A software architecture and framework for Web-based distributed Decision Support Systems

Shifeng Zhang*, Steve Goddard

Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, Nebraska 68588-0115, United States

Accepted 13 June 2005
Available online 11 August 2005

Abstract

This paper presents an integrated method to help design and implement a Web-based Decision Support Systems (DSS) in a distributed environment. First, a layered software architecture is presented to assist in the design of a Web-based DSS. The layered software architecture can provide a formal and hierarchical view of the Web-based DSS at the design stage. Next, a component-based framework is presented to implement the Web-based DSS in a distributed environment. Finally, an instance of the layered software architecture and 3CoFramework applied to the Web-based National Agricultural Decision Support System (NADSS) is presented.

Keywords: Architecture description language; Component; Connector; Component framework; Layered software architecture

1. Introduction

With the development of the Internet, Web-based Decision Support Systems (DSS) have become a new trend in DSS research. In [22], Power defined Web-based DSS as: “a computerized system that delivers decision support information or decision support tools to a manager or business analyst using a ‘thin-client’ Web browser like Netscape Navigator or Internet Explorer.” Compared to traditional DSS, there are two changes brought by the Web-based DSS. First, the underlying architecture for Web-based DSS has moved from main frames, to client–server systems, to Web- and network technology-based distributed systems. Consequently, large amounts of data and related decision support tools from multidisciplinary sources, which may be located in a distributed computing environment, have the possibility to be integrated together to support decision-making. Second, differing from traditional DSS, such as data-driven DSS focusing on access to and manipulation of a time-series of data, model-driven DSS focusing on access to and manipulation of a domain-related model, and knowledge-driven DSS focusing on high-level knowledge extraction and data mining [22], Web-based DSS
provide the ability to build a hybrid DSS. For example, the United States Department of Agriculture Risk Management Agency (USDA RMA) would like to integrate crop, climatic, topographic, geologic, and pedologic (soil survey) data in their spatial analysis of crop losses (indemnities) and risk management. To analyze and interpret these data, domain-related analysis tools are needed to provide value-added information. Furthermore, knowledge discovery tools and data mining tools will extract useful knowledge from the information and data to support decision-making. Given the multidisciplinary data sources and related decision support tools, the design, specification, and implementation of a Web-based DSS in a distributed environment is still an open research problem. First, a Web-based DSS often consists of the data and related tools, which come from multidisciplinary areas. Those data and related tools originally are not designed to work together. Traditional DSS design methods lack the ability to help organize them in a hierarchical view and specify the software architecture of a Web-based DSS in a formal way. Second, with the assistance of Web and network technology, the data and decision support tools from multidisciplinary areas can be located on computers distributed over a network. In such a distributed environment, a Web-based DSS needs a distributed framework to manage and integrate the data and tools in a seamless way.

This paper presents an integrated solution, from design to implementation, to tackle these problems. We propose a layered software architecture to provide a hierarchical view to organize and specify the data and related tools from multidisciplinary sources. In the layered architecture, the data and related tools are viewed as components/services. An Architecture Description Language (ADL) is then applied to provide formal specifications for components and component composition (connectors) [31]. The ADL specifies components and connectors with open interfaces and formal behavior and gives unambiguous guidance for component and connector analysis and implementation. To implement the layered architecture in a distributed computing environment, a component-based framework is also presented. The framework, called 3CoFramework, has three major roles: component, connector, and coordinator [30]. The implementation of components and connectors in the 3CoFramework follows the specification of components and connectors at the design stage. A component is a computation or storage server while a connector is a server enabling component composition; a coordinator is a server that manages components and connectors in the distributed computing environment. Fig. 1 shows the relationship between the layered software architecture and the component-based framework in a Web-based DSS. The layered software architecture creates a formal and hierarchical architecture view for the Web-based DSS while the 3CoFramework concretizes the layered software architecture in a distributed computing environment.

We illustrate an instance of the layered software architecture and component-based framework implementation using the National Agriculture Decision Support System (NADSS), a Web-based DSS developed for USDA RMA to provide decision-making for agriculture risk management. The ultimate goal of decision analysis for NADSS is to develop relationships explaining the landscape and cultural processes that describe agriculture and transform these relationships into knowledge that can be used to make sound decisions for risk management. The layered software architecture assists in the design of NADSS by providing a formal and hierarchical view. With the help of the component-based framework, NADSS provides distributed services such as computing and rendering a suite of drought indices that can quantitatively describe the intensity, duration, and magnitude of events at multiple windows of resolution (from county-level to the community- and farm-level). As a Web-based DSS, NADSS is used by researchers, analysts, and public officials working on drought risk management, drought education, drought impact assessment, and drought vulnerability.

The rest of this paper first introduces the related background knowledge in Section 2. Section 3 presents an ADL to specify the data and related analysis tools in a DSS, and how to organize those formally specified components and connectors in a layered software architecture. Section 4 introduces a component-based framework to implement the layered software architecture in a distributed computing environment. The implementation for NADSS is presented in Section 5. Section 6 gives a review of related DSS design and implementation research, and dis-
cusses our contributions. Conclusions are made in Section 7.

2. Background

There are two research areas that construct the base of our work on the design and implementation of a Web-based DSS. First is the research on software architecture (SA) used at the design stage, which provides a formal way to specify the architecture of a Web-based DSS. Second are the component-based middleware technologies, which provide a solid basis to implement a Web-based DSS in a distributed environment. Section 2.1 introduces the background of SA research; the component-based middleware technologies are introduced in Section 2.2.

2.1. Software architectures

SA emerged in the 90s as a discipline to structure complex software systems and provide a high-level system description [2,6,18,25,26]. In [25], Garlan and Shaw describe an SA as “a collection of computational components – or simply components – together with a description of the interactions between these components – the connectors”. SA typically plays a key role in bridging the gap between requirements and implementation. Ideally SA provides a tractable guide to the overall system, permits designers to reason about the ability of a system to satisfy certain requirements, and suggests a blueprint for system construction and composition [3,7]. As a research result of SAs, a number of Architecture Description Languages (ADLs) were proposed to specify software architectures; examples include ACME [8], C2SADL [18], Wright [2], and Unicon [26]. Most of these ADLs are able to abstract the software architecture of an application system as components and their interaction (connector) and provide formal specifications for components and connectors. The formal specification of connectors can help to analysis and validate component composition correctness.

However to view a complex system only as components and their interaction relationship (connectors) is not always enough, especially for domain-
related applications such as DSS or Management Information Systems (MIS), because domain-related architectural information cannot be reflected only from components and connectors. For example, for data-driven, model-driven, knowledge-driven DSS or their hybrids, data sources and related tools are required to work together so that the data from multidisciplinary sources can be aggregated into pieces of information that can lead to domain knowledge usable by non-experts. To effectively organize and understand the data, information, knowledge, and related tools in a Web-based DSS, we proposed a hierarchical architectural view in [10], which consists of four layers: data layer, information layer, knowledge layer, and presentation layer. In the data layer, components collect and manage distributed raw data and their meta-information, which provide the corner stone for the information layer. The information layer is organized around a collection of domain-specific components and connectors that provide value-added information. The knowledge layer provides components and connectors that apply data mining, knowledge discovery algorithms, and simulation models to combine information into useful knowledge for decision-making. The presentation layer uses the Web as a portal for the Web-based DSS, providing friendly interfaces based on the meta-information from the underlying data layer, information layer, and knowledge layer.

2.2. Component-based middleware technologies and frameworks

Service-oriented or component-oriented software can facilitate the complexity of distributed software development. Currently, several mature technologies are available to support service-oriented or component-oriented software development, such as Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), Java Enterprise JavaBeans (EJB), and Jini network technology. These middleware technologies are primarily concerned with standardizing external component properties, such as components interfaces and inter-component communication protocols. They also provide services to facilitate distributed system development. For example, CORBA provides naming service to register and locate remote services; Jini network technology also provides a similar service through its look up service.

However, all of them lack the ability to provide the “big picture” in developing component-based systems, e.g., component composition. Thus they decouple architecture design from implementation, allowing inconsistencies to accumulate as a software system evolves. Moreover, they do not support explicit connectors; all of the composition information is hidden in the implementation of the components [1,28]. Component composition is common and necessary in a layered Web-based DSS because dependency relationships exist among the data layer, information layer, and knowledge layer. To transform the architecture specification to an implementation, we developed the 3CoFramework [30]. There are three roles in the 3CoFramework: component, connector, and coordinator. The component and connector correspond to the component and connector specifications in the layered architecture at the design stage; they work together to solve component composition at the implementation stage. Although the coordinator has no corresponding roles in the layered architecture, it plays an important role in distributed component management. The coordinator manages the meta-information of the components and connectors at run-time. In contrast to CORBA naming services and Jini lookup services, the coordinator allows the same services to be registered in different locations, and it helps to find the optimal components or connectors for a request based on their performance metrics. Since the 3CoFramework can be implemented using different middleware technologies, it is also possible to implement the coordinator based on the Jini lookup service or CORBA naming service.

Our 3CoFramework is related to the work presented in [1] and [18]. In [1], Aldrich et al. present ArchJava, which helps to transform architecture design into a Java implementation. In [18], The C2 framework was developed to help implement component interconnection and message passing protocols for C2-style applications. Both of them are similar to our work in implementing connectors at the implementation stage. They map architecture structure to the implementation smoothly. Comparing to them, in addition to the component and connector implementation, 3CoFramework also provides a component and connector management role, coordinator, at the imple-
mentation stage. Further, 3CoFramework does not depend on specific component technology; it can be implemented either under CORBA, DCOM or Java environments.

3. A layered software architecture and formal specifications for Web-based DSS

A Web-based DSS is a complex software system; it may integrate multidisciplinary data sources and related tools to generate value-added information to support decision-making. In [22], Power introduced three DSS development methods: System Development Life Cycle (SDLC), Rapid Prototyping (RP), and End-User Development (EU). All of them, especially the SDLC and RP, usually use informal box-and-line descriptions to clarify the architecture of a DSS. Such methods bring an informal and ambiguous architecture specification to system analysis. In contrast to those traditional design methods, this section introduces a formal and hierarchical design method for a Web-based DSS. We develop the architecture specification for a Web-based DSS using a layered software architecture coupled with an Architecture Description Language (ADL). In Section 3.1, we first introduce our ADL, which provides interface descriptions and formal specifications for components and connectors in a Web-based DSS. Since the ADL also overlooks the context information of domain-related applications like DSS or MIS, in Section 3.2 we introduce our layered architecture to help organize multidisciplinary data and related tools in the Web-based DSS under a hierarchical view.

3.1. Applying architecture description language to DSS design

Like traditional software design methods, most of today’s DSS design methods use informal box-and-line descriptions that lead to a number of problems. First, the box-and-line diagrams do not provide open access points for the architecture elements; second the diagrams cannot be formally analyzed for consistency, completeness, or correctness. To specify the architecture of a software system, the software engineering community has introduced the concepts of components paired with connectors to support architecture-based software descriptions at the design level: “a component represents an independent computational unit; a connector represents an interaction among a set of components” [2]. We have developed an ADL, xSADL, to specify the interfaces and formal behavior of components and connectors [31,32]. xSADL uses Finite State Process (FSP) [14] as the formal specification method; FSP provides a theoretical base for specifying a component-based system and a model checking tool to validate the architecture specification. In xSADL, the component specification is specified as a 3-tuple: declared_message_set, declared_signature_set, and provided_service_set. The declared_message_set and declared_signature_set provide interface descriptions for a component while the provided_service_set provides interface descriptions for a component.

For example, a formal specification for a simplified Map service can be stated as below:

\[
\begin{align*}
\text{Declared_message_set:} & \\
\text{siteInfo} = & (\text{siteid: string, site\_x: float, site\_y: float, site\_value: float}); \\
\text{sites} = & \text{sequence(siteInfo)}; \\
\text{mapContent} = & \text{sequence(float)}; \\
\text{mapGenerateError} = & \text{(reason: string)};
\end{align*}
\]

\[
\begin{align*}
\text{Declared_signature_set:} & \\
\text{MapGenerate} = & \text{input sites, output mapContent) throw mapGenerateError}; \\
\text{Provided_service_set:} & \\
\text{MapService} = & \text{MapGenerate.sites -> checkSites-Available -> mapGeneration ->} \\
& (\text{MapGenerate.mapContent -> MapService | MapGenerate.mapGenerateError -> MapService}); \\
\end{align*}
\]

A Standardized Precipitation Index (SPI), which is a drought index [17], computing service can be stated as follows:

\[
\begin{align*}
\text{Declared_message_set:} & \\
\text{siteInfo} = & (\text{siteid: string, site\_x: float, site\_y: float, site\_value: float}); \\
\text{sites} = & \text{sequence(siteInfo)}; \\
\text{spiContent} = & \text{sequence(float)}; \\
\text{spiGenerateError} = & \text{(reason: string)};
\end{align*}
\]

\[
\begin{align*}
\text{Declared_signature_set:} & \\
\text{MapGenerate} = & \text{input sites, output spiContent) throw spiGenerateError}; \\
\text{Provided_service_set:} & \\
\text{MapService} = & \text{MapGenerate.sites -> checkSites-Available ->} \\
& (\text{MapGenerate.spiContent -> MapService | MapGenerate.spiGenerateError -> MapService}); \\
\end{align*}
\]
Like most domain-related analysis tools, the Map service and SPI service may already be implemented and applied in their own domain-specific application (for example, in NADSS the Map service is part of the existing GRASS GIS (Geographic Resources Analysis Support System) [19], and the SPI model has been in use since 1993 [17]). However, before being integrated into our Web-based DSS, both of them lacked open interfaces and formal behavior specifications. Using xSADL, they were assigned open interface descriptions and formal behavior specifications, which were used for system analysis and transformation to a component-based implementation. For example, to generate a SPI map, the DSS needs to make them work together. The integrated new service is called a connector in xSADL. A connector’s specification is a 4-tuple in xSADL: declared_message_set, declared_signature_set, composition_set, and message_map_set. The declared_message_set and declared_signature_set provide an open interface for the connector; the message_map_set maps possible incompatible messages between components; and the composition_set, which consists of required_service_set, provided_service_set, and glue_set, uses FSP to describe component composition. The required_service_set specifies desired components’ behavior; the provided_service_set specifies the service which the connector itself provides; and the glue_set connects the provided_service_set with required_service_set. Below is the xSADL connector example for the SPIMap service.

Declared_signature_set:
SPITempInfo(input sites, output spiContent) throw spiGenerateError;
Provided_service_set:

Declared_signature_set:
SPITempInfoMapGeneration(input sites, output mapContent) throw (spiGenerateError, mapGenerateError);
Provided_service_set:
SPIMapService=SPITempInfoMapGeneration.sites -> (SPITempInfoMapGeneration.mapContent -> SPIMapService | SPITempInfoMapGeneration.spiGenerateError -> SPIMapService | SPITempInfoMapGeneration.mapGenerateError -> SPIMapService);

Required_service_set:
MapService=MapGenerate.sites -> checkSites-Available -> mapGeneration -> (MapGenerate.mapContent -> MapService | MapGenerate.mapGenerateError -> MapService); SPIService=SPITempInfo.sites -> getSitesRaw-data -> spiGeneration -> (SPITempInfo.spiContent -> SPIService| SPITempInfo.spiGenerateError -> SPIService);

Glue_set:

The connector’s behavior can be checked as a composition of provided_service_set, required_service_set, and glue_set. With xSADL, formal analyses can be performed, such as checking whether an architectural description is consistent and complete. For example, by using the model checking tool, LTSA [14], to check the composition, we can find that the above connector contains a deadlock, which was caused by the missing consideration of the exception from the SPI Service and Map Service. The glue_set
specification below corrected the design and guides the connector implementation.

\[
\text{GLUE} = (\text{SPIMapService.SPITempInfoMapGeneration.sites} \rightarrow \text{SPIService. SPIGenerate.spiContent} \rightarrow \text{MapService.MapGenerate.mapContent} \rightarrow \text{SPIMapService.SPITempInfoMapGeneration.mapGenerateError} \rightarrow !\text{SPIMapService.SPITempInfoMapGeneration.spiGenerateError} \rightarrow \text{G} \text{LU}E).\]

3.2. Introduction to the layered architecture

The xSADL helps to transform the traditional DSS design from an informal, ambiguous specification into a formal specification with open interfaces. By introducing the SA into the Web-based DSS design, the early specification can be proven correct before the system is implemented. However, even a component/connector has its limitation: it cannot express the context information inside a domain-specific application, such as a DSS. A Web-based DSS can provide access to not only large time-series data sets (e.g., a data-driven DSS), but also domain-specific mathematical and analytical tools (e.g., a model-driven DSS) and knowledge discovery using underlying data and information (e.g., a knowledge-driven DSS). In a Web-based DSS, tools are made available by domain experts to combine data from multidisciplinary sources into pieces of information that can lead to domain knowledge usable by non-experts via the Web. Those multidisciplinary data sources and related tools can be organized under a hierarchical architecture structure to clarify their relationship. The hierarchical architecture, coupled with the formal specifications for the data source and related tools, provides hierarchical context information during the design of a Web-based DSS.

The abstract, four-layer architecture of Fig. 2 shows the hierarchical structure we have created to support the decision-making process. The data and related analysis tools are grouped into the data, information, or knowledge layer based on their meta-information and domain-specific context information, which are also used in the presentation layer to guide decision-makers. Since we separate the layers based on their meta-information and context information, it is not a rigorous layered architecture concept commonly used in software engineering. We have found in decision-making that higher layers sometimes need to access lower layers that are not adjacent. For example, the knowledge discovery tools, used in the knowledge layer, can use not only value-added data from the information layer but also the raw data from the data layer for data mining. The large vertical interface arrow at the right of the figure is meant to represent the ability of high-order layers to make requests to non-adjacent, low-order layers. Thus, the layers are only partly opaque. To improve the DSS’s performance, each of the three lower layers (data, information, and knowledge) is also associated with a cache.

3.2.1. Data layer

The data layer contains distributed spatial, constraint, relational databases, and their meta-data infor-
mation. The purpose of this layer is to provide transparent access to either local or remote data without concern for their original data formats. The interface for this layer can be hosted locally or remotely. Since the data layer is the most frequently accessed layer, a cache usually exists to help improve the performance. The data layer also provides the base to build a data-driven DSS.

### 3.2.2. Information layer

The information layer contains a collection of domain-specific mathematic or analytic tools or simulation models that help aggregate data into information. These tools, as in each of the other layers in the architecture, can be distributed over a network of computers. Examples of tools in this layer are domain-specific statistical, financial, optimization, and/or simulation models. For example, in NADSS, we grouped a geo-data spline interpolation server, climate index computing models, and soil moisture simulation models in the information layer. These domain-specific tools can provide value-added information based on raw data from the data layer. It is also possible to build a domain-driven DSS based on this layer.

### 3.2.3. Knowledge layer

Knowledge is created or discovered by combining information in new ways. Tools or applications that provide or discover domain-specific knowledge are implemented in the knowledge layer. Examples include data mining and knowledge discovery algorithms. Tools at this level might also provide more traditional domain-specific regression analysis of information (or data) generated (stored) at lower levels. The intent is that decision-makers will interact with this layer, via the presentation layer, to build and gather domain-specific knowledge. The tools in the knowledge layer do not make decisions, rather they contribute and organize knowledge that is used in the decision-making process. This layer can also be viewed as the basis for a knowledge-driven DSS.

### 3.2.4. Presentation layer

Interested users interact with the Web-based DSS via the presentation layer. The presentation layer provides a Web portal to the DSS. It manages the multi-disciplinary meta-information from the data layer, information layer, and knowledge layer. Based on the meta-information, it can reflect and provide internal data and services to Web users. The user interface can take many forms: HTML-based Web pages that interact with the lower layers via a Web server; or Java applets which can be downloaded and run on the client side. Java applets can connect to the underlying layers directly via Java RMI or a CORBA Object Request Broker (ORB).

In summary, the layered software architecture provides a general hierarchical view for the Web-based DSS. It allows DSS designers to partition the complexity of data and related tools management in a Web-based DSS into a sequence of incremental steps.

### 4. A component-based framework for distributed Decision Support Systems

The layered software architecture, introduced in Section 3, provides a formal and hierarchical view of a Web-based DSS, which can help specify and organize the data and related tools inside the Web-based DSS. Within the architecture, we view the data and domain-related tools in each layer as components or connectors (component composition for new functions). However, mapping the components and connectors specified at the design stage to the implementation stage in a distributed environment has not been addressed.

In contrast to traditional information system implementations, component-based development (CBD) transforms mainframe or client–server software to component/service-based software [28]. To implement a DSS as a collection of components in a distributed environment is also an emerging trend for DSS implementation [9,16,33]. This approach is partly motivated by the fact that many domain-specific tools already exist; they can be easily wrapped as components or linked to components in a distributed environment. Mature middleware technologies, such as CORBA, DCOM, EJB, are also available for component-based software development. Finally, the component-based development also has a direct linkage to the components and connectors specified in the design stage.

To implement a Web-based DSS as a component-based distributed system, there are two problems that must be addressed. The first problem is component
composition. Each component is designed to achieve some special task; several components can be composed together in a dependent series to achieve a larger task. Component composition is common and necessary in a layered DSS because dependency relationships exist among the data layer, information layer, knowledge layer, and presentation layer. In Section 3.1, we introduced component and connector specification using xSADL. However, these concepts mainly focus on the software architectural description at the design level, and lack the corresponding focuses at the implementation level. There still exists a gap in transitioning component and connector concepts from the design level to the implementation level.

The second problem is distributed component management. The components in a Web-based DSS may be located in a distributed environment but should be seamlessly integrated and used transparently by decision-makers. A component management server should manage the meta-information of the components and provide the suitable component instances to fulfill a client’s request based on performance considerations.

To address these problems, we have developed a component-based framework, called 3CoFramework [30]. The framework has three major roles: component, connector, and coordinator. A component is a computation or storage server which may come from the data layer, information layer, or knowledge layer; a connector is a server enabling component composition; and a coordinator is a manager server that collects and manages components and connectors in the distributed environment.

Section 4.1 provides a general overview of the 3CoFramework. Section 4.2 presents an implementation view of the 3CoFramework in NADSS.

4.1. Introduction to the 3CoFramework

Implementing a Web-based DSS as a component-based application helps to transform a DSS from a traditional monolithic system to a flexible, evolvable distributed system. As introduced in Section 3, we use xSADL to specify the components and connectors in the design stage. We believe viewing a component-based application as a collection of components and connectors is important at both the design level and the implementation level. Other research projects working on this direction include [1] and [18] (see Section 2 for more information on these projects). The advantages of separating components and connectors at the implementation level are as follows. First, localizing composition information in the run-time entity can eliminate (or at least reduce) the loss of design information. Second, implementing a component without it containing composition information makes the component more reusable. Third, the separation of the components and connectors at the implementation level makes the application evolvable; new components or connectors can be added with less impact on existing components and connectors.

The 3CoFramework pairs the component with the connector at the implementation level. In [28], a component in a distributed environment was defined as “a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and it is subject to composition by a third party”. From this definition and our experiences, we observe that there are three types of meta-information related to a component: interface information, instance information, and composition information. Interface information describes the provided functions of a component. Instance information describes component non-functional property information at run-time. Examples of non-functional property information include reference and security information. Composition information describes cooperative relationships with other components and connectors needed to implement new computational logic functions. Both the instance information and composition information may change at run-time or later during maintenance-time. The separation of these three types of meta-information at the implementation level makes the development clear and flexible. It is also natural to find that the meta-information has a direct link to component and connector concepts at the design stage.

In the 3CoFramework there are three major roles corresponding to the three types of meta-information. The connector is used to separate the composition information from the interface information; the coordinator is used to separate the instance information from the interface information and composition information; the component is independent of the composition information and instance information. Fig. 3 shows the relationship between the three elements.
The component and the connector should register themselves with the coordinator when instantiated. The connector depends on the instantiated components and connectors to process the interaction. The coordinator provides the suitable components’ or connectors’ instance information to the connector at runtime based on the performance considerations. The following sections introduce the functions of components, connectors, and coordinators.

4.1.1. Component

A component is an independent computational or storage unit at a higher level than an object. It contains self-description information and is the composable unit in a software application. A component in the 3CoFramework does not contain composition information, and it can engage in any related interaction. Below are its two functions:

- Computation logic implementation: A component implements or wraps the domain-specific computational logic with an exposed interface.
- Self-registered function: A component has non-functional property information in addition to its computational interface, such as reference information, access control information, mean execution time and execution time variance information. In the 3CoFramework, a component should register itself, with those self-description information elements, to the coordinator once it is instantiated.

4.1.2. Connector

The connector in the 3CoFramework is based on the definition from [25]: “Connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required.” The above definition indicates that the most important function of a connector is to mediate the interaction among components. It leaves unclear the meaning of “any auxiliary mechanisms required.” In the 3CoFramework, at least two mechanisms are considered to be required for a connector. One is the adaptation of interaction between incompatible components. The other is the exception handler. A connector in the 3CoFramework has the following four functions:

- Control link: A connector specifies interaction logic between involved components.
- Data format conversion: A connector is responsible to convert incompatible data transferred between components.
- Fault tolerance: In a distributed environment, the coordinator may provide outdated component property information to the connector. For example, the desired components may not exist because of hosts rebooting or processes crashing. When such errors occur, the connector needs to handle them by either re-contacting the coordinator or terminating the process. Such errors should be transparent to the end-users.
- Self-registered function: A connector performs the same registration function with the coordinator as the component.

4.1.3. Coordinator

The coordinator integrates and manages components and connectors at runtime. It has two functions:

- Component and connector information collection: The coordinator collects the instantiated components’ and connectors’ non-functional property information at runtime.
- Merit-based component and connector selection: The coordinator knows its desired components’ or connectors’ type, not their reference information, at compile-time; it depends on the coordinator to provide the suitable instantiated components or connectors at run-time. The coordinator provides algorithms to find the suitable components or connectors based on performance considerations.

The coordinator can play either a decentralized or a centralized role in the 3CoFramework, depending on...
the application scale and complexity. If the components and connectors are located in different administrative domains, decentralized coordinators are preferred for each domain. In such applications, based on the domain policy, coordinators cooperate together to collect, exchange and share the instantiated component and connector information. With small scale or single administrative domain applications, a centralized coordinator can simplify the design and implementation. The naming conflict problem can also be prevented.

The coordinator is similar to the lookup service in Jini and the naming service in CORBA. They all provide the service registration for servers and service look up for clients. However, Jini and CORBA can only provide a name-match search, while the coordinator depends on non-functional information from the registered server and host information to find the optimal server for a client. Further, the 3CoFramework connector provides a role that is not supported directly by Jini and CORBA.

4.2. An implementation view of the 3CoFramework

We view the 3CoFramework as a flexible guide that can be applied to component-based distributed applications. For example, different distributed computing middleware, such as CORBA, DCOM, or Java RMI, can be used to implement the 3CoFramework; the coordinator can be implemented as either centralized or decentralized; different accessory entities can be added to the implementation based on the security or performance considerations. This section introduces one of the possible 3CoFramework implementations, which is also the implementation of NADSS. In the implementation for NADSS, CORBA is used as the distributed computing middleware, object communication are the concern of an Object Request Broker (ORB); while the concerns of organizing a set of distributed components into a manageable framework are dealt with by the 3CoFramework.

4.2.1. Major roles

The component, connector, and coordinator are implemented as CORBA objects in NADSS. A factory pattern is used in the implementation of components and connectors. As a daemon, a factory object occupies fewer resources than its created objects; and flexible objects can be created from the factory object.

Fig. 4 shows the interface relationships between the component, connector, and coordinator. Both components and connectors are inherited from the interface AppServer, components and connectors are created by their factory and destroyed when they fulfill the client’s requirement.

A connector does not implement any domain-specific computational logic. Instead it supports the cooperation between the involved components. A connector uses a data structure rule to define the involved components and connectors. However a rule does not have run-time reference information about the desired components and connectors; a connector needs to contact the coordinator at run-time to get the location of suitable components and connectors. This design avoids hard-coding information inside the connector and can achieve better performance at run-time. The interaction between components and connectors is implemented in the Composition method. To provide the architecture information, such as the coordinator’s location, at the component factory or connector factory startup.

![Fig. 4. Interface relationships in the 3CoFramework in NADSS.](image-url)
time, an XML-based configure file is coupled with each factory instance.

A centralized coordinator is used in NADSS, which receives and manages the instantiated components’ and connectors’ run-time information. This run-time information can be serialized to an external XML file for system management and maintenance. Using collected non-functional information (see Section 4.2.2), it provides the most-suitable component or connector instance information to the connector. To facilitate development based on the 3CoFramework, the 3CoFramework in NADSS provides default implementations of coordinator and the communication between component, connector, and coordinator. The implementation is hidden from the application developers, so the application developers only need to focus on the logic implementation of components and connectors.

4.2.2. Accessory roles

The roles in this section are not essential in the 3CoFramework. However, they help to make the 3CoFramework more flexible. The Node Resource Manager (NRM) provides a node’s load information (the node is the machine from which the component or connector is instantiated) to the coordinator, which can then help the coordinator choose the most suitable component or connector at run-time based on performance considerations. The ClientBroker is a broker that allows authorized clients to access components and connectors inside the 3CoFramework.

4.2.2.1. Node Resource Manager (NRM). The non-functional property information from the component and connector may not be enough to choose the most suitable component and connector. The performance of some components and connectors depends largely on the nodes’ performance information, especially for GIS-related components such as interpolation. A NRM service can be added to each node in the 3CoFramework. It collects the state information of the host, including the CPU load, available disk space, etc. This information is then sent to the coordinator. Since there exists several successful projects to probe distributed load information, we use tools from the Network Weather Service (NWS) [29] to collect and forecast computer and network load information. NWS has four necessary processes: Name Server, Persistent State Storage, Sensor, and Forecaster. The Sensor is deployed in each NADSS node to collect the CPU, memory, and hard disk information. The Name Server and Persistent State Storage are deployed in the same node as the coordinator. The coordinator accesses the Persistent State Storage directly to get the node history load information.

4.2.2.2. ClientBroker. The ClientBroker is the broker between the clients and the 3CoFramework. The ClientBroker provides user access control; it allows only authorized clients to access the provided services. Actually, the ClientBroker can be implemented as part of the coordinator. To make the framework clear, it is separated as an independent role in the implementation view. The Client uses the ClientBroker to ask the coordinator for the desired services, and then accesses the component or connector.

5. NADSS: an ongoing Web-based DSS

5.1. Layered architecture of NADSS

NADSS is being developed for the USDA RMA. The initial focus of the NADSS project is to improve the quality and accessibility of drought related knowledge, information, and spatial analysis for drought risk management.

The 4-layer architecture introduced in Section 3 was applied to NADSS at the design stage. The data layer collects multidisciplinary data and their metadata from the Unified Climate Access Network [23], the High Plains Climate Center (HPCC), the USDA RMA Policy Database, the NASS Historical Data and Census of Agriculture, the NRCS State Soil Geographic Database, and Census2000. They provide the cornerstone for the upper layers. The information layer provides drought index information, soil moisture and temperature information, which reflects the drought situation. Geo-analysis is also provided in the information layer. GRASS [19], a popular open-source GIS software, is used as the underlying GIS to provide necessary geo-services. The knowledge layer contains data mining tools that discover relationships between various drought, climate, and meteorological indices and crop yields [11,12]. The presentation layer provides Web-based
interfaces to the clients. Two kinds of Web clients are provided in NADSS. One is the HTML-based thin client interface while the other is a thick client interface implemented as a Java applet. The HTML-based thin client uses HTTP to contact the Web Server while the Java applet uses IIOP directly to contact the underlying 3CoFramework. The website for NADSS, http://nadss.unl.edu, is the Web portal for NADSS. Fig. 5 shows the HTML-based thin client interfaces while Fig. 6 shows the Java applet-based thick client interfaces.

Although most domain-related tools in NADSS already exist, they lack open interfaces and formal behavior specifications. By using xSADL, we provide them with open interfaces and formal behavior specifications at the design stages. The xSADL documentations help to find the design errors, especially for components composition, at the early time. They also provide solid directions on how to implement the components composition at the implementation stage.

5.2. Components and connectors in NADSS

NADSS is implemented and deployed in a distributed computing environment by applying the 3CoFramework to the 4-layer NADSS architecture. The 3CoFramework was applied in NADSS based on the follow considerations:

- One objective of NADSS is to implement it as a distributed decision support system. The various components related to decision support – data relevant to a decision, decision models, visualization technologies, etc. – can be located on computers distributed over a network in a way that they can be seamlessly integrated and used by a decision-maker. Such a distributed approach can be applied within organizations where some decision support applications may be deployed over an Intranet.
- In the NADSS 4-layer architecture, it is natural to find composite service chains among the layers. For example, to produce a Standardized...
Precipitation Index (SPI) map for Nebraska; the involved services include: SPI service to compute the SPI index (information layer), and the Spatial Information Service to generate a map (information layer) (see example in Section 3.1). This service chain can be implemented as a connector in the 3CoFramework.

- NADSS has dynamic development and composition requirements throughout its life cycle; not all requirements are known in advance. For example, new application services (implemented either as a component or as a connector) may be added in the future with the evolution of NADSS. New application services should not affect the currently deployed components and connectors.

The 3CoFramework is suitable for such a distributed, dynamic, evolvable application systems. Table 1 shows the available components and connectors currently being implemented in NADSS.

In the data layer, the Meta Data Service collects and manages meta-information of all the collected data. Based on the meta-information, the Raw Data Service provides access to the underlying raw data that exists either in relational databases or in structured flat files. In the information layer, there are three drought-related computing services provided. The Standardized Precipitation Index (SPI) Service generates a precipitation-based index that can be used to monitor drought conditions on a variety of time scales; this temporal flexibility allows the SPI to be useful in both short-term agricultural and long-term hydrological applications [17]. The Palmer Drought Severity Index (PDSI) Service generates an index that represents the moisture departure for a region, implementing a simple supply-and-demand model for a water balance equation [21]. The Newhall Simulation Model (NSM) Service simulates soil moisture regimes and soil temperature regimes based on monthly climate data [20]. The information layer also provides GIS functions through the Spatial Data Service, the Spatial Information Service, and the Spatial Support Service; they wrap the underlying GIS software to provide geo-services such as interpolation, reclass, and mapping. The SPI Mapping Service and the PDSI Mapping Service integrate the GIS service with the index service to generate map format information for the SPI and the PDSI. Currently the services in the knowledge layer are still under development and will be added soon [11,12]. With the support of the 3CoFramework, NADSS is a use-while-under-development DSS; new components and connectors will be added based on requirements from the decision-makers.

With the 3CoFramework, NADSS is built as a collection of components and connectors managed by a central coordinator. It collects and analyzes instantiated component and connector non-functional property information together with node state information from the NRMs. A global naming context is used to avoid potential naming conflicts. The CORBA environments for NADSS consist of omniORB and Java IDL.

Table 1 Components and connectors in the framework of NADSS

<table>
<thead>
<tr>
<th>Layer name</th>
<th>Service name</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data layer</td>
<td>Meta Data Service</td>
<td>Component</td>
</tr>
<tr>
<td></td>
<td>Raw Data Service</td>
<td>Component</td>
</tr>
<tr>
<td>Information layer</td>
<td>Standardized Precipitation (SPI) Service</td>
<td>Component</td>
</tr>
<tr>
<td></td>
<td>Palmer Drought Severity Index (PDSI) Service</td>
<td>Component</td>
</tr>
<tr>
<td></td>
<td>Newhall Simulation Model of Soil Moisture Regime (NSM) Service</td>
<td>Component</td>
</tr>
<tr>
<td>Spatial Data Service</td>
<td>Component</td>
<td></td>
</tr>
<tr>
<td>Spatial Information Service</td>
<td>Component</td>
<td></td>
</tr>
<tr>
<td>Spatial Support Service</td>
<td>Component</td>
<td></td>
</tr>
<tr>
<td>Standardized Precipitation</td>
<td>Connector</td>
<td></td>
</tr>
<tr>
<td>Index Mapping Service</td>
<td>Connector</td>
<td></td>
</tr>
<tr>
<td>Palmer Drought Severity Mapping Service</td>
<td>Connector</td>
<td></td>
</tr>
<tr>
<td>Index Mapping Service</td>
<td>Connector</td>
<td></td>
</tr>
</tbody>
</table>

5.3. Other applications based on the 4-layer architecture and 3CoFramework

The 4-layer architecture provides a formal and hierarchical view for the architecture of a Web-based DSS. Its generalization comes from data-driven DSS, model-driven DSS, and knowledge-driven DSS [22]. We believe a hybrid of data-driven DSS, model-driven DSS, and knowledge-driven DSS will become more popular with the help of Web-based DSS. Since the data and domain-related tools inside the Web-based DSS may come from multidisciplinary sources, the 4-layer architectural view will help to organize, clarify, and formalize the design early.
Because of its generalization, the 4-layer architecture views are now also applied to other related Web-based DSS projects: a Web-based Irrigation Scheduling Program (Web-based ISP) and a Web-based Exposure Analysis and Risk Assessment for USDA (Web-based EARA). Irrigation scheduling is the accurate forecasting of water application (in timing and amount) for optimal crop production. As early as the 1960s, computer-based scheduling models were developed [13]. These programs were not widely adopted and used because they were housed on mainframe computers to which few had access and the early programs required special expertise to deal with the cumbersome input and set-up required to run such programs. In addition, the near-real time meteorological data needed to initialize and form the boundary condition on these models was limited and difficult to access. The 4-layer architecture helps to specify the related data source and domain-specific scheduling tools with open interfaces and formal behavior specifications, then organizes them into the data layer and information layer (the knowledge layer is lack in this design), and finally provides a Web portal for the presentation layer at the implementation stage.

The Web-based EARA is a more complex DSS: it contains multidisciplinary data sources from oceanic parameters, major crop production regions (agroecozones), along with agricultural practices, climate and soils characteristics, and geospatial data; the domain-specific tools include automated data quality control tools, drought, fire and flood indices models, a crop risk (FCI-33) map generation tool, exposure analysis and reporting tools, and a planting date guide tool. The diversity and complexity of domain-specific context information brings difficulty to the design of Web-based EARA. With the help of our 4-layered architecture, these tools ranged from data layer, information layer, and knowledge layer with formal specifications. xSADL helps to provide open interface specifications for those data sources and domain-related tools, then formalize their behavior to check whether there exist composition errors in their integrations. Our 4-layer architecture is not a silver bullet for DSS design. However, from our experience, we find it is practical to assist in the design of Web-based DSS.

6. Related work

Because of the complexity of DSS, the design and implementation methods for DSS are diverse. It is interesting and beneficial to give a brief review of related research on the design and implementation of DSS. In [4], metagraph is applied in the design of a DSS. Metagraph is a graphical structure that represents directed relationships between sets of elements. The research in [4] uses metagraph to represent both the qualitative aspects (e.g., determining whether certain source information is sufficient to produce certain target information) and the quantitative aspects (e.g., estimating the durations of the tasks) of workflow in a formal way. In [5] and [15], a formal logic language, PRO\textit{formal}, was introduced to specify the clinical processes. PRO\textit{formal} is based on an object-oriented Z [24]; it can be viewed as both a knowledge representation language in the artificial intelligence tradition and a formal specification language from the software engineering domain.

Like our proposed design method, researchers in [4] and [5,15] also use formal methods to specify the structure and behavior of a DSS. Comparing to conventional DSS design methods, they provide clear and unambiguous statements at the design stage. However, they still view DSS as traditionally process-oriented systems (e.g., a set of inputs, processes, outputs) instead of architecture-oriented systems (e.g., a set of components and connectors). Our ADL views a DSS as a collection of components and connectors (component composition) in the context of a layered software architecture. This view is closer to the actual implementation of components and connectors that have interfaces and provide services. Moreover, the formal method applied in xSADL is based on a process algebra (FSP), which is more general than metagraph and the formal logic language.

Other researchers have implemented DSS as agent-based systems or component-based systems. Research in [27] proposed a framework for building a DSS using agent technology to support organizations in a physically distributed, enterprise-wide, and heterogeneous information system. Zhuge [33] proposed similar research by using component technologies to implement the workflow system. However, unlike 3CoFramework they did not provide an independent connector role to explicitly deal with the composition.
All of the component composition information is hidden in the implementation of agents or components. Moreover, to the best of our knowledge, ours is the only research that focuses on providing an integrated design and implementation solution for DSS. In our research, xSADL and 3CoFramework have a strong linkage with each other. xSADL provides the specifications for the component and the connector at the design stage; 3CoFramework supports the concrete implementation of the component and the connector at the implementation stage. We also developed a Computer-Aided Software Engineering tool, 3CoCasetool [32], to connect the xSADL and 3CoFramework together. The 3CoCasetool has two purposes. First, it provides a graphical environment to compose xSADL documentation; XML-based xSADL documentation can be automatically generated from the graphical environment. Second, 3CoCasetool provides code generation functions, which help to generate 3CoFramework related codes from a xSADL document.

7. Conclusion

To provide accurate decision support information for decision-makers, data and related analysis tools are essential to a DSS. However, the required data are not only large in quantity but also come from multidisciplinary sources. The same situation exists for related decision support tools. For example, NADSS needs raw data from climatic, topographic, geologic, and pedologic (soil survey) domains. Moreover, with the related decision support tools, NADSS also generates many new kinds of value-added data; those data may also belong to different disciplinary domains. Since different interested communities may not understand information or knowledge from other communities, such complexity brings challenges to the design and implementation of a Web-based DSS.

A layered software architecture was introduced in this paper to assist in the design of Web-based DSS. The layered software architecture applies xSADL to specify the data and tools with open interfaces and formal behavior. Differing from traditional DSS design methods that use informal box-and-line descriptions, the formalization can help to analyze architecture behavior and detect design errors in early time. In addition to the formalization, the layered architecture also separates the data sources and tools into the data, information, or knowledge layer based on their meta-information and multidisciplinary context. The presentation layer provides a Web portal to guide users to data, information, and knowledge inside the Web-based DSS. The hierarchical and formal view clarifies the design of Web-based DSS.

Although the layered software architecture facilitates the design of a Web-based DSS, it does not address the problem of how to implement a service-based or component-based DSS in a distributed computing environment. A Web-based DSS conforming to the layered software architecture can also be implemented as a traditional monolithic and proprietary system. To develop a distributed component-based or service-based application, we also presented a component-based framework, the 3CoFramework, in Section 4. We can view the 3CoFramework as an extension at the implementation stage of the 4-layer software architecture from the design stage.

There are three major roles in the 3CoFramework: a component implements or wraps the domain-specific computational logic or data access; a connector implements the component interaction; a coordinator implements the distributed components and connectors management. All of the components and connectors in the 3CoFramework belong to one of the layers in the layered software architecture. This relationship creates a tight connection between the layered architecture and the framework. However, the 3CoFramework is also a general framework that can be applied to other component-based distributed applications.

A Web-based DSS uses the Web as a portal to the underlying DSS. It lets interested users access and make use of the underlying DSS through the Web. Moreover, we believe a distributed implementation of the underlying DSS is also important for a Web-based DSS. Thus, implementing a Web-based DSS presents a challenge, which needs the combination of a DSS with distributed computing technology. Our proposed layered software architecture coupled with the component-based framework provides one practical way to implement a Web-based DSS. As a result, NADSS adopts the 4-layer software architecture in the design stage and the 3CoFramework in the implementation stage. Although NADSS is still under development,
this design and implementation methodology can help NADSS to add new services later. NADSS can be accessed from http://nadss.unl.edu.

Acknowledgements

This work was supported, in part, by a grant from NSF (EIA-0091530) and a cooperative agreement with USADA FCIC/RMA (2IE08310228).

References


Shifeng Zhang is a PhD candidate in Computer Science and Engineering at the University of Nebraska–Lincoln. He received his master degree in Computer Science from East-China Institute of Computer Technology. His research interests include software engineering, component middleware technologies, DSS, and GIS.

Steve Goddard is an associate professor of Computer Science and Engineering at the University of Nebraska–Lincoln. He received his PhD in Computer Science from the University of North Carolina at Chapel Hill. His research interests lie in the broad areas of real-time and distributed systems with an emphasis on the practical application of research, as well as developing software engineering methods for building complex systems.