FACT: A Fusion Architecture with Contract Templates for Semantic and Syntactic Integration

R. Gamble, R. Baird
Software Engineering & Architecture Team
University of Tulsa
800 S. Tucker Drive.
Tulsa, OK 74104
gamble@utulsa.edu

L. Flagg, M. McClure
Sentar Incorporated
4900 University Square, Suite 8
Huntsville, AL 35816
lflagg@sentar.com

Abstract

Linking components with end-user requests for processing is problematic when there are fundamental language differences between component specifications and how individual users state their needs. Appropriate components may not exist, the users may not know if a component exists until a one matching their requirements is generated, or the users may adjust their requirements. For complex systems governed by a community of interest, we introduce a Fusion Architecture coupled with Contract Templates (FACT). The Fusion Architecture assists with syntactic and semantic unification of user directives. Contract Templates provide a standardized mechanism to collect heterogeneous systems within domains of interest. Based upon specific analysis of each component, Contract Templates attach connectors to generate integrated systems to which queries can be directed. A case study demonstrates how FACT enables military analysts to direct the use of simulation software for the experimentation of command and control behaviors within missions.

1. Introduction

Software has advanced to the point where domain constituents simultaneously provide and consume information introducing an array of competing components that can appropriately respond to meet system goals. We define a *component* to be a set of software functionalities defined through non-standardized and often distinct semantic descriptions which are discoverable by a *community of users* that are engaged to perform a specific software-related task. Components can be a variety of software types, including COTS, web services, games, simulations, and legacy systems. Like social communities, software communities of interest are related by geographic areas, an administrative domain, or common goals. Thus, their general identity, functionality, and interaction can be predetermined [1]. The domain of the community narrows the scope of components needed to fulfill their functional requirements. However, connecting a community to components without knowing comprehensively the individual features and functions expected can result in interoperability problems.

One challenge is reducing the time invested in reviewing a component for fit, functionality, and unique contribution to the community. The goal is for rapid, decisive access to components, as facilitated by smooth, seamless integration when components are dynamically available, such as services for emergency management. This access requires an understanding of the component functions and the interface it offers to the community. However, even with complete access to documentation, API specifications, and source code, this integration process may still be troublesome. Software developers may describe functionality using completely different characteristics than the community of users resulting in mismatched semantics. Guaranteeing that the user goal can be matched to the service results is part of the integration comprehension that is needed.

There are often syntactic and semantic differences in the interfaces provided by software components. A software component may expect to be invoked in a particular manner, may only accept certain types of input data, or may not expose a standard set of APIs. Rarely do software components provide all the necessary information required to implement integration [2]. These issues indicate that analysis efforts are best spent accumulating the properties of software components and bridging the gaps between their uses and styles.

Throughout the remainder of this paper we introduce FACT and its two main technologies: a Fusion Architecture and Contract Templates. FACT promotes plug-and-play incorporation of new technologies with multiple modes of operation and interaction. Emphasis is

978-1-4244-2660-7/08/$25.00 ©2008 IEEE

placed on integrating components with the community using Contract Templates to determine functional interoperability. The Fusion Architecture customizes the community interface according to the available services and their expressed functionality. It maintains semantic consistency within the infrastructure, allowing refinement by both the community and the Contract Templates as components are introduced. An implementation of FACT for a military community that fuses wargames for simulation experimentation and analysis is presented.

2. Background

Integrating complex, heterogeneous components within software architectures can be accomplished by inserting connectors to establish seamless interaction [3, 4]. A wide variety of software connectors have been cataloged to manage syntactic issues [5], and correct application of connectors is required for seamless integration. Therefore, ad hoc connector usage is not only inefficient but also may not provide adequate linkage between users and functions in multi-component systems. Most research in the use of connectors is associated with integrating components with other components, not linking components to users. Developing custom-made connectors for each component facilitates integration, but disregards accepted software reuse practices. Reuse of connectors can be beneficial to reduce customization and maintenance issues [5]. Deeper analysis of the properties and characteristics of software components is needed to uncover the specific connector styles for an integration scenario [6]. Thus, properties must be incorporated into a connector when delivering functionality. Increased uniformity is needed for services, tasks, and results to be delivered to the community for viewing and comparison.

Specifications written according to Architecture Description Languages (ADLs) can describe the overall architecture of software components in relation to the community and assist in determining which connectors need to be written or applied to specific components for integration. Expanding ADLs with information about the structure and the state of the environment has been used to provide users with information about optimal configurations [7]. This research focuses on selecting components based on performance indicators, not on the functionality that its services provide. Frameworks have been generated that formally define the interfaces of multiple software components [8]. However, the goal of these systems has been for the interchange of data between components and has not focused on standardizing fusion with end-users. Furthermore, execution of the components may result in a problems due to expectation of communication or interactivity [9].

Semantic issues arise among providers and users even in the context of the same domain or task expectation. Therefore, a framework is needed to semantically match the appropriate components to a domain, community, task, and goal by clearly defining the needs of the community and then comparing this information to the data (often, meta-data) given in the provider's specifications. Analysis must be conducted to uniformly configure the communication expectations between components and users. Each component expects input to be formatted in a particular manner further requiring the translation and management of user requests. Thus, uniformity is essential to rapid access and information delivery.

Ontologies formally define concepts within a domain such that anyone interested in the domain can consistently understand their meaning unambiguously. Ontological specifications enable providers to express the functionality that their software component contains so users can understand the specific semantics attached to that component. However, ontologies may vary in the depth of their expressiveness. Though, ontologies have been developed for the domain of games [10], these specifications would provide very little semantic use to a community that wishes to utilize commercial game engines for military simulations since it lacks details which would be contained within a military ontology. Sometimes the union of one or more semantic specifications is required for useful meaning to become apparent. When semantics are not accounted for, they can introduce integration problems that may not be apparent until a system is deployed.

Traditional component integration is accomplished via the use of established communication platforms and connector software. The communication platform is some form of middleware that often spans many technologies and programming environments to address issues of heterogeneity including: languages, connectivity, dynamism, performance, and reliability [11]. This can resolve some of the difficulty in component integration. Frequently, however, the attachment of additional connectors is necessary to link software components with middleware [12]. Thus, an architectural understanding of the middleware and the participating components is still required. Furthermore, middleware does not embody support for resolving the semantic problems associated with components.

Typical middleware solutions focus on providing cross-communication between multiple heterogeneous software components. The middleware technology serves as a communication platform linking the individual components together. Connectors are used to link the individual components with the middleware, and the middleware facilitates component communications. Communication can be established between components

within the same domain or between domains. Details about linking the individual users of the community with the components are typically left to the community to resolve after integration has occurred. Hence, a different architecture is needed within which a community can engage multiple, available components to accomplish a task. The properties of a community, the components, and sources of information are those that:

- Contribute to the overall community tasks and information needs
- Are stand alone & autonomous, that is, do not rely on or expect to interact with other components that are not part of its own infrastructure
- Require fusion to incorporate unique characteristics, identify redundant or similar characteristics provided by desired and fused components, and determine real-time availability

FACT needs to place sufficient emphasis on generating a consolidated information space such that each user can take advantage of available components and services as needed. It assumes that connectors are still required but seeks to establish uniformity across them. Thus, the goal of FACT is to resolve the semantic and syntactic difficulties associated with linking a diverse set of components to a community. Furthermore, the solution should be practical and easy to develop. Our approach is to combine connectors and an ontological foundation to provide access to a critical mass of components, while reducing the complexity of integration.

3. Fusion Architecture

In this section we introduce the first portion of FACT, the *Fusion Architecture*. The Fusion Architecture uses two interfaces to link users to components, resolving the semantic and syntactic issues common to heterogeneous software systems. Figure 1 depicts the entities within the Fusion Architecture that are used to link multiple users to *component providers* (*providers* hence forth) via a *multi-user interface* and a *fusion interface*.

The *multi-user interface* specified by the Fusion Architecture exposes to the user only the currently connected component providers. Any user requests directed through the interface are guided to one or multiple of these providers according to their appropriateness to achieve the user's goal. The user may have no knowledge of which provider was selected.

The *fusion interface* is used to communicate in a loosely coupled manner with available providers. The interface performs the actual linking between the community of users and the specific components that have been fused into the architecture. The key challenge is to normalize the set of semantic properties associated with the provider and the functionality it supplies. Once the selection of a provider is accomplished for the user via the multi-user interface, the Fusion Architecture uses descriptions of each component, specified by *Contract Templates* (see Section 4), to map the properties of each provider to an internal representation understandable by the community. *Fusion calls* incorporate each provider into the architecture.

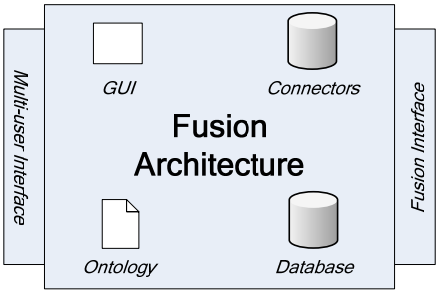

Figure 1. Fusion Architecture

The four components in Figure 1 are described as follows. The *Graphical User Interface* (GUI) provides the community with access to FACT, including the underlying providers which have been integrated for use. The GUI offers users the ability to input requests for functionality by selecting particular attributes and displaying their current dependencies. Thus, the user interface shows the specific providers that support partial or full user requests as directed by parameters set forth by the GUI. Since the results displayed within the interface must already have provider backing, the execution of the component can immediately proceed. Thus, the Fusion Architecture is accurately guided to select an appropriate, *available* provider for execution.

The *Ontology* stores attributes about the community and the specific components it uses. It contains a semantic representative of desired properties and goals, along with known providers. The ontology manipulates the stored attributes to direct the end users to the desired provider(s). For example, the ontology may hold military simulation terminology and use it to direct an analyst to the best fit simulation package for their experiment. Therefore, ontology specifications within the architecture, if used properly, can enable each provider to be commonly expressed such that users understand the specific features each component provides.

The *Connector Library* stores connectors that were used to resolve the interoperability problems discovered during integration. Storage includes the generic interoperability problem resolved and the deployment details of the connector for future reference and analysis. This information assists in reuse connectors that are common for the different component providers.

The *Database* is used for several functions. First, the results of each user and provider interaction are stored for review and to determine the conditions under which components were selected in relation to user requests.

This allows for deeper analysis across multiple executions of the same scenarios, as well as the historical storage of decisions mapped to current information tasks. Second, a library of necessary Contract Templates is stored so that the user interface can select components based on their characteristics and availability.

4. Contract Templates

To overcome the challenge of linking software from a variety of external organizations and vendors, we devise and implement reusable, customizable *Contract Templates*. The primary objective of a Contract Template is to expose the functionality of a variety of components in a uniform intermediate language to achieve a common abstraction. Figure 2 shows an expanded view of the infrastructure depicting the relationship between the specifications to the providers and the architecture.

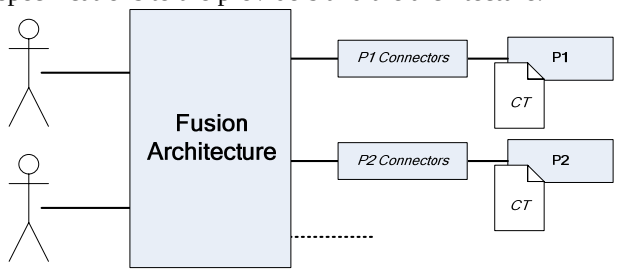

Figure 2. Expanded Fusion Architecture

A component binds itself to a community by instantiating a dedicated Contract Template. Each Contract Template is separated into three distinct levels (Figure 3): (1) the community's predefined ontology, (2) expected user directives, and (3) end processing. Each level is associated with and expands different sections of the full ontology managed by the Fusion Architecture.

Contract Template
Level 1: Predefined Ontology **• Component Type** **• Functions / Methods** **• Results / Return Values**
Level 2: User Directives **• Parameter values** **• Command line arguments**
Level 3: End Processing **• File names, Default values** **• Connectors**

Figure 3. Contract Template

The first level of the Contract Template contains ontology information, such as component type, functions or methods it exposes. Incorporating unique semantics for each component allows the ontology to expand with each fusion call. As the ontology matures, new component providers can leverage it to describe their semantic functionality in a way that is unique to the community and their needs. This process affords end users with additional details about each component which may influence its component selection and accessibility.

Initially, the community develops an ontology related to the application domain for which FACT is being used. An initial set of providers are gathered. The semantic descriptions of their functionality are collected and compared to refine the ontology. Together the set of specifications generate an initial Contract Template to be used. Taking the current ontology, each provider then refines the Contract Template, which propagates back to the Fusion Architecture to maintain consistency. The final ontology consolidates and classifies the information expected by the component and its provider using its attributes and values to fully complete the Contract Template where information may be missing. This process fills any gaps between the full ontology specified by the community and the implementation details provided by components.

As the information space of providers matures, the ontology is updated such that the community users can continually take advantage of all providers. The Contract Template is easily stored as an XML document facilitating portability, extendibility, and reuse between providers. Figure 4 shows the initial XML document. The XML representation initially begins with elements such as <*title*> and <*author*>, and expands depending on community needs.

```
<contractTemplate>
  <title></title>
  <author></author>
  <description></description>
  <type></type>
  <modes></modes>
  <selectors>
    <selector type="..." />
  </selectors>
  <modifiers>
    <modifier type="..." />
  </modifiers>
  <runtime></runtime>
</contractTemplate>
```

Figure 4. Contract Template XML

The second level relates to the user and their employment of the Contract Template to select a particular component for execution (stored in XML as <*selectors*>). Within this level resides the user modifiable values provided by the component (stored in XML as <*modifiers*>). The community defines the set of available selectors and modifiers which can be used within their Contract Templates. Each component is reviewed to determine which specific XML elements should be stored in the contract. Ultimately this allows a user to define a set of criteria upon which they wish to select components, using the Contract Templates to those

that do not match the desired requirements. Because of the flow-through connectivity, the GUI will only display to the user the available selectors and modifiers given the currently accessible components.

The last level is the end processing calls to and results from the component. Execution details are divulged, such as file names and invocation details, both default and ranges of parameters, and any connectors that must be used for communication. This information is specific to each provider and includes the component location, an API or method names, any support for command line execution, and supported data formats. End processing yields the information necessary to call connectors to instantiate the component with the user-selected modifications to execution information. This is where the connector database of the Fusion Architecture is used. Connectors are bound to Contract Templates to resolve any syntactic or semantic differences in the calls to the component. The complete Contract Template is stored and made available within the Fusion Architecture so that the GUI has real-time access to determine what is available to the community.

5. Multi-Game Fusion

We demonstrate the effectiveness FACT within an example community that analyzes military simulations. Multi-Game (MG) Fusion [13] is a Java application that has an experimentation-based authoring, execution and analysis capability that uses FACT to integrate a variety of military simulation software, such as wargames. For an initial ontology, the MG-Fusion community defines a *scenario* to refer to the specific simulation event that a wargame delivers. Wargames often use different scenario terminology (e.g., mission, operation) depending on the domain. Providers must have predefined scenarios or an API from which to create scenarios into order to form their representative *Contract Template*.

An *experiment* includes one or more scenario executions allowing the simulations to execute consistently with changes to prescribed settings. Note that these terms can easily be analogously defined for other communities.

The MG-Fusion ontology is authored in the Web Ontology Language (OWL) [14], using the information commonly available and related to wargames. The ontology includes 22 classifications of information and over 80 attributes. The full specification is omitted due to length considerations. Within the ontology, an *experiment* is composed of a set of *scenarios*, a game engine (*wargame*), a set of *players* with associated command hierarchy, and a description of the artificial intelligence used by the game. Each scenario for an experiment correlates to a specific area (*map*), a designation for the type of campaign (ground or air), and behaviors and tasks.

For each wargame component, the Contract Template is instantiated at level 1 directly from the ontology. Analysis of each component ensures that the properties defined within the ontology match those that its provider describes in documentation and execution data. For example, a military wargame may provide air support within a particular mission, but that would not be indicative of the full scope that the simulation represents if the simulation realistically models ground troops.

The next level of the Contract Template, user directives, appends information to the XML document relating to the selectors and modifiers that the component provides. After an investigation of military wargames, the allowable list of selectors includes simulation features such as: *Air Support* (air forces to support operations), *Deteriorating Operations* (operations conducted under less than optimal conditions), *Fog of War* (limited information about forces), and many others (engineers, responsiveness, supplies, weather, etc.). To represent that the wargame supports a specific selector, the Contract Template is updated to contain a reference to that selector type. For example, to indicate that a component contains attributes related to available air support, the entry *<selector type="Air Support" />* is be added to the contract template.

Modifiers are notated similarly. The modifiers, as determined by the initial sampling of providers, include: *Launch Interval* (the delay between the deployment of units or weapons), *Leadership* (capabilities of commanders within the simulation), *Morale* (overall morale of units), *Preparedness* (a level of readiness), and *Sensors* (existence and the information gathered from their locations).

```
<contractTemplate>
  <title>Tour of Duty</title>
  <author>John Tiller</author>
  <type>Large-Scale Ground Sim</type>
  <modes>Batch, AI</modes>
  <selectors>
    <selector type="Air Support" />
  </selectors>
  <modifiers>
    <modifier type="Leadership" />
  </modifiers>
  <scenarios>…</scenarios>
  <runtime>
    <preprocessor>apxml.exe</preprocessor>
  </runtime>
</contractTemplate>
```

Figure 5. Tour of Duty Contract Template

To complete the Contract Template, the final level adds wargame specific information about file names, variable ranges, and how the specific game can be invoked by the fusion interface. Execution and connector information is added to the *<runtime>* element of the Contract Template. For example, to indicate that a

wargame requires the connector "*apxml.exe*" the <*runtime*> element is added as shown in Figure 5.

Figure 6. Selector User Interface Dialog Box

After fusion the community relies on the GUI and Contract Template to describe attributes of the software. The GUI presents the community with decisions for narrowing the selection of a provider. For example, a community user can specify the type of simulation (e.g. "Air Mission", "Ground Mission"), and then choose the desirable characteristics, such as "Leadership," for selectors and modifiers. Figure 6 shows the dialog box used by MG-Fusion to display the options. Finally, the user is presented with a list of providers that match their search criteria.

Since the Contract Template XML file contains the necessary execution information, once a user has selected a provider and parameter values, the component is executed by automatically invoking the appropriate connector as specified in the Contract Template, yielding a seamless process of execution. If the community allows MG-Fusion to choose the component, the execution is completely transparent.

MG-Fusion has successfully generated an extensible and reusable experimental environment to promotes plug-and-play incorporation of new simulations. The support for batch execution enables military analysts to carry out complex, time-intensive experiments whose results can be stored by MG-Fusion for further analysis.

6. Conclusion

Using different providers presents communities with a variety of semantic and syntactic issues with no clear solution for transparent execution. Within FACT, interoperability problems are resolved before a component is made available to a community. FACT has successfully been deployed for MG-Fusion and is currently being expanded to federated information spaces.

One drawback of FACT identified by MG-Fusion is the manual generation of an initial ontology. When significantly different providers are fused, the ontology must be updated to reflect their unique features. This process of ontology refinement can be difficult as the specification changes rapidly. Eventually, the process stabilizes when the Contract Template and ontology have evolved into a cohesive format.

Acknowledgement. Special thanks to John Tiller for providing the initial set of military wargames for inclusion within the Fusion Architecture and for helping develop the initial ontology used within MG-Fusion.

This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA8750-05-C-0210. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

7. References

[1] S. Renner, "A "Community of Interest" Approach to Data Interoperability," Federal Database Colloquium, 2001.

[2] J. A. Stafford and A. L. Wolf, "Annotating Components to Support Component-Based Static Analyses of Software Systems," in Grace Hopper Celebration of Women in Computing Hyannis, Massachusetts, 2000.

[3] L. Davis, R. Gamble, and J. Payton, "The Impact of Component Architectures on Interoperability," Journal of Systems and Software, 2002.

[4] N. Medvidovic, R. Gamble, and D. Rosenblum, "Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms," in Fourth International Software Architecture Workshop (ISAW-4) Limerick, Ireland, 2000.

[5] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," in Int Conference on Software Engineering, Limerick, Ireland, 2000.

[6] G. Jónsdóttir, "Notating Problematic Architecture Interactions," M.S. Thesis, Department of Mathematical and Computer Sciences: University of Tulsa, 2002.

[7] N. Arshad, D. Heimbigner, and A. L. Wolf, "Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems," Software Quality Journal, vol. 15, pp. 265-281, 2007.

[8] G. A. Mills-Tettey, G. Johnston, L. F. Wilson, J. M. Kimpel, and B. Xie, "The ABELS system: designing an adaptable interface for linking simulations," Proceedings of the 2002 Winter Simulation Conference, pp. 832-840, 2002.

[9] R. Fujiomoto, "Distributed Simulation Systems," in Winter Simulation Conference, 2003.

[10] J. P. Zagal, M. Mateas, C. Fernández-Vara, B. Hochhalter, and N. Lichti, "Towards an Ontological Language for Game Analysis," in Changing Views – Worlds in Play, Digital Games Research Association (DiGRA), 2005.

[11] N. Medvidovic, E. M. Dashofy, and R. N. Taylor, "The Role of Middleware in Architecture-Based Software Development," International Journal of Software Engineering and Knowledge Engineering, vol. 13, pp. 367-393, 2003.

[12] D. Flagg, R. Gamble, R. Baird, and W. Stewart, "Migrating Application Integrations," in International Conference on COTS-Based Software Systems , 2004.

[13] Sentar Inc., "MG-Fusion," 2008, http://www.sentar.com/mgfusion.htm.

[14] D. L. McGuinness and F. Harmelen, "OWL Web Ontology Language Overview," W3C, 2004, http://www.w3.org/TR/2004/REC-owl-features-20040210/.