tools to analyze structure problems and evaluate structure abstractions are required, based on the notions of structural completeness and decidability. Regarding the practical aspects, integrated developments environments are needed; these environments should manipulate domain-specific structure syntax and behavior as well as new methods in locating and binding structure services with client applications. We have described how our proposed methodology is supported by the *Callimachus* CB-OHS, emphasizing the tools that enable rapid prototyping of new structure servers. In addition, a number of other contemporary structural computing environments are briefly presented, emphasizing the supported development methodology.

Our future research plans mainly focus on two parallel aspects: on extending the set of tools of *Callimachus* in order to further assist the development of structure servers; and on establishing methodologies and tools to facilitate the convenient integration of structure services into web applications.

The *Callimachus* development framework does not support the specification of the behavioral semantics of a domain; instead the developer should fill–in the body of the methods created by the CTE in order to make the desired behavior happen. Behavior modeling is a significant research issue in the field of structural computing in general. Instead of perceiving behavior as something orthogonal to structure, in [49] we have made some initial attempts to perceive structure as a function transforming input states to output states, according to the semantics of the given domain. An initial stimulus invokes a series of transformations by propagation through the structure elements, until the achievement of the final state. These preliminary efforts should be further extended in order to formalize an algebra or language able to describe both the syntactic and the behavioral aspects of structure as a whole. Based on such formalization, CTE could generate code dealing with behavioral aspects of the domain, thus reducing development efforts.

Finally, some initial systematic attempts have been made for integrating structure services into web applications. In [21] a step-by-step methodology is presented for the provision of structure ser-

vices on the web, based on the *web services* technology [50]. An experiment has successfully been carried out, making available the Babylon taxonomic structure server through the SOAP protocol. However, a great amount of effort from the developer is required (especially for generating the appropriate code on the web application side) to integrate existing structure services into web applications; while at the same time there is no support for structure service's evolution. During structure server evolution, web applications (being clients of structure servers) need to evolve as well. This makes web applications ad-hoc, error-prone and difficult to maintain. Currently, we are concentrating on a prototyping methodology for the integration of structure services into web applications, and on the support of their seamless evolution through an appropriate set of tools.

References

1. Agrawal, R., Bayardo, R. Jr., Gruhl, D., Papadimitriou, S., *Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications*, in Proceedings of the 10th Int'l Conference on World Wide Web (WWW '01, Hong Kong, Hong Kong), 2001, pp. 355–365.
2. Anderson, K. M., *Integrating Open Hypermedia Systems with the World Wide Web*, in Proceedings of the 8th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '97, Southampton, UK), 1997, pp. 157–166.
3. Anderson, K. M., Sherba, S. A., Lepthien, W. V., Structural Templates and Transformations: The Themis Structural Computing Environment, *Journal of Network and Computer Applications*, 26(1), January 2003, pp. 47–71.
4. Anderson, K. M., Sherba, S. A., Lepthien, W. V., *Structure and Behavior Awareness in Themis*, in Proceedings of the 14th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '03, Nottingham, UK), 2003, pp. 138–147.
5. Anderson, K. M., *Software Engineering Requirements for Structural Computing*, in Proceedings of the 1st Int'l Workshop on Structural Computing (SC1, Darmstadt, Germany), Technical Report AUE-CS-99-04, Aalborg University Esbjerg, Computer Science Department, Denmark, 1999, pp. 22–26.
6. Anderson, K. M., Taylor, R. N., Whitehead, E. J. Jr., A Critique of the Open Hypermedia Protocol, *Journal of Digital Information (JoDI)*, 1(2), 1997.
7. Atzenbeck, C., Nürnberg, P. J., *Constraints in Spatial Structures*, in Proceedings of the 16th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '05, Salzburg, Austria), 2005, pp. 63–65.
8. Beringer, D., Melloul, L., Wiederhold, G., *A Reuse and Composition Protocol for Services*, in Proceedings of Symposium on Software Reusability (SSR'99, Los Angeles, California, USA), 1999, pp. 54–61.
9. Buschmann, F., Meunir, R., Rohnert, H., Sommerland, P., Stal, M., *Pattern Oriented Software Architectures: A System of Patterns*, John Wiley & Sons, 1996.
10. Christodoulou, S., Zafiris, P., Papatheodorou, T. S., Web Engineering: The Developers' View and a Practitioner's Approach, *Web Engineering, Software Engineering and Web Application Development*, Springer-Verlag LNCS 2016, 2001, pp.170–187.
11. Engelbart, D., Keynote talk, 4th Int'l Workshop on Open Hypermedia Systems (OHS4, Pittsburgh, PA, USA), 1998.
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
13. Garzotto, F., Paolini, P., Schwabe, D., HDM – A Model-Based Approach to Hypertext Application Design, *ACM Transactions on Information Systems*, 11(1), 1993, pp. 1–26.
14. Ginige, A., Murugesan, S., Web Engineering: An Introduction, *IEEE MultiMedia*, 8(1), Jan.–Mar. 2001, pp. 14–18.
15. Halasz, F. G., Schwartz, M., *The Dexter Hypertext Reference Model*, in Proceedings of the NIST Hypertext Standardization Workshop (Gaithersburg, MD, USA), 1990, pp. 95–133.
16. Halasz, F., *"Seven Issues" Revisited*, Keynote talk, 3rd ACM Int'l Conference on Hypertext (Hypertext '91, San Antonio, Texas, USA), 1991.

17. Hu, J., Schmidt, D. C., JAWS: A Framework for High-performance Web Servers, in Fayad, M., Johnson, R. (eds.), *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, John Wiley & Sons, 1999.

18. Isakowitz, T., Stohr, E. A., Balasubramanian, P., RMM: A Methodology for Structured Hypermedia Design, *Communications of the ACM*, 38(8), 1995, pp. 34–44.

19. Johnson, R., Frameworks=Patterns + Components, *Communications of the ACM*, 40(10), 1997.

20. Karousos, N., Pandis, I., *Developer Support in Open Hypermedia Systems: Towards a Hypermedia Service Discovery Mechanism*, in Proceedings of the 2nd Int'l Metainformatics Symposium (MIS'03, Graz, Austria), Springer-Verlag LNCS 2994, 2004, pp. 89–99.

21. Karousos, N., Pandis, I., Reich, S., Tzagarakis, M., *Offering Open Hypermedia Services to the WWW: A Step-by-Step Approach for Developers*, in Proceedings of 12th Int'l Conference on World Wide Web (WWW '03, Budapest, Hungary), 2003, pp. 482–489.

22. Ladd, B. C., Capps, M. V., Stotts, P. D., *The World Wide Web: What Cost Simplicity?*, in Proceedings of the 8th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '97, Southampton, UK), 1997, pp. 210–211.

23. Lepthien, W. V., Anderson, K. M., *Unifying Structure, Behavior, and Data with Themis Types and Templates*, in Proceedings of the 15th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '04, Santa Cruz, California, USA), 2004, pp. 256–265.

24. Lowe, D., Hall, W., *Hypermedia and the Web: An Engineering Approach*, Wiley, 1999.

25. McConnell, S., *Rapid Development*, Microsoft Press, 1996.

26. Michaelides, D. T., Millard, D. E., Weal, M. J., DeRoure D., *Auld Leaky: A Contextual Open Hypermedia Link Server*, in Proceedings of the 7th Workshop on Open Hypermedia Systems (OHS7, Aarhus, Denmark, 2001), Springer-Verlag LNCS 2266, 2002, pp. 59–70.

27. Millard, D. E., Discussions at the data border: from generalised hypertext to structural computing, *Journal of Network and Computer Applications*, 26(1), January 2003, pp. 95–114.

28. Millard, D. E., Moreau, L., Davis, H. C., Reich, S., *FOHM: A Fundamental Open Hypertext Model for Investigating Interoperability between Hypertext Domains*, in Proceedings of 11th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '00, San Antonio, Texas, USA), 2000, pp. 93–102.

29. Nanard, M., Nanard, J., King, P., *IUHM: a hypermedia-based model for integrating open services, data and metadata*, in Proceedings of the 14th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '03, Nottingham, UK), 2003, pp. 128–137.

30. Nürnberg, P. J., Leggett, J. J., A Vision for Open Hypermedia Systems, *Journal of Digital Information (JoDI)*, 1(2), 1997.

31. Nürnberg, P. J., Leggett, J. J., Schneider, E. R., *As We Should Have Thought*, in Proceedings of the 8th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '97, Southampton, UK), 1997, pp. 96–101.

32. Nürnberg, P. J., Schraefel, M. C., Relationships among Structural Computing and Other Fields, *Journal of Network and Computer Applications*, 26(1), January 2003, pp. 11–26.

33. Nürnberg, P. J., Wiil, U. K., Hicks, D. L., *A Grand Unified Theory for Structural Computing*, in Proceedings of the 2nd Int'l Metainformatics Symposium (MIS '03, Graz, Austria, September 2003), Springer-Verlag LNCS 2994, 2004, pp. 1–16.

34. Nürnberg, P. J., Wiil, U. K., Hicks, D. L., *Rethinking Structural Computing Infrastructures*, in Proceedings of the 15th ACM Int'l Conference of Hypertext and Hypermedia (Hypertext '04, Santa Cruz, California, USA), 2004, pp. 239–246.

35. Open Hypermedia Systems Working Group (OHSWG), http://www.csdl.tamu.edu/ohs/, http://www.cs.aue.auc.dk/ohswg/

36. Papazoglou, M. P., Georgakopoulos, D. (eds.), Service-Oriented Computing, *Communications of the ACM*, 46(10), 2003.

37. Parunak, H. Van Dyke, *Don't link me in: Set based hypermedia for taxonomic reasoning*, in Proceedings of the 3rd ACM Int'l Conference on Hypertext (Hypertext '91, San Antonio, Texas, USA), 1991, pp. 233–242.

38. Parunak, H. Van Dyke, *Hypercubes Grow on Trees (and Other Observations from the Land of Hypersets)*, in Proceedings of the 5th ACM Int'l Conference on Hypertext (Hypertext '93), 1993, pp. 73–81.

39. Pfleeger, S. L., *Software Engineering: Theory and Practice*, Prentice Hall, 2001.

40. Pressman, R. S., *Software Engineering – A Practitioner's Approach*, McGraw-Hill, Fourth Edition, 1997.

41. Reich, S., Wiil, U. K., Nürnberg, P. J., Davis, H. C., Gronbaek, 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*, 5, 2000, pp. 207–248.

42. Schmidt, D. C., Vinoski, S., Comparing alternative programming techniques for multi-threaded CORBA servers: Thread pool, *SIGS C++ Report Magazine*, 1996.

43. Schwabe, D., Rossi, G., Barbosa, S. D. J., *Systematic Hypermedia Application Design with OOHDM*, in Proceedings of 7th ACM Int'l Conference on Hypertext (Hypertext '96, Bethesda, Maryland, USA), 1996, pp. 116–128.

44. Shipman, F., Hsieh, H., Airhart, R., Maloor, P., Moore, J. M., *The Visual Knowledge Builder: A Second Generation Spatial Hypertext*, in Proceedings of the 12th ACM Int'l Conference on Hypertext and Hypermedia, (Hypertext '01, Århus, Denmark) 2001, pp. 113–122.

45. Shum, S. B., The missing link: hypermedia usability research and the Web, *ACM SIGCHI Bulletin*, 28(4), 1996, pp. 68–75.

46. Tzagarakis, M., Avramidis, D., Kyriakopoulou, M., Schraefel, M., Vaitis, M., Christodoulakis, D., Structuring Primitives in the Callimachus Component-Based Open Hypermedia System, *Journal of Network and Computer Applications*, 26(1), January 2003, pp. 139–162.

47. Tzagarakis, M., Karousos, N., Christodoulakis, D., Reich, S., *Naming as a fundamental concept of open hypermedia systems*, in Proceedings of 11th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '00, San Antonio, Texas, USA), 2000, pp.103–112.

48. Vaitis, M., Papadopoulos, A., Tzagarakis, M., Christodoulakis, D., *Towards Structure Specification for Open Hypermedia Systems*, in Proceedings of the 2nd Int'l Workshop on Structural Computing, Springer-Verlag LNCS 1903, 2000, pp. 160–169.

49. Vaitis, M., Tzagarakis, M., Grivas, K., Chrysochoos, E., *Some Notes on Behaviour in Structural Computing*, in Proceedings of the 2nd Int'l Metainformatics Symposium (MIS '03, Graz, Austria, September 2003), Springer-Verlag LNCS 2994, 2004, pp. 143–149.

50. Web Services Architecture Domain, http://www.w3.org/2002/ws .

51. Wege, C., Portal Server Technology, *IEEE Internet Computing*, 6(3), 2002, pp. 73–77.

52. Whitehead, E. J. Jr., An Architectural Model for Application Integration in Open Hypermedia Environments, in Proceedings of 8th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '97, Southampton, UK), 1997, pp. 1–12.

53. Wiil, U. K., Hicks, D. L., *Providing Structural Computing Services on the World Wide Web*, in Proceedings of the 3rd Int'l Workshop on Structural Computing (SC3, Aarhus, Denmark, August 2001), Springer Verlag LNCS 2266, 2002, pp. 160–171.

54. Wiil, U. K., *Multiple Open Services in a Structural Computing Environment*, in Proceedings of the 1st Int'l Workshop on Structural Computing (SC1, Darmstadt, Germany), Technical Report AUE-CS-99-04, Aalborg University Esbjerg, Computer Science Department, Denmark, 1999, pp. 34–39.

55. Wiil, U. K., Nürnberg, P. J., Hicks, D. L., Reich, S., *A Development Environment for Building Component-Based Open Hypermedia Systems*, in Proceedings of 11th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '00, San Antonio, Texas, USA), 2000, pp. 266–267.

56. Wiil, U. K., *Using the Construct Development Environment to Generate a File-Based Hypermedia Storage Service*, in Proceedings of the 2nd Int'l Workshop on Structural Computing (SC2, San Antonio, Texas, USA), Springer Verlag LNCS 1903, 2000, pp. 147–159.