International Journal of Information Technology
Vol. 8, No. 1 August 2002

Automated Component Ensemble Evaluation

Somjai Boonsiri
Chulalongkorn University
Bangkok, Thailand 10330
+662 218 6991
Somjai.B@chula.ac.th

Robert C. Seacord
Software Engineering Institute
Pittsburgh, Pennsylvania 15213
+1 412/268-7608
rcs@sei.cmu.edu

Russ Bunting
Software Engineering Institute
Pittsburgh, Pennsylvania 15213
+1 412/268-9150
rbunting@sei.cmu.edu

Keywords: component repository, Enterprise JavaBeans, component specification, component evaluation

Traditional, large-scale component repositories have failed for a variety of reasons. One of these reasons is that they have focused on the identification and selection of individual components, while system development often depends on the selection of compatible component ensembles. In this paper, we discuss an alternative approach to traditional component repositories that integrates knowledge-based techniques to automate the selection of component ensembles. In particular, we focus on the attributes of component specifications in the constrained problem space of Enterprise JavaBeans, and the development of associated integration rules that evaluate these attributes with respect to component integratability. We also discuss the role of patterns in automated component ensemble evaluation.

1 Introduction

The benefits of developing an effective component library are readily apparent: by allowing system integrators to fabricate software systems from pre-existing components rather than laboriously develop each system from scratch, enormous time and energy can be saved in the development of new software systems. The President's Information Technology Advisory Committee (PITAC) interim report [12] to the President states that:

> *The construction and availability of libraries of certifiably robust, specified, modelled and tested software components would greatly aid the development of new software.*

However beneficial a component library might be, a useful and effective repository has been an elusive goal. Traditional software libraries have been conceived as large central databases containing information about components and often the components themselves. Examples of such systems include the Centre for Computer Systems Engineering's Defence System Repository, the JavaBeans Directory, and the Gamelan Java directory.

Challenges with this currently employed approach include design of the classification scheme for organizing the contents of the repository and the suitability of search results during retrieval. Given the dynamics of the commercial marketplace and technology in general, classification schemes often exhibit fragility over time. That is, classification categories based upon product features may over time evolve or be subsumed completely. Secondly, users of the repository must be familiar and capable with the chosen scheme. The complexity of the scheme can be hindrance to new users of the repository and to its adoption outside a group of advanced users, [13] limiting the usefulness of the repository. In addition, software repositories face an inherent dilemma: for the approach to be useful, the repository must contain enough components to support users, but when many qualified components are available, finding and choosing an appropriate one becomes troublesome [9]. Furthermore, most software retrieval systems retrieve a set of candidates ranked by suitability to search criteria. Often, it is impractical for the user to analyse and evaluate the entire list of candidate components. Instead, the user assumes that the components at the top of the retrieved list (e.g. the first three) are the most appropriate. Then, to select a component from the list, the user examines only those first components. If no components satisfy the requirements, the system integrator may modify the retrieval search query, but in most cases the search will be abandoned. For this reason, repository retrieval systems must exhibit more precision in their answers, by discarding some obviously unwanted components from the set of candidates and by retrieving only the best ones [7].

Ideally, an improved system should avoid a common deficiency of exclusively allowing searches for individual components, as component-based systems are not built from individual components, but component ensembles [18]. A component ensemble is a set technologies, products, and components that interact to provide some useful behaviour. Since individual components rarely satisfy the entirety of requirements in a component-based system, component ensembles replace components as the fundamental architecture and design unit. For component repositories to be successful, they must allow system integrators to search for highly compatible component ensembles and not simply individual components.

2 Component-based Development

Component-based system development requires the simultaneous survey of the marketplace (including commercially available components and applicable standards), the system context (including requirements, cost, schedule, business processes) and the existing system architecture [8]. The system context may also be constrained by corporate standards and policies and the system architecture constrained by existing legacy systems.

Existing component-based development (CBD) processes are largely manual and labour-intensive. The following steps are usually present in most CBD processes:

First, the system integrator creates an initial system *manifest*. The manifest defines:

- the system requirements, for example, that all components must be developed in Java or run on the Linux platform;

- the system context, including any components that may already be fixed in an existing (or legacy system); and
- the architecture for the new system, including any component patterns that are implemented, and the number and functionality of missing components.

Once the manifest is defined, the system integrator identifies components that meet the documented functional requirements and systems constraints. It is insufficient, however, to evaluate each of the individual software components in isolation as this does not guarantee that selected components are mutually compatible.

The system integrator must next discover an ensemble—a collection of compatible components—that satisfies the functional requirements. When multiple components are found that match the requirements the number of possible ensembles increases according to the factor of the cardinality of each set of qualified components. This situation deteriorates further when you consider that there may be multiple versions of each component.

Evaluating all possible component ensembles by hand is, of course, prohibitively expensive. In practice, system integrators must rely on experience to select components with a high degree of compatibility. The result is that the component space is largely unexplored, and the possibility that an optimal component ensemble has been selected is low.

Our proposed solution to this problem is to automate at least part of the process by which component ensembles consisting of compatible components are identified. It is anticipated that, by automating part of this process, a savings in evaluation and development costs can be achieved and that a greater percentage of the component space can be considered. We implemented a prototype system to test our solution called K-BACEE (pronounced kay-base) for Knowledge-Based Automated Component Ensemble Evaluation [16].

3 K-BACEE

K-BACEE is intended to be generally useful for a wide range of components, including EJB, COM, JavaBeans and commercial-off-the-shelf (COTS) products that do not conform to a standard component model. Unfortunately, the set of all possible components is an overly broad problem space that would make it expensive to prove (or disprove) the usefulness of the approach. Resultantly, we narrowed our area of interest to Enterprise JavaBeans to constrain the problem space and make it easier to produce a realistic prototype with a reasonable amount of effort.

Enterprise JavaBeans (EJB) technology defines a model for the development and deployment of reusable Java server components called enterprise beans. Enterprise beans are deployed in an EJB server/container that provides support for transactions, security, and other system services.

While reduced in scope, the EJB domain is still non-trivial. There are currently 167 different application servers listed on the JavaSoft Web site including products from BEA Systems, Borland, Forte, Gemstone, IBM, IONA, iPlanet, Oracle, Silverstream, and Sybase.

Figure 1 illustrates the K-BACEE architecture, consisting of a searchable repository of component specifications, integration rules, pattern database, query server, and component ensemble evaluator. System integrators provide a manifest that defines the functional requirements of the components as well as any overriding constraints on how the components are integrated.

3.1 Component Selection

As suggested in the introduction, K-BACEE is a component repository that is used at the *system*, rather than the component level. The heart of the system, however, is still the component repository, consisting of component specifications. This repository is searched using the manifest.

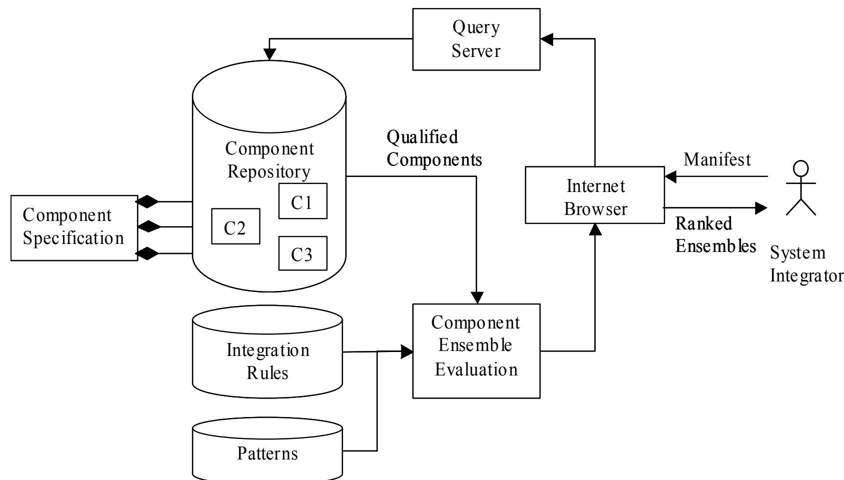

Figure 1: K-BACEE Architecture

XML (eXtensible Markup Language) is used to represent both the component and system requirement specifications. XML is a World Wide Web Consortium (W3C) recommendation that has become universally accepted as the standard for document interchange [1]. XML is well suited for this application as it provides a formal language for mapping attributes to values and is fully extensible [11], [3].

The manifest provides the initial context for evaluating components. Figure 2 shows a simplified manifest for a hypothetical system. This manifest defines constraints, possibly driven by corporate policy decisions, concerning the implementation language and GUI mechanism. The specification also includes a functional description of components required by the system.

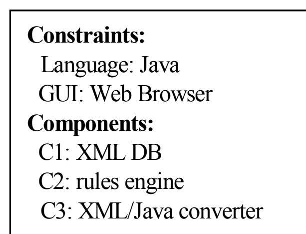

Figure 2: Manifest.

K-BACEE uses the manifest to identify an initial working set of qualified components. To identify all the components in the component repository that satisfy the manifest, we extract the set of constraints from the manifest and transform them into XML Query Language (XQL) [14] queries. XQL is one of several XML query language proposals; however, it has the support of several commercial products.

The manifest is converted into a series of queries on the component repository. Components that match the requirements specified in the manifest are selected from the component repository and placed into the working set. In Figure 3, we show candidate components that provide the functionality required by our example system. Component **`x1`**, **`x2`** and **`x3`** satisfy the functional requirements for the XML DB, component **`y1`** satisfies the functional requirements for a rules engine; **`z1`**, **`z2`** and **`z3`** satisfy the functional requirements for an XML/Java converter.

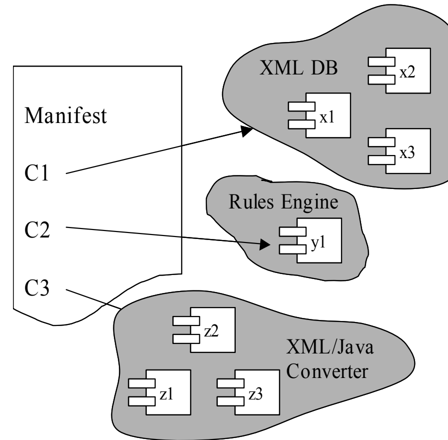

Figure 3: Component Selection

Once a working set of components has been identified, they must be grouped into possible ensembles. Based on the above example, there are a total of 9 possible ensembles (3 XML DBs x 1 Rules Engine x 3 XML/Java converters). These ensembles must now be ranked based on compatibility.

3.2 Ensemble Evaluation

Components in each ensemble are evaluated for compatibility based on their *attributes* (defined in the component specification) and patterns of interactions. A repository of software engineering integration rules is used to evaluate compatibility between components in an ensemble, and assign a numerical ranking to each ensemble.

Component attributes define characteristics that impact compatibility with other components, for example, the protocols supported by the component. Integration rules define how attributes affect component integration. These rules identify both those attribute combinations that simplify—and those that complicate—system integration.

The pattern database contains the reusable component interaction patterns that are used during ensemble evaluation to constrain the manner in which components are evaluated for compatibility.

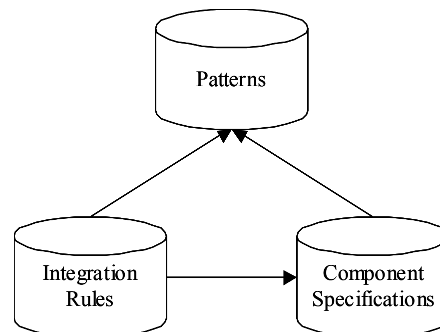

Figure 4: Database relationships.

Figure 4 illustrates the relationship between the three K-BACEE databases. Component specifications may reference patterns stored in the pattern database, indicating which role(s) they can implement in the pattern. Integration rules may refer to both component attributes stored in the component specification and patterns stored in the pattern database. For example, an integration rule may be defined which states that if a component conforming to a client role invokes a component conforming to a server role, and if the client is written in Java and the server is written in C++ (component attributes) then add 8 to the overall compatibility rating for the ensemble.

3.3 Database Extensibility

The component specification, integration rules and pattern databases are each fully extensible by system integrators, component vendors, and independent evaluators. This requires that K-BACEE provide mechanisms to add new integration rules, and to modify and delete existing rules in these databases.

The greatest problem in extending K-BACEE databases is conflict resolution. It is likely that component vendors and system integrators have a different perspective of the integration rule database, for example. A component vendor may easily take the view that a component is fully compatible with another component until proven otherwise, while a system integrator may take the more pragmatic "I'll believe it when I see it" perspective. K-BACEE must support a resolution process to eliminate conflicting integration rules.

We decided to implement a two-phase process to support conflict resolution. When a rule is submitted, it is evaluated to determine if it conflicts with an existing rule. If the rule does not conflict, it is assumed to be valid and added to the integration rules database. If the rule does conflict, email is sent to the person who submitted the new rule and the person submitting the original (conflicted) rule, asking them to resolve the conflict.[1] Either user is capable of removing their rule and resolving the conflict, or both users could remove the rule and replace it with an agreed upon rule. If the two parties cannot agree on a resolution, or if the conflict remains unresolved for an extended period of time, the K-BACEE system administrator resolves the conflict.

We examine each of the three K-BACEE databases in more detail in the following sections.

4 Component Repository

The component repository contains the descriptions of components that can be selected by K-BACEE to satisfy a requirement in a manifest and evaluated as part of an ensemble of components that satisfies (some portion of) the overall manifest. Component specifications consist of *attributes*, which are name/value pairs specified in XML. These attributes serve a variety of purposes. In particular they include information about functionality and behaviour useful in determining semantic compatibility, as well as attributes used to determine syntactic compatibility with other components, for example sharing the same deployment platform, and general information used to locate and purchase the component.

Defining a flat space of component attributes is somewhat unwieldy since there are many attributes that are specific to EJB, COM, or other component models. There are also attributes that are common to all components, including the general information. This combination of shared and specialled attributes is best represented in a class hierarchy. This hierarchy might consist, for example, of a `component` root node that can be specialized into `JavaComponent` that in turn

[1] This requires, of course, that email address and other contact information be collected from users updating information in any of the K-BACEE databases.

can be specialized to `JavaBeanComponent` and `EJBComponent`. The class hierarchy can be implemented using types derived by extension in the XML Schema definition language [17].

Table 1 identifies a number of attribute names that are used to describe Enterprise JavaBean components along with sample values. This list is (necessarily) incomplete but representative. These attribute names are established to provide a common vocabulary for both component developers and system integrators. In addition to these EJB specific attributes, the functionality of the component is also described by attributes as well as common interactions, yielding a behavioural profile of the component. Component interactions are further elaborated in the following section.

The first attribute listed is the component ***ID***. This is a string that uniquely identifies the component, and could be for example, a COM GUID. Component identifiers can be referenced from other component specifications—allowing a component's interface to be defined in terms of another component. For example, a component specification may claim that a component uses CORBA (Common Object Request Broker Architecture) as an integration mechanism and that it supports version 2.0 of IIOP (Internet Inter-ORB Protocol). However, differences in implementation, such as incompatibilities in the naming service may make it more difficult to integrate a component implemented using Inprise VisiBroker with one developed using IONA Orbix [15].

Attribute	**Description**	**Sample value(s)**
ID	Component identifier	01, 02
Name	Component name	General Ledger
Function	Component function described by keywords	accounts, double entry bookkeeping, journal entries, budgeting
Pattern (name, role)	Name of pattern and component role in pattern.	SessionFaçade, Business Entity
JDK (vendor, version)	Vendor and version of JDK	Sun, 1.2.1 IBM, 1.1.8
Platform	Tested environment.	Solaris, NT, Linux
Protocol (name, version)	Communication protocol	IIOP, 2.0 RMI, 1.1
Transaction attribute	Specifies how the container manages transaction boundaries when delegating a method invocation to an enterprise bean's business method.	NotSupported, Supports, Required, RequiresNew, Mandatory, Never
Persistence-type	Specifies an entity bean's persistence management type.	Bean, Container
EJB Server Container (vendor, product, version)	Container for which this component has been tested	IBM, WebSphere, 3.5.3

Table 1: EJB component attributes.

The ***name*** and ***function*** attributes listed in Table 1 are part of the general information used to identify and locate the product. General information also includes the product version and vendor information.

The ***pattern*** attribute is used to identify one or more design patterns in which the component may provide a role. The ***name*** subattribute identifies the pattern while the ***role*** subattribute identifies the role the component may provide in the pattern. The use of component patterns in K-BACEE is described in more detail in the following section.

The ***pattern*** attribute is an example of an attribute that can be repeated multiple times, in this case, for each different pattern supported. The ***role*** subattribute can also be repeated for each role a component may support in each pattern. This allows for components that are polymorphic or support multiple attributes.

The ***JDK*** attribute is specific to components written in the Java programming language. These might include JavaBeans and Enterprise JavaBeans. The ***JDK*** attribute is important in evaluating the compatibility of an ensemble. A system integrator has to decide in which environment a bean or bean client will run. An EJB server can support different types and/or versions of the runtime environment on both the client and server side. However, if the system consists of multiple enterprise beans (which is likely), and these enterprise beans were all required to run on the same server (also likely), then these beans must all run in the same environment, and this environment must be supported by the EJB server.

While not specific to Java applications, the ***platform*** attribute must often be considered along with the ***JDK*** attribute as EJB servers may support different versions of the JDK on different platforms.

The ***protocol*** attribute identifies protocols supported by the component that are critical for integration. For example, does the component communicate using RMI or IIOP? Which version of the protocol is used? This is a relatively general attribute, and may apply to components other than enterprise beans. It is also possible to include a reference to the component ID that implements the protocol as part of the attribute definition, as described earlier in this section. The ***protocol*** attribute should be repeated for each supported protocol.

The ***transaction*** attribute is specific to EJB components and an example of an attribute derived directly from the deployment descriptor. Not all fields in the deployment descriptor are necessarily included in the component specification for an EJB, but many of these fields provide information about the internal structure of the enterprise bean that is critical in determining integratability. For example, there are two cases when invoking an enterprise bean with the wrong transaction environment will generate an exception (when the transaction attribute is `mandatory` and the client *does not* specify a transaction and when the transaction attribute is `never` and the client *does* specify a transaction). In addition to these error conditions, it is easy to get unexpected or incorrect behaviour from mismatched assumptions concerning transactional behaviour. Providing information about the transactional nature of the enterprise beans in the component specification allows K-BACEE to reason about these properties while determining compatibility.

Persistence-type is another example of an attribute derived from the deployment descriptor. The persistence type can be either container managed or bean managed. In bean-managed persistence, the enterprise bean contains the logic to persist the bean state to permanent storage. In container-managed persistence, this logic is generated from the deployment descriptor. Bean-managed

persistence typically makes the bean more dependent on a particular data store while container-managed persistence makes the enterprise bean more dependent on a particular EJB container. Of course, attributes must be defined for both the data store and EJB container for K-BACEE to evaluate compatibility based on the persistence type.

Finally, the ***container*** identifies the vendor, product name and product version of the EJB container.[2] This field is used to indicate the EJB server platforms on which the enterprise bean has been tested. A separate "record" should be provided for each platform in which the enterprise bean has been tested.

There is also an alternative (and preferred form) for specifying the EJB container. This involves creating a component specification for the EJB container, and specifying the ID of this specification for this attribute.

Specifying the ID for an EJB container implies that EJB containers are handled in a similar manner to any other component. Although the EJB server could be assumed by K-BACEE when processing a specification that includes Enterprise JavaBeans, we decided that using the existing mechanism for specifying components would be preferable for the following reasons:

1. New Enterprise JavaBean containers and new versions of existing containers are continuing to be developed. Using the existing component specification mechanism allows these new products to be easily introduced to the system
2. EJB containers can be partially or fully specified in the manifest, allowing the system integrator to constrain this aspect of the system.
3. Existing EJB container specifications can be easily extended with new attributes, as new integration rules are discovered.

K-BACEE component is designed to be extensible and support the continual evolution of the system. Component properties may be initially assigned by the component developer, but may also be updated by system integrators and third party evaluators. There must, however, be a core set of ubiquitous attributes that supports comparison and evaluation by existing integration rules. We have begun in this paper and the K-BACEE prototype to define these attributes, but the final specification should be created by a group effort and ratified by an established standards body.

5 Component Patterns

We recognize that components are not always organized in a unique fashion but often follow common patterns of interaction. Thanks to work done by the Gang of Four [6] and others [2], [5] these patterns are more universally recognized and applied. As the use of design patterns becomes more pervasive, we should see more and more components designed to satisfy a defined role in an existing pattern.

5.1 Rationale

Design patterns are of interest to us as a means of improving our ensemble evaluation capability as they enable a description of component interaction, or more precisely, behavioural interaction between components. For example, if a given component assumes the role of "client" in a design pattern we can evaluate specific compatibility issues related to client/server communications. The assumption of a given role in a design pattern is also used as a criterion in the initial selection of

[2] Theoretically both an EJB-container and EJB-server attribute should be defined, but since most commercial implementations currently include both a server and container this is not really necessary in practice.

components, allowing K-BACEE to select a smaller number of better-qualified components for evaluation.

5.2 EJB Patterns

Sun Microsystems has defined (multiple) collections of design patterns for creating software systems using Enterprise JavaBeans and other J2EE technologies. One collection is Java 2 Platform Enterprise Edition Blueprints described in Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition by Kassem [10] and also on-line at the Sun Web site. The Sun Java Centre has also defined a series of design patterns also available on-line at the Sun Java Connection.

Figure 5 is a sequence diagram for the Session Façade design pattern defined by the Sun Java Centre. This design pattern uses a session bean as a façade to encapsulate the complexity of interactions between the business objects participating in a workflow. The session façade manages the business objects, and provides a uniform, coarse-grained service access layer to clients. The Session Façade design pattern defines relationships between five different components, several of which are implemented as either session or entity beans.

Patterns are maintained in K-BACEE in XML Metadata Interchange (XMI) format [19]. XMI enables easy interchange of metadata between modelling tools based on the Object Management Group (OMG) Unified Modelling Language (UML) and metadata repositories based on the OMG Meta Object Facility (MOF) in distributed heterogeneous environments.

The selection of XMI has immediate advantages. Design patterns can be entered using a Rational Rose and XMI generated using the XMI add-in. The XML representation of design patterns can then be managed within K-BACEE using the same tools and processes employed to manage the (XML-based) component-specifications.

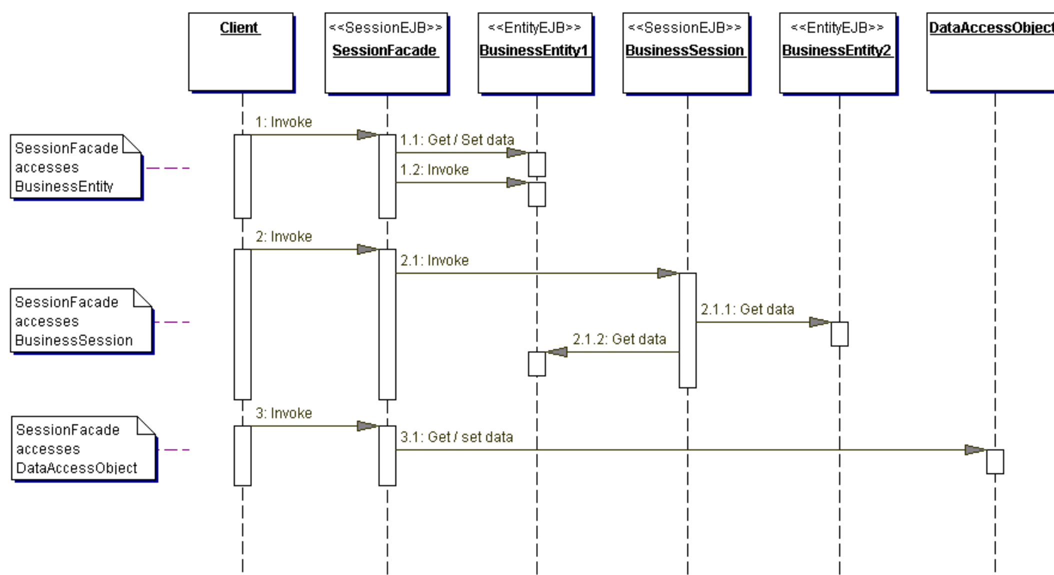

Figure 5: Session Façade design pattern.

New patterns of interaction are added to K-BACEE by the system integrator, as these patterns are primarily used to build their systems. In fact, by introducing a pattern into the pattern database a

system architect can foster reuse of the interaction as well as the components that implement roles in the pattern.

6 Integration Rules

Components in each ensemble are evaluated for compatibility based on a repository of software engineering integration rules. The component specification includes attributes that define characteristics of the component that impacts compatibility with other components, for example, the protocols supported. Integration rules define how attributes affect component integration. These rules identify both those attribute combinations that simplify—and those that complicate—system integration. It is important to note that it is the cumulative ensemble score

The EJB comparison project [4] evaluated four EJB application servers for compliance to the EJB 1.0 specification, benchmarking and interoperability: IBM WebSphere 3.0, GemStone/J 3.0 from GemStone Systems, Inc., Sun MicroSystem's NetDynamics 5.0.22 SP2, and BEA WebLogic 4.5.1.[3] Their findings, particularly from the interoperability tests, are useful for identifying attributes that affect the integratability of EJB components.

One issue, for example, is the version of the JDK supported by the EJB server. An EJB server can support different versions and/or types of the runtime environment on both the client and the server side. IBM WebSphere 3.0, for example, supports JDK 1.1.7B, JDK 1.2.2, and MS SDK for Java 3.2 on WinNT platforms as the client and server-side JDKs.

We can now derive a series of integration rules based on this information. If two components did not share the same JDK we would consider them less compatible with respect to this attribute that two components which did share the same JDK. Figure 6 shows two sample integration rules for scoring ensembles based upon the JDK version compatibility of components within the ensemble. If the system integrator sets an constraint in the manifest stating that the EJB server must be IBM WebSphere 3.0 we would have an additional factor to consider while evaluating compatibility: We would now need to determine if the components were compatible with IBM WebSphere 3.0 and if the components were compatible with each other.

```
when (comp1->JDK version  = "1.3" and  comp2->JDK version = "1.3" ) then
        Ensemble_score += 10;
when (comp1->JDK version  = "1.3" and  comp2->JDK version = "1.2" ) then
        Ensemble_score += 7;
```

Figure 6: Integration rule for JDK version compatibility.

In some ways, these rules are overly pessimistic (that is, they would result in a lower overall ranking than is warranted in some cases). If the manifest were again extended to include a constraint that the solution implement the Session Façade design pattern described in Section 5.2 and that one of the identified components implements the `SessionFaçade` role in the pattern while the other component implements the `Client` role. As a result, we know that the existing

---

[3] While most (if not all) of these product versions have been superseded by newer releases, data collected about them is not at all irrelevant for our purposes. Instances of these product versions will stay deployed for many years, as well as the components that have been developed and tested for these products. To be effective, K-BACEE repositories must span multiple versions of products over multiple years to accurately describe the marketplace and encompass legacy system modernization efforts.

integration rule is unnecessarily pessimistic because the architecture of the EJB server allows us to use different versions of the JDK on the client and server side.

Apparent in this discussion is that different rules can apply depending on how much information is provided. Depending on the rule-based system used, this problem can be resolved in two ways. The first is by establishing precedence of rules. In general, more specific rules would hold precedence over more general rules. Therefore, in the last case, when the EJB server and the design pattern are both defined, the most specific rule (the one that assumes all these variables are set) would be applied. When only the component JDKs are specified the more general rule is applied.

There are also cases when two components that are known to be compatible may score lower than expected because of apparent architectural mismatch (or incompatible components score higher than expected because of hidden problems). This can be corrected, using existing K-BACEE mechanisms, by specifying integration rules that reference the component ***ID*** attributes of the components and directly add (or remove) points based on the combination of the specific components involved. It is important to note that the ranking of an ensemble is the result of the aggregate, cumulative score of all rules executed.

Integration rules typically reflect known compatibilities and incompatibilities between attributes. The discovery and refinement of these rules is a normal part of the system integration process. System integrators extend the integration rule database from lessons learned, overtime improving the reliability of the knowledge base.

7 Summary and Conclusions

Prototyping efforts for K-BACEE have demonstrated that the automated evaluation of component ensembles is feasible and successful for small test cases. The initial prototype also proved helpful by highlighting the impact of standards on component compatibility. In particular, two components used in the construction of K-BACEE, each supporting the XML Schema Structures standard, did so to different levels of compliance, necessitating additional development effort to resolve the mismatch. From this experience, we are generalizing the impact of standards on compatibility as interactions based upon communication protocol and data standards play an increasingly import role in systems integration activities.

Future goals include scaling the application of K-BACEE technology to provide a generalized, Web-based system that can be used, for example, by component brokers to market components, component providers to provide a distribution channel and system integrators to identify compatible component ensembles.

8 Acknowledgements

The authors would like to acknowledge the help of David Mundie in some of the early prototyping efforts, and the support of John Foreman and Yunyong Teng-amnuay. Thanks also to Santiago Comella-Dorda and Dan Plakosh for their valuable and useful comments.

9 References

[1] Bray, Tim; Paoli, Jean; & Sperberg-McQueen, C.M. Extensible Markup Language (XML) 1.0. W3C Recommendation, February 10, 1998.

[2] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter and Stal, Michael. "Pattern-Oriented Software Architecture-A System of patterns.", Wiley Press, 1996-2000

[3] Cover, Robin. "Literate Programming with SGML and XML." The XML Cover Pages. http://www.oasis-open.org/cover/xmlLitProg.html.

[4] EJB Comparison Project, MLC System GmbH, Distributed Systems Research Group, Charles University, January 12, 2000.

[5] Fowler, Martin. "Analysis Patterns: Reusable Object Models." Addison Wesley, 1997.

[6] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. "Design Patterns, Elements of Object-Oriented Software.", Addison-Wesley, 1995.

[7] M. R. Girardi and B. Ibrahim, "New Approaches for Reuse Systems," Position Paper Collection of the 2nd. International Workshop on Software Reuse, E. Guerrieri ed., March 24-26, 1993.

[8] Hansen, W.J.; Foreman, J.T.; Carney, D.J.; Forrester, E.C.; Graettinger, C.P.; Peterson, W.C.; & Place, P.R. Spiral Development–Building the Culture: A Report on the CSE-SEI Workshop (CMU/SEI-2000-SR-006, ADA382585). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 2000. http://www.sei.cmu.edu /pub/documents/00.reports/pdf/00sr006.pdf

[9] Henninger, S., "Supporting the construction and evolution of component repositories," Proceedings of the 18th International Conference on Software Engineering, 1996.

[10] Kassem, Nicholas "Designing Enterprise Applications with the Java 2 Platform." Enterprise Edition, Addison-Wesley, ISBN 0-20-1702770.

[11] Mundie, David. "Standardized Data Representations for Software Testing." Pacific Northwest Conference on Software Quality, Portland, Maine, October 1997.

[12] President's Information Technology Advisory Committee Interim Report to the President, National Coordination Centre for Computing, Information, and Communications, Arlington, VA, August 1998.

[13] Poulin, J.S., "Populating Software Repositories: Incentives and Domain-Specific Software", The Journal of Systems and Software 30(3), Elsevier Science, New York, NY, September 1995.

[14] Robie, Jonathan. "XQL (XML Query Language)," August 1999. http://www.ibiblio.org/xql/xql-proposal.html.

[15] Seacord, Robert C.; Wallnau, Kurt; John, Robert; Comella-Dorda, Santiago; & Hissam, Scott A. "Custom vs. Off-the-Shelf Architecture." Proceedings of the 3rd International Enterprise Distributed Object Computing Conference. Mannheim, Germany, September 27-30, 1999.

[16] Seacord, Robert C.; Mundie, David; Boonsiri, Somjai. "K-BACEE: Knowledge-Based Automated Component Ensemble Evaluation", published in proceedings of the 2001

Workshop on Component-Based Software Engineering held in conjunction with the 27th Euromicro Conference, Warsaw, Poland, September 4th – 6th, IEEE Computer Society.

[17] Thompson, Henry S.; Beech, David; Murray; Mendelsohn, Noah. XML Schema Part 1: Structures W3C Recommendation 2 May 2001, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[18] Wallnau, Kurt; Hissam, Scott; Seacord, Robert. Building Systems from Commercial Components, Addison-Wesley, June 2001, ISBN: 0201700646.

[19] XML Metadata Interchange (XMI) Version 1.1, OMG Document ad/99-10-02, October 25, 1999.