

Structure and Behavior Awareness in Themis

Kenneth M. Anderson
Dept. of Computer Science
University of Colorado
430 UCB
Boulder CO 80309-0430 USA
kena@cs.colorado.edu

Susanne A. Sherba
Dept. of Computer Science
University of Colorado
430 UCB
Boulder CO 80309-0430 USA
sherba@cs.colorado.edu

William V. Lephthien
Dept. of Computer Science
University of Colorado
430 UCB
Boulder CO 80309-0430 USA
lepthien@cs.colorado.edu

ABSTRACT

Structural computing provides techniques and tools to ease the task of developing application infrastructure; infrastructure that provides common services such as persistence, naming, distribution, navigational hypermedia, etc., over a set of application-specific or domain-specific *structures*. Within structural computing, “structure” refers to a combination of data and relationships over that data. Structure servers support the specification and manipulation of structures. One important aspect of structural computing is the power and flexibility it provides application developers in constructing new applications. A large part of this power is due to structural computing’s ability to provide *awareness services* over both structure and behavior. We define this concept and describe the awareness services provided by the Themis structural computing environment. The utility of these services are demonstrated by presenting the impact they have had on the InfiniTe information integration environment. In particular, these services help to increase the efficiency and reduce the size of domain-specific applications built using structural computing technology. We conclude by discussing how these services might influence the open hypermedia field and the development of new hypermedia services.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures—*Domain-specific architectures*; H.5.4 [Information Interfaces and Presentation]: Hypertext/Hypermedia—*Architectures*

General Terms

Design

Keywords

Themis, structural computing, structure, behavior, awareness

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HT’03, August 26–30, 2003, Nottingham, United Kingdom.
Copyright 2003 ACM 1-58113-704-4/03/0007 ...\$5.00.

1. INTRODUCTION

Structural computing provides techniques and tools to ease the task of developing application infrastructure; infrastructure that provides common services such as persistence, naming, distribution, navigational hypermedia, etc., over a set of application-specific or domain-specific *structures*. Structural computing is a new paradigm of computation that “asserts the primacy of structure over data [13].” Indeed, *structure* in this domain is thought of as a combination of data and relationships over that data. The building block of structural computing is the *structural atom*, a piece of information that has the ability to be related to other information. This orientation is not surprising; structural computing has its origins in the field of open hypermedia [15]. In [13], Nürnberg and Leggett argued that open hypermedia had been successful in developing middleware useful in constructing hypermedia-enabled applications; these applications possess desirable characteristics, such as support for distribution, collaboration, persistence, etc. However, these services were restricted to a single set of structures, namely those found in the navigational hypermedia domain. Open hypermedia was incapable, at the time, of supporting other hypermedia domains such as taxonomic hypertext [16] or spatial hypertext [10]. Thus, structural computing can be seen as an effort to generalize the techniques used by open hypermedia to develop middleware (*structure servers*) for other types of structures and domains.

Our work in producing the Themis structural computing environment [4] and using it to implement the InfiniTe information integration environment [3] has revealed the power of this approach. InfiniTe was originally implemented as a set of servlets manipulating information in a repository of XML files [2]. The new implementation, hosted on top of Themis, is smaller in terms of code size, easier to use and maintain by its developers, and more flexible and powerful than its predecessor in terms of the functionality it provides via its application program interface (API). A key factor in this new found power is the ability of structural computing technology to provide *structure awareness* to domain-specific applications and support customization of structural services via *behavior awareness*. We refer to the combination of functionality that provides both structure awareness and behavior awareness as *awareness services*.

We define *structure awareness* as the ability for domain-specific applications to *reflect* on the structure and relationships of information stored in a structure server. We define *behavior awareness* as the ability for a structure server to reflect on the characteristics of its domain-specific behav-

iors in order to customize its services. Our use of the word “reflect” in these definitions refers to the ability of some programming languages to support reflection [18], a capability that allows a program to ask questions about the structure of an object or interface at run-time, such as “What operations do you implement?” or “How many parameters does the `calculateTaxes()` operation take?”

Structure servers allow applications to learn about the characteristics of a structure in a dynamic fashion, including what information it contains, the number, names, and values of its attributes, and how it is related to other structures. This functionality allows the creation of generic operations that alter their behavior when applied to different types of structures. It also allows the creation of operations that can search for certain types of structures when presented with a seemingly incompatible structure. For instance, without structure awareness, an operation that performs keyword extraction on textual data may fail to produce any results when passed a graphical structure, such as a vector-based image. But, with structure awareness, the keyword extractor can query the parts of the graphical structure to see if any of them represent strings and then perform keyword extraction on those strings. Or, it could check to see if any textual metadata has been associated with the graphical structure and perform keyword extraction on that, before having to report failure.

With respect to behavior awareness, a structure server can ask questions of its domain-specific behaviors, such as “Are you a new type of search algorithm?” or “What types of structures do you operate on?” This functionality allows structural computing technology to provide a very generic interface to components intended to extend or customize its services. For instance, a structure server might provide a generic “transitive closure” operation that given a structure returns the set of all structures that are related to it (via parent-child relationships or attribute-value relationships). An application developer may find that such an algorithm returns too many structures for a particular purpose and can write a new algorithm, package it in a component or plug-in, and upload it into the structure server. As long as the component interface allows the developer to specify properties such as the component’s functional category, the types it operates on (or produces), etc., then the structure server can determine how to use the component and update its services dynamically.

Several mechanisms exist to provide structure and behavior awareness. For the former, the most direct mechanism is provided by the API of the structure server. A structure server’s API provides operations to retrieve structures, issue queries over them, and create/manipulate them. A second mechanism, *metadata*, is simple (typically implemented as sets of attribute-value pairs), widely used, and extremely effective. As long as each structure stored in a structure server can have an associated set of metadata, then domain-specific application developers can write behaviors that store, for instance, type information as attributes, and write other behaviors that check for the existence of this metadata and alter their behavior accordingly. A third mechanism is known as *structure templates*. A structure template gives a name to a particular organization of structure and allows operations to check if a structure is an instance of a template or to ask the structure server to search for all instances of a particular template.

With respect to behavior awareness, the most common mechanism is plug-ins that are dynamically imported into a structure server at run-time. However other mechanisms can be used, such as allowing generic behaviors to be configured via the use of environment variables or through the use of configuration files that are parsed at server initialization time. Another technique, used by Themis and described in Section 2.2, is to define an interface that information stored in structural atoms can implement to augment a generic behavior. However, this technique typically requires the use of a particular component technology and thus hinders the ability of a structure server to support applications written in multiple programming languages.

Awareness services provide a lot of power to application developers since these services allow them to develop functionality that flexibly handles many types of structures in a generic fashion. Furthermore, common algorithms can be packaged in components and moved to the structure server, reducing the footprint of domain-specific applications. In addition, templates can efficiently create large, complex structures with a minimum of application code, and can reduce error handling code since a structure server will not allow incomplete template instances to exist.

The rest of this paper is organized as follows. Section 2 presents the ways in which Themis supports structure and behavior awareness. Section 3 then describes how InfiTe uses these features. Section 4 attempts to characterize and quantify the benefits provided by these services. We then present related work in Section 5 and conclude in Section 6 with a discussion of our future research plans including how these services might impact open hypermedia technology.

2. SUPPORT FOR STRUCTURE AND BEHAVIOR AWARENESS IN THEMIS

The Themis structural computing environment has been described in [4]. Its contributions to structural computing include a simple conceptual model, a unique approach to structure templates, and the ability to support structure transformations. In this section, we provide a brief overview of Themis and then discuss how Themis supports structure and behavior awareness. In particular, we focus on how Themis’s template and transformation mechanisms have evolved since the work reported in [4].

Themis is a generic structural computing environment consisting of a structure server based on a simple conceptual framework for the specification and manipulation of structure (see Figure 1). The framework defines a generic structural **Element** that can be associated with a set of metadata, or attributes. Attribute values can take on a variety of primitive types and can also reference other structural **Elements**. An **Element** is an abstract entity that is realized by either a **Collection** or an **Atom**. A **Collection** has the ability to group other **Atoms** and **Collections**. An **Atom** has the ability to store domain-specific objects. These objects are, in general, *opaque* to the structure server, which simply stores them and allows them to be retrieved by domain-specific applications. However, as part of Themis’s support for behavior awareness, certain objects are now only partially opaque, allowing them to provide information to Themis in order to augment the operation of certain behaviors (see Section 2.2). This framework provides Themis with the ability to define a wide range of simple and complex structures.

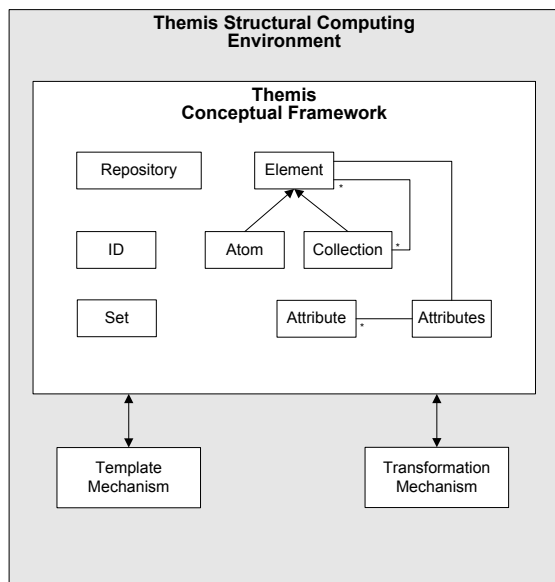


Figure 1: The Themis Architecture.

A Themis structure server can be customized for a particular application domain by defining a set of domain-specific *structure templates*. Templates allow structures to be specified declaratively and instantiated on demand. Furthermore, Themis provides a generic transformation service that can transform instances of one structure template into instances of another template, as long as a plug-in has been provided that can perform the transformation. Themis is implemented in Java and makes use of JDBC to store information in a relational database. Database integration is handled by an additional plug-in mechanism, similar to the mechanism used in the Chimera open hypermedia system [1]. Both plug-in mechanisms are implemented using a combination of Java’s ability to dynamically load new code at run-time and the specification of a set of hierarchical interfaces that plug-ins must implement. This generic extensibility mechanism provides the basis for Themis’s support of behavior awareness and is discussed, along with Themis’s support for structure awareness, in more detail below.

2.1 Support for Structure Awareness

Themis provides three mechanisms for structure awareness: the Themis API, metadata and templates. Themis’s API and Themis’s support for metadata have not changed significantly since the publication of [4]. The API provides operations to create, manipulate, and search for instances of the Themis conceptual framework. Support for metadata is inherited by **Atoms** and **Collections** from Themis **Elements**. The Themis API allows clients to create attributes, associate them with **Elements**, and retrieve their values at a later time.

As such, this section details how Themis’s support for structure templates has evolved since [4]. Briefly, the original Themis template mechanism supported a “template by example” approach in which a template was created by “recording” a sequence of calls on the Themis API. The new template is given a name and can then be instantiated on demand. A problem with this approach is that it does not handle the situation in which the structure of a template needs to vary based on information that is available at

template instantiation time. For instance, a template that represents a **shipment** structure will need to have a different destination **address** each time it is instantiated. In [4], we introduced the notion of a *structural placeholder* to address this issue, but we did not have this feature implemented. We discuss below the power and flexibility that structural placeholders provide, now that they have been implemented.

The new Themis template mechanism is composed of services for template definition, template instantiation, and template instance manipulation. Themis templates have two new concepts associated with them: **parameters** and **labels**. **Parameters** are defined at template creation time as structural placeholders for values that will be filled in at instantiation time. In many ways, they are analogous to formal function parameters in programming languages, with the exception that the valid types of a parameter are determined by its usage in the template, not by explicit declaration. Depending on its usage, a parameter value may be a string, a number, a Themis element, an object, a set of Themis elements, or even the name of a subtemplate that is to be included within the template structure (see Figure 2).

The parameters and labels of a subtemplate are available to the enclosing template by prefixing them with the subtemplate’s **Role** name. Thus, if an **address** template has a parameter called *City* and a **shipment** structure contains two **address** subtemplates, one playing the role of a **From** address and one playing the role of a **To** address, then the city parameter is available to clients of the **shipment** template as *From.City* and *To.City*.

Once a template’s structure has been defined with its associated parameters and labels, a client application can create an instance of the template using a Themis *template instantiator*. The instantiator takes the name of a template and a set of “actual” parameters and it creates a new structure that matches the template, with the supplied parameter values “plugged in” to the appropriate positions. In addition, those substructures that had labels specified for them in the template, have been “annotated” (through the use of private metadata) by the structure server to make it possible to retrieve those structures by label. Parameters and labels provide an important structure awareness capability, *structure abstraction*, since it reduces the amount of information a client application has to know about a particular structure. First, structure abstraction is achieved for applications that create structures, since client-side code does not need to understand the “layout” of a template structure; it simply supplies the required parameters and the instantiator does all of the required structure navigation *on the server side* to create the desired structure. This approach reduces the size of client-side code and increases the efficiency of structure creation, both in creating large, complex structures and in creating large numbers of smaller structures.

Second, structure abstraction is achieved for applications that use a template-based structure, since the client can retrieve everything it needs to know about a structure using its template-defined labels. Thus, a client-side application that processes a **shipment** structure does not need to know the specific location of its **From** and **To** addresses (see Figure 3). Instead, it can simply retrieve the appropriate **address** structure by label using a method provided in the Themis API. (Note: Subtemplate **Role** names are reified as labels in instantiated structures.) Here, the benefits of structure abstraction are reduced size of client-side code

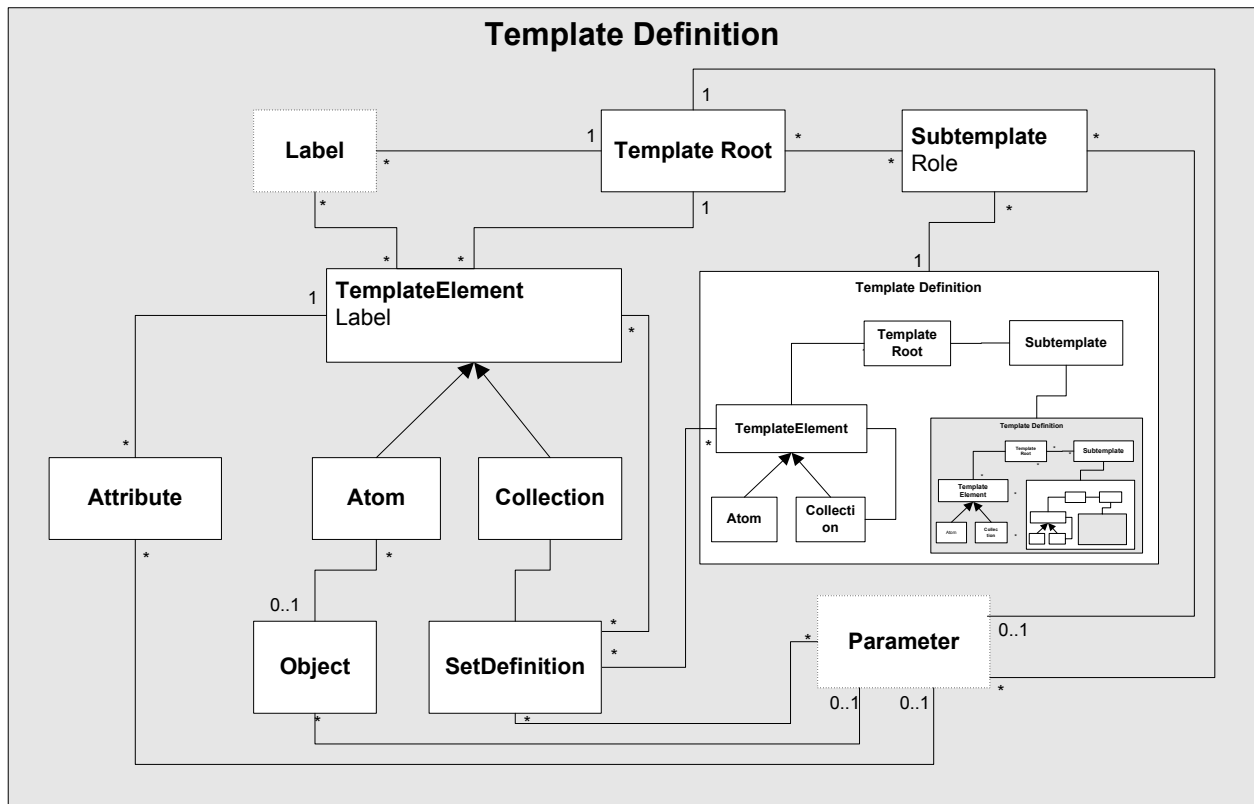


Figure 2: The Structure of a Themis Template.

and increased efficiency in accessing labeled substructures, since the navigation required to retrieve the substructure is performed by the structure server and not the client.

2.2 Support for Behavior Awareness

Themis provides three mechanisms for behavior awareness: a generic extensibility mechanism implemented via a hierarchical set of interfaces and dynamically loaded plug-ins, a structure transformation mechanism, and an optional interface that objects stored in Themis `Atoms` can choose to implement to customize the behavior of some of Themis’s generic structure operations.

The first mechanism defines an interface that all Themis plug-ins must implement in order to be loaded into a Themis structure server at run-time. This interface contains methods that allow a plug-in to supply generic metadata about itself, such as its name, version, and author. This interface is then extended hierarchically to define different plug-in categories, such as plug-ins for transforming structures (discussed below), searching structures, or computing transitive closure. Plug-ins are stored in a pre-defined location that is scanned at structure server initialization time. Additionally, the structure server can be asked to rescan this location for new plug-ins at run-time via an operation in the Themis API. The server queries each plug-in to determine what interface they have implemented and assigns each plug-in to its associated mechanism. Only structure transformation plug-ins are supported currently, but other categories will be explored and implemented in the future. To implement this extensibility mechanism, Themis makes use of Java’s facilities to define interfaces and dynamically

load code. This mechanism is not tied to the Java programming language, however, since these capabilities exist in other object-oriented programming languages.

The Themis structure transformation facility is a mechanism by which applications can request that an instance of a template-based structure be transformed into an instance of another template-based structure. (Structures created without the use of templates cannot currently be handled by the transformation mechanism.) In [4], the transformation mechanism was accessed via the Themis transformation API. This API consists of a single `transform` method that takes as parameters the names of the source and destination templates and the “root node” of an instance of the source template. The transformation mechanism then searched for a plug-in that could handle the transformation. If found, the plug-in performed the transformation and returned the new instance of the destination template. This mechanism is still supported but has now been generalized to handle the “chaining” of structure transformations (see Figure 4).

Thus, the previous mechanism could not transform a structure of type A into a structure of type C, unless a specific plug-in handled that transformation. The new transformation mechanism, however, can detect that there is a plug-in that transforms structures of type A into structures of type B, and another plug-in that transforms structures of type B into structures of type C. It can then perform the desired transformation by chaining the operation of the two plug-ins without the knowledge of the requesting application. (Our transformation mechanism makes use of techniques similar to the ones used in Odin [8] when checking to see if a valid transformer chain exists for a requested transformation.) In-

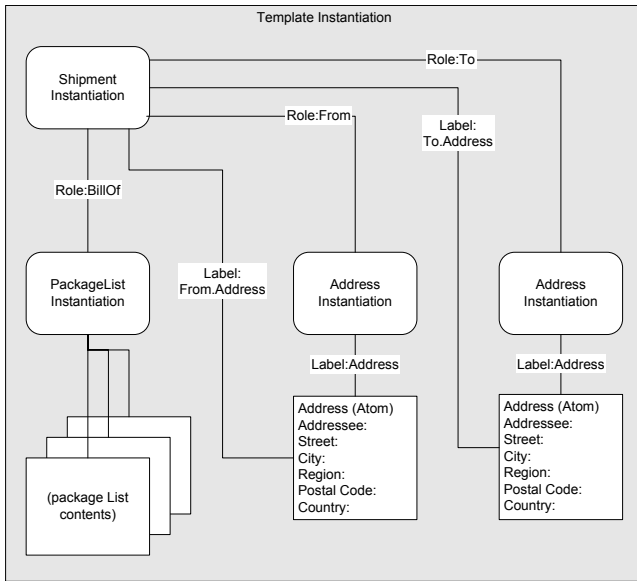


Figure 3: An Instantiated Themis Template.

deed, the requesting application no longer needs to know anything about the source structure to request a transformation. It simply asks the transformation mechanism if it is possible to create an instance of type C from a given source structure. The transformation mechanism makes use of Themis’s structure awareness services to learn the type of the source structure and then determine if the desired transformation is possible. Therefore, this mechanism helps to raise the level of abstraction at which client applications deal with structures even further, with similar benefits to what was discussed above in Section 2.1.

The final behavior awareness mechanism takes the form of an optional `ThemisObject` interface that the objects of Themis’s `Atoms` can implement to augment the behavior of generic Themis operations. In the current implementation, this interface is defined using Java, and thus, while Themis can store serialized objects from many different programming languages (bytes are bytes), this mechanism only applies to serialized Java objects. Currently, this interface defines two operations: `equals` and `matches`. The former is used to alter the algorithm to determine if two serialized objects are equivalent. (The default algorithm compares the values of an MD5 hash on the serialized bytes of the two objects.) The latter is used to alter the algorithm to find Themis `Elements` and serialized objects that match a given search criteria.

For example, the `matches` operation can be used to perform regular expression searches of Themis attributes and serialized objects that contain string values. Thus, when a search algorithm has reached a Themis `Atom` and has not found any attributes that match the current search criteria, it can now deserialize the `Atom`’s object (on the server side) and see if it implements the `ThemisObject` interface. If it does, it then invokes the appropriate operation to continue the search. This mechanism allows the capabilities of the structure server to be applied to the otherwise completely opaque domain-specific objects it stores, without requiring information from or interaction with client-side applications.

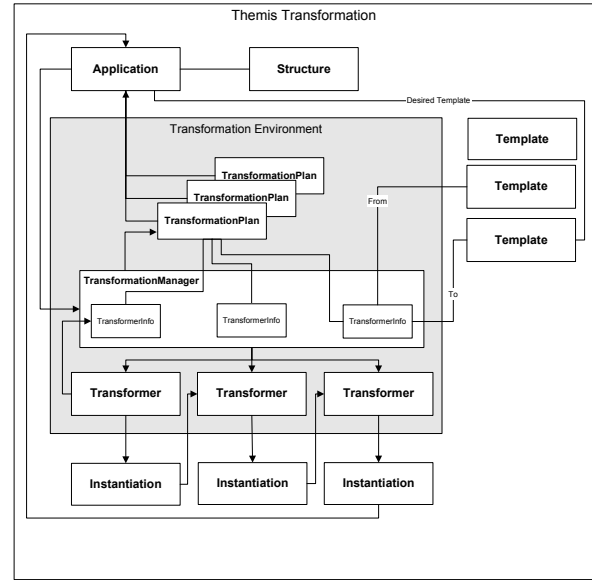


Figure 4: The Themis Transformation Mechanism.

2.3 Summary

In this section, we outlined the mechanisms that Themis provides to support structure and behavior awareness. These services provide powerful benefits in terms of reduced size and increased efficiency of client-side code. We now show how these services are being used in the InfiniTe information integration environment.

3. USE OF STRUCTURE AND BEHAVIOR AWARENESS IN INFINITE

InfiniTe [3] is an information integration environment that enables the discovery, creation, and maintenance of relationships between heterogeneous software artifacts. Information integration uses the concept of a *data source* to represent information outside an information integration environment. *Translators* are responsible for importing information from data sources into the environment and storing it in an InfiniTe *document*. Translators can also export information from InfiniTe documents out of the environment into (possibly new) data sources. Translators can be developed for different types of data sources. For example, a translator might be created to import a requirements specification; another translator might import defects recorded in the database of a defect tracking system. *Integrators* work within the environment to automate the discovery and creation of relationships. Specific integrators can be developed to find different types of relationships within the environment. These relationships can be between artifacts, other relationships, or collections of artifacts and relationships. In addition, information integration uses *contexts* to partition its information space as well as to support multiple and different views of the information space. In the first version of InfiniTe, data sources were translated into XML documents. This provided a homogeneous format for heterogeneous artifacts [3]. However, the operations of the translators and integrators were tightly coupled with the XML format of the translated documents. The second version of InfiniTe (see Figure 5) is being built using the Themis structural computing environ-

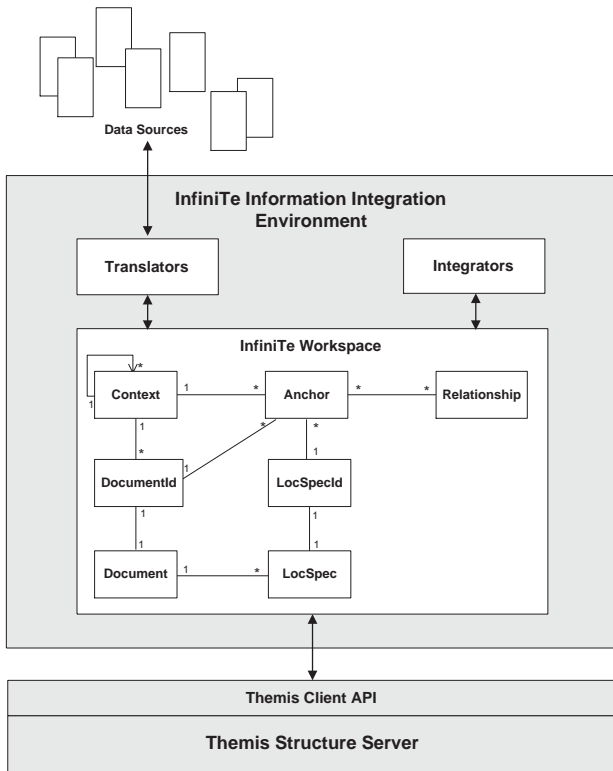


Figure 5: The InfiniTe Architecture.

ment [4]; use of Themis provides more flexibility and power for translators and integrators, especially due to Themis’s support for awareness services.

As can be seen in Figure 5, the InfiniTe Workspace interacts with translators and integrators and stores the results of their operations in a Themis structure server. The workspace provides an InfiniTe API to translators and integrators that supports such services as creating and manipulating InfiniTe structures (e.g., documents, contexts, relationships, etc.), and searching the InfiniTe information space. The workspace in turn converts these requests into calls on the Themis API that are passed to its associated structure server. Most calls that involve structure creation are handled by instantiating a template that has been pre-defined for InfiniTe’s structure server.¹ For instance, an InfiniTe document is modeled as a Themis collection that contains three subcollections, one each for storing a document’s locspecs (a term borrowed from open hypermedia [17] for a structure that stores information about the location of a “region of interest” within a document), formats (discussed below) and metadata. Within the formats collection, an additional collection is stored for each of a document’s possible formats. The code to create an InfiniTe document programmatically requires 32 lines of code and involves 33 separate calls to the Themis API. With templates, this code shrinks to 6 lines of code and only 2 API invocations (one to retrieve a handle to the InfiniTe workspace, and one to

¹Themis provides an API that allows template specification to occur external to the applications that make use of the templates [4]. Most templates are created by “single shot” standalone programs that simply connect to a structure server, define a particular template, and quit.

instantiate a new instance of the Document template). Note that in both cases, four lines of code are part of a try-catch block that surrounds the code that calls the Themis API.

InfiniTe makes further use of Themis’s structure awareness services by storing InfiniTe type information as attributes on the structures it creates. For instance, the formats collection of an InfiniTe document contains an attribute whose name is `infinite-type` and whose value is `doc-formats`. This type information is then used by translators when looking for the collection that will store translated information for a particular document. In general, InfiniTe metadata is used by integrators to identify structures that may be relevant to the type of relationships they are creating. Thus, an integrator that establishes `tested_by` relationships between a test suite and a system’s code modules will be completely uninterested in a document whose type information identifies it as a requirements or design document. Finally, InfiniTe integrators make use of Themis’s ability to search for instances of a particular template. Thus, an integrator searching for particular types of anchors can save itself a lot of work searching InfiniTe’s information space, if each anchor type is defined as a Themis template and is created by template instantiation (as opposed to created programmatically), since Themis can return all such instances as the result of a single query.² The integrator can then make use of the InfiniTe API to retrieve, for example, the InfiniTe document associated with a particular anchor or the relationships that contain a particular anchor.

Themis’s behavior awareness services have provided additional power to InfiniTe’s integrators. When importing data sources, translators are able to indicate the format they are using to store information within an InfiniTe document. For example, a requirements specification might contain a discussion of one requirement per paragraph. These requirements might then be grouped into sections based on functionality. A translator can choose to translate these requirements into an InfiniTe document in paragraph format, e.g., one requirement at a time. This might be useful for an integrator that is creating `allocated_to` relationships between the requirements specification and a component diagram with the objective of determining if all requirements have been covered by a system’s components. However, another integrator might be concerned with component cohesion. This integrator would need to use requirements in a section format (grouping similar functionality) to determine if related requirements have been assigned to the same component (indicating cohesion) or if a set of related requirements has been distributed across multiple components or if multiple sets of requirements have been assigned to a single component (both potentially indicating lack of cohesion). Unfortunately, the original translator has not stored the imported information in the desired format.

There are three approaches to solving this problem. The cohesion integrator can be designed to parse the paragraph-based format, attempting to group related requirements into sections on its own. This approach has the disadvantage that now the cohesion integrator is acting like a translator in addition to being an integrator, thus blurring the line

²InfiniTe anchors are sometimes created by translators as a data source is being imported into the information space or by other integrators whose purpose is to search the information space for instances of the anchor type in imported documents.

between the responsibilities of these two roles. Another approach is to create a second translator that imports the same data sources as the original requirements specification translator but stores the information in section format rather than paragraph format. This approach is better than the first since integrator/translator responsibilities remain distinct but has the disadvantage that it slightly increases the complexity of both translators. Now both translators have to detect if a particular data source has been translated before and, if so, avoid creating a new InfiniTe document and instead store their information in the existing document.³

A third approach is to use Themis’s support for structure transformation (part of Themis’s behavior awareness services) to define a transformer that can convert between the paragraph and section formats. This approach avoids the need for a new translator and simplifies the logic of integrators processing imported requirements specifications. In particular, after the paragraph to section transformer has been loaded into InfiniTe’s structure server, the cohesion integrator can simply ask a requirements specification document for its information in section format. If the section format does not exist, the transformer creates it without the integrator knowing that the information was created on demand. This approach simplifies the logic in the cohesion integrator since it does not have to “code defensively” with respect to processing the information of a requirements specification document. If it finds such a document, then it can be assured that its information is available in section format. InfiniTe has developed a suite of document formats for particular software artifact types, and, where appropriate, transformers have been developed to transform between the various formats. These transformers, while conceptually a part of the InfiniTe system, actually live in the structure server, thus helping to reduce the application footprint of the InfiniTe prototype.

4. IMPACT

The benefits of the structural computing approach can be measured by the impact it has on the construction of domain-specific applications. We have realized significant benefits in moving InfiniTe from its original prototype that made use of an XML repository to its current Themis-based implementation. As reported in [4], the code for existing integrators and translators was reduced by approximately 30% in terms of lines of code. In addition, a problem in which the code of the existing integrators and translators was tightly tied to the XML representation being used in InfiniTe’s repository was completely eliminated [4].

Awareness services have had an additional impact on the InfiniTe framework. In particular, the use of templates has streamlined the creation of InfiniTe’s structures. As mentioned in Section 3, the use of templates reduced the code to create an InfiniTe document from 28 lines of code (if you ignore the 4 lines devoted to a `try-catch` block) down to only 2 lines of code, a savings of 93%! Similar savings were achieved for other InfiniTe concepts, including contexts, anchors, and relationships. More importantly, the number of API calls was reduced drastically from 33 separate calls before the use of templates to only 2 API calls after, a savings

³There are additional issues with respect to translating the same data source more than once, but they are out-of-scope for this particular paper. See [3] for more details.

of 94%! This reduction is much more significant since it reduces the number of times that a Themis client must access the network and thus saves bandwidth and improves performance. Furthermore, another significant benefit is that the level of abstraction at which an application programmer works has been raised. The code is simpler to understand and easier to use and this can beneficially impact its maintainability and evolvability.

In addition, InfiniTe’s API became more flexible and powerful after the transition to Themis, because anchors and relationships became first-class citizens in InfiniTe’s conceptual framework. In the XML repository implementation, anchors and relationships were represented as XML constructs, XPointers [24] and XLinks [23] respectively. Integrators and translators were responsible for reading and writing these constructs individually with no help from the InfiniTe framework. Now these constructs are part of the InfiniTe framework and are accessed and manipulated via the InfiniTe API. Integrators and translators no longer have to worry about the structure or persistence of these concepts, and the use of Themis templates within the InfiniTe API makes it easier to create and search for new types of InfiniTe anchors and links.

The structure transformation mechanism has also helped to reduce the size of the new InfiniTe implementation while improving its maintainability. In particular, each InfiniTe document type has a number of formats associated with it. The conversion between these formats are handled by a set of transformation plug-ins that are loaded into InfiniTe’s structure server at run-time. While each individual plug-in is quite small (on the order of tens of lines of code), the impact on InfiniTe as more and more plug-ins are created is quite significant. The use of a transformation plug-in avoids the creation of new translators (which are more heavyweight entities than transformation plug-ins) and keeps transformation logic out of integrators, avoiding needless complexity in their logic. Furthermore, they increase InfiniTe’s modularity, since all transformation logic “lives” in a single place, rather than distributed across a set of entities. Finally, the transformation mechanism’s ability to chain structure transformations produces a “network effect” [21] in which each new transformation plug-in adds significant value to the system because it may enable many more transformations to occur than just the single transformation it is defining.

As can be seen, the impact that structural computing technology and the awareness services that it can provide can be quite significant. In using these techniques to implement the new InfiniTe prototype, we have seen many software engineering benefits in terms of code that is smaller, simpler, easier to use, and easier to maintain. Furthermore, we have been able to implement new functionality that increases the power of InfiniTe integrators and translators and raises the level of abstraction at which InfiniTe integrator/translator developers perform their work.

5. RELATED WORK

Themis is related to the work performed on the Construct, Callimachus, and FOHM projects. In this section, we describe these projects and situate their contributions with respect to our work on structural computing and awareness services. To save space below, it is clear from the referenced papers that all three systems support structure awareness via the common mechanisms of metadata and APIs. We

highlight additional ways that these systems support awareness services below.

More broadly, work in structural computing is related to work in other fields such as components, application infrastructure, and data models. We briefly discuss this work and its relationship to structural computing below.

5.1 Structural Computing

5.1.1 The Construct Development Environment

The Construct development environment is a structural-computing based environment used to generate component-based open hypermedia systems [22]. In particular, it provides a specification language and toolset to ease the task of constructing the services that modern open hypermedia systems provide. The development process consists of using the Construct UML tool to specify in UML the classes and methods that make up a new hypermedia service. This tool transforms the UML class diagram into an IDL specification.⁴ The Construct Service Compiler transforms the specification into a related set of files: an XML Document Type Definition, a skeleton service file, and a small set of common service behaviors. To finalize the process, the developer must implement the methods of the specified service. Then, the service is ready to be loaded into the Construct structure server and deployed.

The Construct project complements our work on Themis in that its focus is on the generation of infrastructure to support service construction, while Themis is concentrating on mechanisms that support structure definition and transformation. Each of these areas is important to the evolution of the structural computing field. With respect to awareness services, the Construct approach represents an additional mechanism for achieving behavior awareness in which new behaviors are automatically generated based on high-level specifications and then compiled into an existing structure server.

5.1.2 The Callimachus Component-Based Open Hypermedia System

The Callimachus component-based open hypermedia system makes use of structural computing techniques to ease the development of structure servers that provide support for a variety of open hypermedia domains [20, 19]. In particular, Callimachus provides a structural primitive called the *structural element* that enables the explicit specification of structures. Structural elements are based on a type system that is rooted by an abstract class, the *abstract structural element*. This class is similar to the *Object* abstract class that is present in many object-oriented (OO) programming languages [19]. Creating new structures in Callimachus is then similar to programming in OO languages: New structure classes are defined by extending the *abstract structural element* class or one of its subclasses, and then instances of these classes can be created and used by structure servers. Structural elements can be grouped into a domain with the use of a template. A template can include the elements of other templates via a process akin to multiple inheritance in OO languages.

The Callimachus approach differs from the Themis approach in that Callimachus defines a type system for creating structures, similar to the type systems of OO lan-

guages. Themis does not define a formal type system and thus cannot provide type checking or the ability to create new structures from others via inheritance. Instead, Themis provides an object-oriented framework that defines a set of structural primitives (ELEMENT, ATOM, and COLLECTION). These primitives are used to assemble new structures using the Themis API. To ease the creation of structures, the Themis template mechanism adopts a “template by example” approach. This is similar to OO programming languages that support object prototypes [7]. A template defines an arrangement of ATOMS and COLLECTIONS as a named structure. New instances of this structure can be created using the Themis template mechanism. The template thus serves as a prototype that guides the creation of new instances. While an instance has the same structure as the template, it can have different values for its attributes, atom objects, and collection members. Once an instance has been created, it no longer maintains a connection with the template, other than to store its name as an attribute on the root element of the structure.

It should be clear from this discussion that Themis templates provide an entirely different function than Callimachus templates. Indeed Callimachus templates do not support structure awareness in the same way that Themis templates do, because a Callimachus template is simply a grouping mechanism that allows a developer to assert that a particular set of structural types represent a particular application domain. Themis templates are thus most similar to the structural types that can be defined in Callimachus by extending the *abstract structural element* class, in so far that they can both be used to instantiate structures. As such, Callimachus can support structure awareness services via its structural types. Indeed, Callimachus may be able to support awareness services not possible in Themis, since Themis’s template mechanism does not support the notion of inheritance between different templates.

5.1.3 FOHM

The Fundamental Open Hypertext Model [12] unifies the structures and services of three hypermedia domains (navigational, spatial, and taxonomic) in a single data model. Having specified this model, the FOHM project has constructed tools which can support all three domains and, more importantly, can link information in one domain to information in another. FOHM can be viewed as a type of structural computing in which the target application domains were specified in advance and the infrastructure built to support FOHM-aware tools contain optimizations suited to the needs of these target domains.

FOHM offers interesting insights into the issues surrounding awareness services in that it has explored issues such as how are hypertext structures from one domain represented in other domains and how do services, such as link traversal, change from one domain to the next [12]. Their results demonstrate how issues at the client application level are simplified when knowledge about the different domains (e.g. structure awareness) is unified in a single conceptual model that all client applications know how to interpret. Themis’s awareness services are attempting to bring some of the same benefits to its clients within an approach to structural computing that does not attempt to preselect its target application domains.

⁴IDL refers to CORBA’s Interface Definition Language [14].

5.2 Component Frameworks

A standard technique to support the dynamic assembly of software tools is through the use of components. The idea of building systems from piece-parts, or components, began with the invention of the subroutine over forty years ago [11]. Since that time, computer scientists have been working to develop models, techniques, and tools to more effectively specify, document, design, and build larger and more complex system components, including the “divide and conquer” approach to develop large systems [6] and the compositional approach of software architecture and component-based software engineering in the 1990s. Popular component frameworks include CORBA [14], Sun’s J2EE, and Microsoft’s .NET.⁵

Structural computing shares with component technology the goal of making it easier for software developers to create software systems. Components are a generic technique for encapsulating functionality within an entity with a well designed interface. These entities can then be “assembled” into a functioning software system. Structural computing technology can be built using components; furthermore, developers can, for instance, choose to use both component technology and structural computing technology to develop their applications. That said, structural computing technology excels in defining the “entity objects” or domain-specific objects of an application along with the services they provide to the application, and thus may be a better choice for implementing the domain model of an application and managing distributed access and persistence of its instances, more so than the generic services provided by component technology. With respect to awareness services, all component frameworks provide a form of run-time reflection over the methods of individual components, thus providing a limited form of structure awareness.

5.3 Application Infrastructure

Structural computing is also related to work performed in other fields on tools that support the generation of applications (such as compiler construction tools [9]), tools that support the generation of application infrastructure (such as Sybase’s PowerBuilder), and/or tools that provide frameworks upon which to build domain-specific applications. The latter category can include anything from generic frameworks (such as Java Servlets and Java Server Pages or the Struts framework from the Apache Jakarta project) to application frameworks specific to a particular platform (such as Apple’s Cocoa framework) to specific technologies such as SOAP, RDF, and the like. Structural computing has built on the techniques of these fields, where appropriate, such as Construct’s adoption of a generation-based approach to structure servers.

However, as mentioned above, we believe structural computing technology is well positioned for creating the domain objects of an application. Thus, for instance, when developing a Web-based application using Java Servlets and Java Server Pages, developers discover that these two frameworks provide services that map to the “view” and “control” aspects of the well known MVC (Model-View-Control) design pattern, but offer no help in developing an application’s “model,” e.g. its domain-specific objects. As such, a de-

veloper of Web-based applications might choose to combine structural computing technology with these two frameworks to aid in the construction of a complete application.

5.4 Data Models

A question frequently arises when the specific data model of a structure server is examined for the first time. In particular, “How is this data model different from the data models provided by, say, object-oriented programming languages?” People who ask such a question claim that they can implement any example given for structural computing (such as the shipment example discussed in Section 2.1) using the data model of any programming language. We do not argue with this claim, because they are correct. We believe, however, that structural computing has several advantages in developing the domain-specific objects for applications that would argue for its use over just developing an application from scratch in a developer’s programming language of choice.

The first advantage is structural computing’s notion that any structure can be related to any other structure. This notion follows from structural computing placing more emphasis on structure than data. While it is certainly possible to achieve this in other data models, it may not have been a requirement when the data model was first developed. Thus a developer may have to “patch” an existing data model if they desire this feature; whereas with structural computing data models, this feature is an absolute requirement and will have been designed in from the start. The Themis framework supports this requirement with its `Collection` element and by the fact that the values of an `Element`’s attributes can be other `Elements`.

The second advantage is that once a structure has been described to a structure server, that server is able to provide a variety of infrastructure services for that structure, such as naming of instances, distributed access, persistence, access control, and the like. If the data model of a programming language is used to describe a structure, the programmer still has to perform a significant amount of work to provide these types of services over that structure.

Finally, a third advantage comes from the main topic of this paper, e.g. the awareness services that structural computing technology, and Themis in particular, can provide over structures. For instance, the Themis structure template mechanism can provide multiple “views” of a structure via its label and parameter mechanism. That is, the user of a Themis structure may not care about the specific layout of a particular structure; instead, it knows that this structure contains a “From Address” and it can ask the structure server to provide access to just that portion of the structure and nothing more. This mechanism provides a powerful means of encapsulating the details of a structure from the users of that structure, allowing the structure’s creator to change and evolve that structure over time without negatively impacting its use.

6. CONCLUSIONS

Awareness services are a very powerful aspect of structural computing. In particular, they offer multiple opportunities for increasing the efficiency and reducing the size of domain-specific applications built using structural computing technology. This paper has presented the multiple mechanisms by which Themis supports both structure and

⁵<http://java.sun.com/j2ee/>
<http://msdn.microsoft.com/netframework/>

behavior awareness and shown how these services have benefited InfiniTe. Our future research plans involve developing additional Themis extensions that provide ways to customize services other than search and structure transformations. Furthermore, it would be interesting to investigate how awareness services can be applied within the open hypermedia domain.

Applying structural computing to open hypermedia has already been explored via the work on Construct [22], Callimachus [19], FOHM [12] and others, allowing open hypermedia systems to support simultaneously many different hypertext domains (for instance, providing open hypermedia linking services over the spatially arranged structures of spatial hypertext). However, there is still room to explore how structural computing techniques can enable entirely new types of services within open hypermedia systems, especially with respect to the modeling of complex link structures that span multiple documents as single conceptual units (as may be enabled by structure templates). As such, we are planning to migrate the Chimera open hypermedia system [5] from its current database implementation to one hosted on top of Themis and begin to explore what it means to apply awareness services within an open hypermedia context.

7. ACKNOWLEDGMENTS

This material is based upon work sponsored by the NSF under Award number CCR-99-88517.

8. REFERENCES

- [1] K. M. Anderson. The Extensibility Mechanisms of the Chimera Open Hypermedia System. *Journal of Network and Computer Applications*, 24(1):75–86, Jan. 2001.
- [2] K. M. Anderson and S. A. Sherba. Using XML to support Information Integration. In *2001 International Workshop on XML Technologies and Software Engineering*, Toronto, Ontario, Canada, May 2001. Part of the 2001 International Conference on Software Engineering.
- [3] K. M. Anderson, S. A. Sherba, and W. V. Lepthien. Towards large-scale information integration. In *24th International Conference on Software Engineering*, pages 524–535, Orlando, FL, USA, May 2002.
- [4] K. M. Anderson, S. A. Sherba, and W. V. Lepthien. Structural templates and transformations: The themis structural computing environment. *Journal of Network and Computer Applications*, 26(1):47–71, Jan. 2003.
- [5] K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr. Chimera: Hypermedia for heterogeneous software development environments. *ACM Transactions on Information Systems*, 18(3):211–245, July 2000.
- [6] G. Bergland. A guided tour of program design methods. *IEEE Computer*, 14(10):13–37, 1981.
- [7] A. Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference*, pages 36–40, Dallas, Texas, USA, Nov. 1986.
- [8] G. Clemm and L. Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, Jan. 1990.
- [9] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, Feb. 1992.
- [10] C. C. Marshall and F. M. Shipman, III. Spatial hypertext: Designing for change. *Communications of the ACM*, 38(8):88–97, 1995.
- [11] M. D. McIlroy. Mass produced software components. In *1968/1969 NATO Conferences on Software Engineering*, pages 88–98, New York, NY, USA, 1976.
- [12] D. E. Millard, L. Moreau, H. C. Davis, and S. Reich. FOHM: A Fundamental Open Hypertext Model for Investigating Interoperability between Hypertext Domains. In *Eleventh ACM Conference on Hypertext*, pages 93–102, San Antonio, Texas, USA, June 2000.
- [13] P. J. Nürnberg and J. J. Leggett. As We Should Have Thought. In *Eighth ACM Conference on Hypertext*, pages 96–101, Southampton, UK, Apr. 1997.
- [14] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc., 1996.
- [15] K. Østerbye and U. K. Wiil. The Flag Taxonomy of Open Hypermedia Systems. In *Seventh ACM Conference on Hypertext*, pages 129–139, Washington DC, USA, Mar. 1996.
- [16] H. V. D. Parunak. Don’t link me in: Set based hypermedia for taxonomic reasoning. In *Third ACM Conference on Hypertext*, pages 233–242, San Antonio, Texas, USA, Dec. 1991.
- [17] S. Reich, U. K. Wiil, P. J. Nürnberg, H. C. Davis, K. Grønbaek, K. M. Anderson, D. E. Millard, and J. M. Haake. Addressing interoperability in open hypermedia: The design of the open hypermedia protocol. *The New Review of Hypermedia and Multimedia*, 5:207–248, 1999.
- [18] B. C. Smith. Reflection and semantics in lisp. In *Eleventh ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.
- [19] M. Tzagarakis, D. Avramidis, M. Kyriakopoulou, M. Schraefel, M. Vaitis, and D. Christodoulakis. Structuring primitives in the callimachus component-based open hypermedia system. *Journal of Network and Computer Applications*, 20, 2003.
- [20] M. Vaitis, A. Papadopoulos, M. Tzagarakis, and D. Christodoulakis. Towards Structure Specification for Open Hypermedia Systems. In *Open Hypermedia and Structural Computing, Lecture Notes in Computer Science*, volume 1903, pages 160–169, Sept. 2000.
- [21] E. J. Whitehead, Jr. Control choices and network effects in hypertext systems. In *Tenth ACM Conference on Hypertext*, pages 75–82, Darmstadt, Germany, Feb. 1999.
- [22] U. K. Wiil, P. J. Nürnberg, D. L. Hicks, and S. Reich. A development environment for building component-based open hypermedia systems. In *Eleventh ACM Conference on Hypertext*, pages 266–267, San Antonio, Texas, USA, June 2000.
- [23] XML linking language (XLink) version 1.0. <<http://www.w3.org/TR/xlink/>>.
- [24] XML pointer language (XPointer) version 1.0. <<http://www.w3.org/TR/xptr/>>.