# Automatic Validation of Deployed J2EE Components Using Aspects

John Grundy[1, 2] and Guoliang Ding[1]

[1]Department of Computer Science and [2]Department of Electrical and Electronic Engineering,
University of Auckland, Private Bag 92019, Auckland, New Zealand
john-g@cs.auckland.ac.nz

## Abstract

*Validating that software components meet their requirements under a particular deployment scenario is very challenging. We describe a new approach that uses component aspects, describing functional and non-functional cross-cutting concerns impacting components, to perform automated deployed component validation. Aspect information associated with J2EE component implementations is inspected after component deployment by validation agents. These agents run automated tests to determine if the deployed components meet their aspect-described requirements. We describe the way component aspects are encoded, the automated agent-based testing process we employ, and our validation agent architecture and implementation.*

**Keywords**: software component validation, automated testing, component characterisation, validation agents

## 1. Introduction

Component-based software development uses the approach of composition of self-describing, reusable and tailorable units of functionality (components) [31, 4, 6]. During component design, implementation and testing developers need to ensure both individual components and groups of composed components meet the component users' functional and non-functional requirements. In order to ensure components meet these constraints, developers can carry out three kinds of quality assurance: design-time constraint reasoning, implementation-time component testing, and deployed component composition and configuration validation. Unfortunately while the first two approaches ideally allow developers to demonstrate components meet specifications, they have key problems.

Many researchers have investigated approaches to describing software components formally [6, 11, 25, 29]. These theoretically allow design-time verification of component characteristics and therefore verification that groups of composed components will meet system requirements. Similarly researchers have developed various tools to assist verifying an implemented component meets its desired requirements constraints [2, 15, 23]. Design-time validation suffers from the problem that components run on un-verified hardware, networks, operating systems and in conjunction with unverified third-party COTS components [20, 21]. Implementation-time testing may verify components meet constraints in "laboratory situations" but not necessarily that they do when deployed with other, often 3rd party components.

We have developed a new method for developing software components called Aspect-oriented Component Engineering (AOCE) [6]. This technique uses "aspects", or cross-cutting concerns, to characterise software component constraints. In this paper we describe new work we have done developing "validation agents" that use aspect characterisations to verify software components meet constraints in actual deployment situations. We motivate this work and survey existing component validation approaches. We then describe the AOCE method and illustrate how component implementations have aspect characterisations associated with them. We describe the architecture and implementation of our agents, illustrating their use with some examples. We conclude with a discussion of experiences and directions for future work.

## 2. Motivation

Figure 1 (a) shows one of the user interfaces from a component-based software application, an on-line furniture store. Some of the software components making up this application are shown in Figure 1 (b). These include Java Server Pages providing web-based user interfaces (including product searching, login, a shopping cart and groupware messaging support); thick-client data maintenance applications (e.g. for maintaining staff, customer, product, catalogue, and order data); server-side Enterprise Java Bean components implementing business logic (product catalogue, search engine, order processing, and messaging) and enterprise data management (orders, products, staff, customers, and messages). Some

middleware, database, security and GUI components are also used.

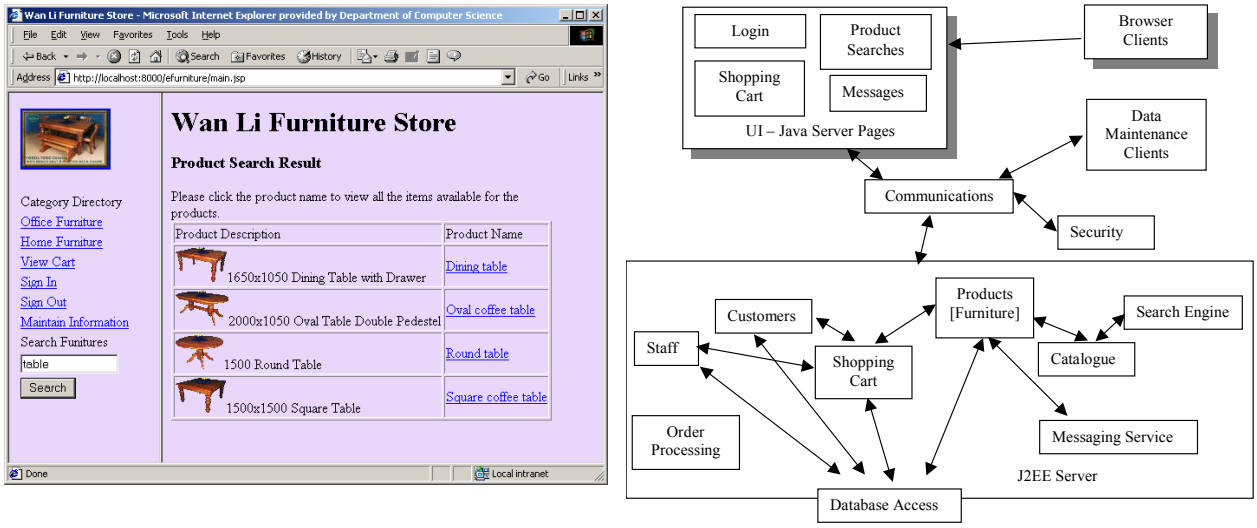


**Figure 1. (a) Example component-based system and (b) a J2EE-based architecture for this system.**

Many of these components can be reused in other applications, either similar on-line business-to-customer applications (e.g. the shopping cart, catalogue, and customer sales tracking) or a wider range of multi-tier applications (customer, staff and order information, groupware messaging client and server support, middleware and data persistency support). Many of these components can be deployed with different 3rd party components, middleware, and component versions of e.g. J2EE servers, database servers and so on, from those the original designers and implementers planned or envisaged.

Both the individual components and the components they are deployed with to form an application have various functional and non-functional constraints that typically "cross-cut" the components that make up the system:

- Performance – speed for component method(s) to complete; required number of concurrent users; data transfer rates and so on. Performance is notoriously difficult to predict [9, 17].
- Resource usage – memory and disk space consumption, network bandwidth, web server CPU time and so on. Groups of components can adversely affect each other's performance and resource utilisation in very unpredictable ways [17].
- Transaction support – transaction recovery, concurrency control, distributed transaction support and so on.
- Security – authentication of users and client hosts, access control to server-side functions, data encryption and decryption and so on.
- Data persistency and distribution – data storage, location and retrieval, storage mechanism (text, binary, XML, multimedia), data transmission, distributed event subscription and notification, and so on.

These issues cross-cut software components in multiple places and in ways orthogonal to the functional decomposition of the application typically used by most component (and object) design approaches [32, 24, 6]. Validating that component designs and compositions meet overall system constraints, and that individual component constraints are satisfied in a given deployment situation, is very difficult [9, 7]. Our experience with AOCE has shown some design-time and implementation-time validation can be done, but this must be complemented by deployment-time validation [8].

Most component design approaches, such as Catalysis™, COMO and SelectPerspective™ [1, 4], do not capture the cross-cutting concerns impacting components adequately [13, 8]. This means it is difficult to reason about component compositions at design-time in terms of their overall constraints and whether these will likely be met. Our own aspect-oriented component engineering method provides much richer component description, including aspects capturing the above cross-cutting issues, but similarly cannot adequately support component validation at design-time. Most other component description techniques focus on formally specifying component functional properties [3, 11, 29, 25, 30]. Few support non-functional constraint specification though component "trust" and some specialised non-functional requirement description techniques, such as security, have been developed [11, 19, 18, 2]. All of these support a degree of design-time composition reasoning. However none can adequately validate component deployment

compositions due to inability to specify 3rd party component, hardware and operating system characteristics.

Various approaches to component testing, middleware performance evaluation and deployed component validation have been investigated. Most component testing approaches and tools apply tests to individual or small groups of components under "laboratory conditions", not usually realistic deployment situations [15, 21, 23, 14, 10]. Some approaches use extracted component meta-data and wrappers to formulate automated component tests [26, 21], but these techniques tend to only focus on functional component validation, not validating non-functional constraints. Middleware performance testing work typically focuses on only limited aspects of component functional and non-functional requirements [9, 22, 7]. In addition, many of these approaches also suffer from a high degree of manual effort on the part of developers to build test harnesses with which to evaluate their components and middleware [5, 7, 9, 14]. Most current deployed component validation approaches also typically require extensive test bed prototyping to evaluate components [16, 9]. A major problem with these dynamic component testing approaches is that the components under test have poor self-description in terms of the required component functional and non-functional characteristics [8]. This makes automated component validation very difficult.

## 3. Our Approach

In order to adequately support automated deployed software component validation we need to:

- Encode information about cross-cutting component characteristics and constraints in a unified manner. We use "component aspects" to do this.
- Allow this information to be accessed at run-time by validation agents. We encode this component aspect information using XML documents.
- Run extensive, realistic tests on components in their desired deployment situation. We use queried aspect information to both configure component testing and to allow validation agents to act as test oracles, validating component behaviour meets constraints.
- Leverage existing testing frameworks and tools. Some of our validation agents make use of our SoftArch/MTE performance test-bed generator to generate realistic component loading tests.

Aspect-oriented component engineering is an extension to component engineering methods that focus on decomposition of system requirements into software component divisions of responsibility along functional lines. Component *aspects* represent cross-cutting concerns that impact on components in terms of the systemic services a component requires in order to operate or provides to other components in a system. Typical component aspects we use include user interface, distribution, persistency, security, transaction processing, component configuration and co-operative work support facilities (though many others are also possible [6]). Each component in a system is impacted by one or more of these systemic aspects - they either provide services to the system as a whole in this category or require them from other components. Each aspect category is divided into a number of *aspect details*, each of which may be *provided* or *required* by a software component. For example, a "shopping cart" component might provide a user interface window and menu bar, a data structure, transactional support, and event generation, but might require data persistency, event transmission and security management.

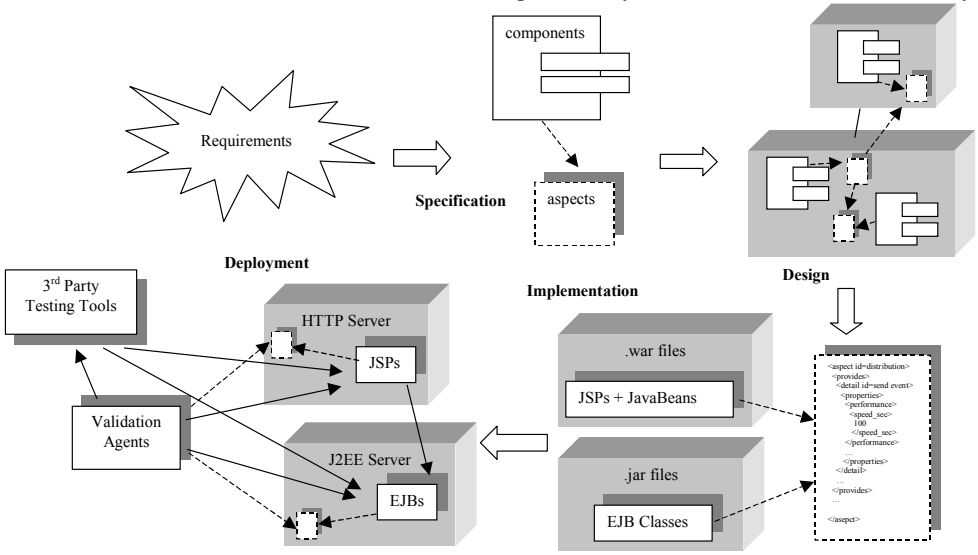


**Figure 2. Using aspects to ensure component-based system properties are met.**

Provided and required component aspect details have *properties* that further specify/constrain them e.g. minimum rate of data transmission provided or required; data indexing scheme and functions; whether parts of interface can be extended by other components; and so on. Aspect details typically characterise the systemic provided or required services (functional behaviour) impacting parts of a software component (i.e. that it provides or requires), whereas aspect detail properties tend to characterise non-functional constraints on these provided/required services (though this is not a hard rule) [6, 8].

Figure 2 illustrates how we make use of component aspects to validate deployed software components. Candidate component specification are identified and each of these components has a number of component aspect details they provide or require. Refinement of component specifications into detailed designs identifies implementation approaches to be followed in order to realise components. At design-time inter-component aspect relationships can be reasoned about. For example, consider a component requiring event broadcasting services (functional, Distribution aspect detail) that must support 100 events per second over a local area network (non-functional aspect detail property constraint). To be a valid component configuration one or more other components providing such services and meeting the performance constraints must be composed with this component.

We have developed a J2EE component implementation method that uses aspect-oriented component designs to guide component implementation maximising reusability and configuration [8]. As part of this process, developers encode component aspect information in XML documents. After component deployment this information is accessible for a number of purposes: other components introspect aspect information and perform automatic re-configuration; automated indexing of components for a repository; and allowing users to view component aspect information. In the work described in this paper validation agents also use this aspect information. It allows them to determine required component functional and non-functional characteristics and to perform tests on deployed components to determine if these are met. Our validation agents obtain component aspect information encoded in XML, determine appropriate tests that need to be run, then run these tests. Developers are informed of any non-working functional services or non-functional constraint violations. Some agents run test themselves while others use 3rd party testing tools to run required tests.

## 4. Component Characterisation

After designing software components using aspects a developer implements them using an appropriate technology. In this work we focus on using Java 2 Enterprise Edition (J2EE) implemented components [28], though the concepts are applicable to CORBA, .NET and COM+ implemented components. The J2EE architecture is illustrated in Figure 3, along with how we manage aspect encodings for J2EE components. The key components in the J2EE software architecture include clients (thin clients via browsers and thick clients via applets and applications); middle-tier web servers (comprised of Java Server Page (JSP); JavaBean and Servlet "components"); enterprise servers (comprised of Enterprise JavaBean (EJB) components); and databases. Thin clients are rendered HTML presented to users in web browsers. Thick clients incorporate Swing JavaBean GUI components and possibly data structure and other components. JSPs and Servlets are not strictly components but we treat them as such – they can be self-describing (via aspects) and dynamically deployed. These may make use of JavaBeans, simple software components that run in the hosting web server. Enterprise JavaBeans are a component model incorporating enterprise business logic ("Session Beans") and data management ("Entity Beans"), and are hosted by EJB containers and servers (providing persistency, transaction, security and resource management support).

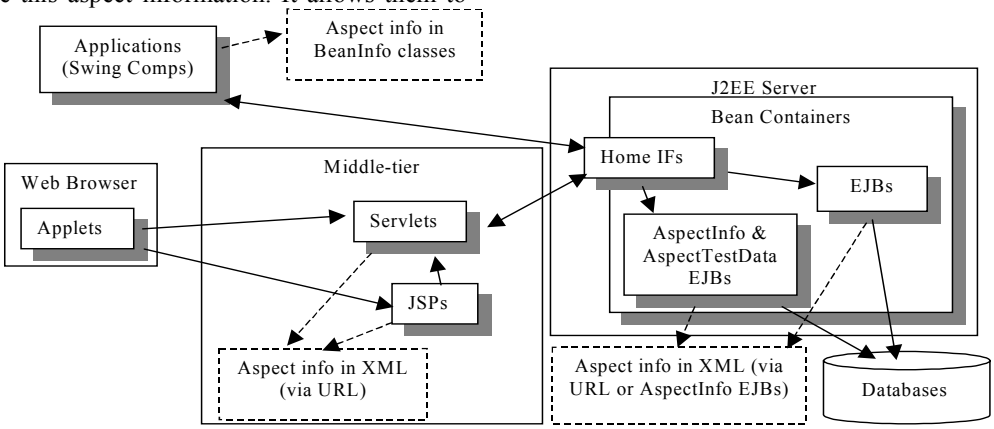


**Figure 3. J2EE component-based architecture with aspects.**

We have used the J2EE architecture to incorporate aspect descriptions for EJBs and JSPs Each JSP and Servlet hosted by the web server have an XML document describing the component's aspect information, accessible via a URL. EJB aspect information is accessed via a URL or by meta-data management AspectInfo entity EJBs deployed with the EJBs being described. Test data for use by validation agents is provided by AspectTestData EJBs.

Component aspect information is encoded using XML documents. We chose to use XML as it is flexible, can be straightforwardly stored and processed using a range of technologies and can be flexibly queried via Xpath-implemented queries. Aspect encodings describe systemic services provided and required by components and describe constraints on these services. Aspect information is consistency checked by a CASE tool [8].

Part of our Document Type Definition (DTD) describing component aspect XML document structure is illustrated in Figure 4 (a). An aspect specification is for a specified component or group of components. Group, or "aggregate" aspects, allow functional and non-functional characteristics to be specified for a set of composed components. A component's specifications are aspects, aspect details and aspect detail properties describing cross-cutting systemic concerns impacting the component's methods and state. Details are provided or required, and properties include types and expressions constraining the property's value(s). In addition, "validation" methods and URLs can be specified for components. These allow validation agents to construct and invoke method and/or URL calls to test the component at run-time. While some standard EJB and JSP method/URL calls are built-in to different validation agents, some must be component-specific and make use of deployment-specific example data. Such parameter values for EJB method calls and argument values for JSP page POSTs are accessed from named AspectTestData EJBs, along with expected method return values and a subset of the expected HTML result data values.

Figure 4 (b) shows parts of two components' XML-encoded aspect information, the first for a Staff Entity EJB (for maintaining Staff information) and the second for a Product Search JSP. The EJB has a persistency aspect detail describing the data storage support the EJB provides, and one describing the write-data support it needs to provide this Staff information data storage. The provided StoreData functionality cross-cuts the ejbCreate and ejbStore methods. A detail property specifies that storing the component state data should take less than 50 milliseconds. A component method call is specified that can be used by validation agents when verifying this. A JSP data provision detail (provided distribution aspect) specifies a maximum number of concurrent users the JSP must support. Another data output detail (provided user interface aspect) specifies the user response time and specifies a URL call and EJB to access example test data from. This will be used to formulate a URL with argument values specific to the JSP's deployment scenario (in this case the furniture store). The EJB can also provide some expected output data for agents to verify – method return values or example data in the returned HTML text.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT (Component|ComponentGroup)+>
<!ELEMENT Component (CompName, MappingName, CompProperties,
  CompMethods, CompEvents, CompAspects)>
...
<!ELEMENT Aspects (Aspect)+>
<!ELEMENT Aspect (AspectName,Details)>
<!ELEMENT Details (Detail)+>
<!ELEMENT Detail (DetailName, DetailType, Provided,
  DetailProperties, ImpactedMethods, DetailInfo)>
...
<!ELEMENT DetailProperty (DetailPropName, DetailPropType,
  DetailPropConstraint, DetailTestMethods, DetailPropInfo)>
...
<!ELEMENT DetailPropConstraint Expr>
...
<!ELEMENT DetailTestMethod (MethodCall|URLCall)>
<!ELEMENT MethodCall (MethodName, MethodArgumentData)>
<!ELEMENT URLCall (URLName, URLArgumentData)>
```

(a) DTD for component aspect information.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Component SYSTEM "componentaspects.dtd">
<Component CompName="StaffEJB">
  <MappingName>java:comp/env/ejb/staff</MappingName>
  <Aspects>
   <Aspect AspectName="Persistency">
    <Detail DetailName="Store" DetailType="StoreData" Provided='true'>
     < ImpactedMethod Name='ejbCreate' /><ImpactedMethod Name='ejbStore' />
     <DetailProperties>
      <DetailProperty DetailPropName="StoreSpeed" DetailType=" ResponseTime">
       <DetailPropType>Milliseconds</DetailPropType>
       <DetailPropConstraint><Expr><LessThan>50</LessThan>...
       <DetailTestMethods>
         <DetailTestMethod MethodName="ejbStore"
            MethodArgumentData=' java:comp/env/ejb/staff_testdata'>
    </DetailProperties>
    </Detail>
    <Detail DetailName="Write" DetailType="WriteData" Provided='false'>...
    </Aspect>...
</Component>

<Component CompName="ProductSearch.jsp">
  <MappingName>jsp/furniture/ProductSearch.jsp</MappingName>
  <Aspects>
  <Aspect AspectName="Distribution">
   ... <Detail DetailName="DataProvision" DetailType="DataOutput" Provided='true' ...>
       ... <DetailProperty DetailPropName="MaxUsers" ...>
           ... <DetailPropConstraint><FixedValue>50</FixedValue>
  <Aspect AspectName="UserInterface" ...>
   ... <Detail DetailName="PostData" DetailType="DataOutput" Provided='true' ...>
      ... <DetailProperty DetailPropName="ResponseTime" ...>
        .... <DetailPropConstraint><Expr><LessThan>2500</LessThan>...
            <DetailTestMethod><URLCall URLName="ProductSearch.jsp"
              URLArgumentData= java:comp/env/ejb/ProductSearch_testdata' />
```

(b) Examples of component aspect descriptions.

**Figure 4. Encoding component aspects with XML.**

## 5. Run-time Validation Agents

After deploying J2EE components we use a set of "validation agents" to determine whether the components' aspect-encoded requirements are met in their current deployment and configuration scenarios. Figure 5 illustrates the process by which deployed J2EE components are validated using this technique.

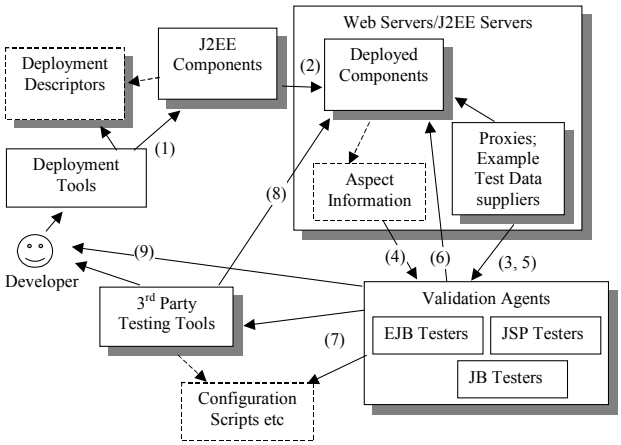


**Figure 5. Our agent-based validation process.**

A developer deploys J2EE components using a deployment tool (1), which reads the component deployment descriptor to properly configure the component. The deployment tool loads the component into a J2EE or web server, configures it and registers it for use (2). Validation agents are informed of the new component configuration by the developer or proxies in the J2EE server (3). A validation agent reads the component aspect information (4) using a URL, AspectInfo or BeanInfo objects to obtain the XML-encoded component aspect information. It queries the aspects using Xpath queries to extract required information for test generation. Test data for URL or method call arguments and expected result values are obtained from AspectTestData objects (5). Some validation agents carry out tests on the deployed component themselves (6) e.g. those checking conformance to specified service provision/requirement. Others configure 3rd party testing tools (7) to perform validation tests on the component (8). Validation results are presented to the developer (9) by validation agents and/or 3rd party testing tools.

Our component validation agents come in three main flavours: agents that perform simple validation tests; agents that deploy other agents to perform tests; and agents that configure and deploy third party testing tools to validate deployed components. We have built simple testing agents that check single-client response time, basic functional operation and data storage/retrieval of single component methods (EJBs) or pages (JSPs). We have built validation agents that deploy other agents to test transaction support, data loading (size) support and security (authentication and encryption) of EJBs, and resource usage and reliability of JSPs and EJBs. Separate agents are used that can be authenticated as specific clients, can generate large data sets for testing, and can participate in distributed transactions, all co-ordinated by another validation agent. We have also developed agents that make use of the SoftArch/MTE performance test-bed generator to test maximum user loading, required response times and transaction processing time for realistically loaded EJB and JSP components. We utilised this separate, 3rd party testing tool to avoid our component validation agents having to generate complex performance test beds themselves. Validation agents typically run independently and non-concurrently to avoid each others' test cases interfering.

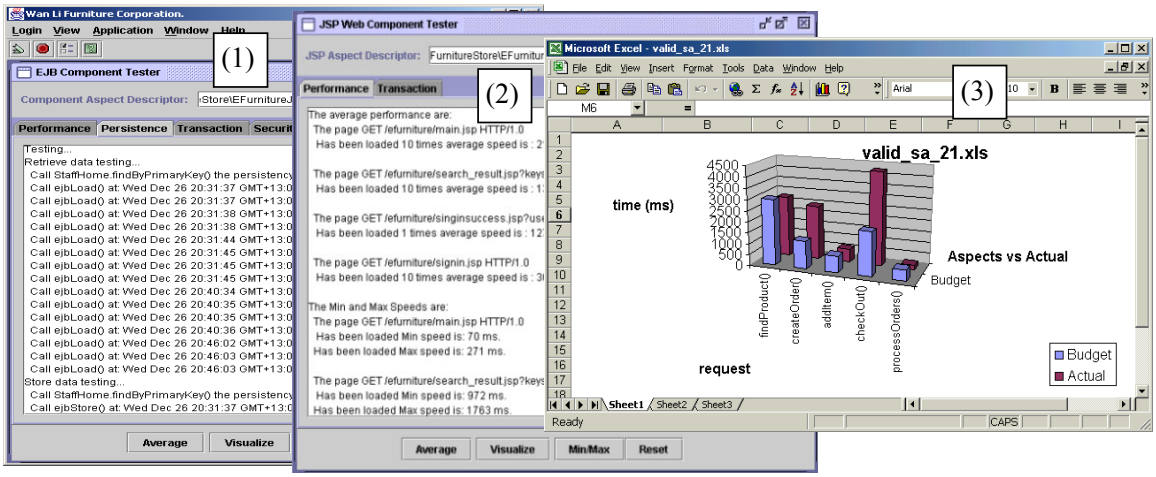


**Figure 6. Examples of validation agent output.**

Figure 6 shows three examples of component validation agent feedback to developers. (1) shows output from an entity bean EJB persistency testing validation agent. The agent extracts persistency aspect information defined for a component and ensures the component's state can be stored and retrieved appropriately in its current deployment situation. It does this by modifying the component's state, storing it, retrieving it and validating it. (2) shows output from a JSP validation agent that tests both posting to a JSP, retrieving JSP output and testing the response time of a JSP, all validated against constraints specified for the JSP component in its aspect information. (3) shows an MS Excel™ chart graphing the performance of some EJB component methods under a realistic loading test. This validation agent deployed the SoftArch/MTE test bed generator to generate and run this loading test with multiple clients. It used Excel to graph the results obtained from SoftArch/MTE performance tests against the required performance measures expressed in the component's aspect information.

As the component configuration of a system evolves e.g. new components are deployed or existing components re-configured, these deployed components must be re-validated. Our validation agents detect changes to component configurations and re-run tests as necessary to re-validate changing configurations. This approach could be extended to proactively validating systems with highly dynamic architectures, allowing system co-ordination components to use validation agents to determine if evolving configurations are valid.

## 6. Architecture and Implementation

Our validation agents are J2EE components themselves that share a common set of classes supporting aspect querying and result reporting. We have developed three kinds of validation agents to date: agents that perform tests themselves; agents that co-ordinate the running of tests by other agents; and agents that configure and run 3rd party testing tools. Each kind of agent generates test cases from queried aspect information and then runs these tests. Test results are compared to aspect-specified constraints against services to determine if a component deployment is valid.

Figure 7 shows how an EJB persistency validation agent tests if a component's persistency services function. In this example a shopping cart session bean EJB makes use of product and order entity beans. One entity bean uses EJB container persistency services, the other a relational database API directly. The persistency validation agent reads the order component aspects ad queries them to extract aspect detail information about persistency requirements from the components' aspect information (1): this includes methods that store data, component state data to try and store, and methods to test restored component state. The agent then formulates requests on the component to create, update, store, retrieve the component state. Many of these requests are generic for entity bean EJBs, though complex queries and updates can be encoded in the persistency aspect information to allow the validation agent to perform a wide range of tests. Example data for method (and URL) invocation is obtained from AspectTestData components (2). This can be tailored to be suitable for different deployment situations of the components under test. Invocations of methods and URLs (3) to test the component are then performed by the validation agents and the results analysed. Results of method invocations are compared to expected results from the AspectTestData components. The contents of the HTML returned by a URL invocation is parsed and example data items from the AspectTestData components are searched for in this returned HTML.

We have implemented several validation agents that perform basic tests on deployed components to ensure the component functions work in their deployment context. These agents are implemented as JavaBean components that can be run in Java applications or themselves deployed with components under test. All of our agents provide both a GUI interface to display information to developers and an API to allow other validation agents to deploy them and to gather test results from them.

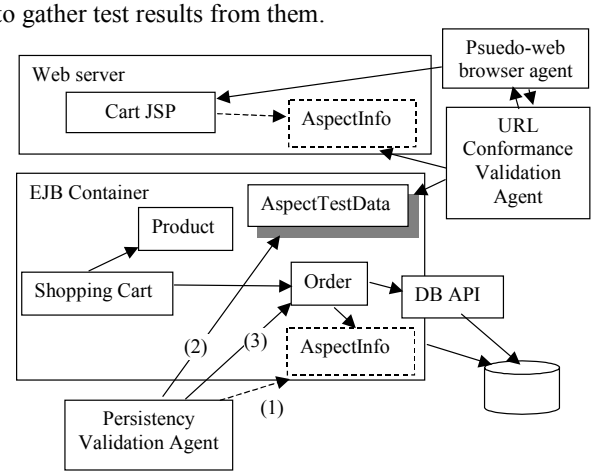


**Figure 7. Example of simple validation agents.**

Some tests require one agent to configure and deploy one or more other agents, for example transaction consistency testing, security testing and resource usage testing. This is because transactions and security need the testing agent to participate with the components under test e.g. be in its transaction or be authenticated as a specific client. Both of the validation agents compose method and URL calls using aspect information and example test data supplied by AspectTestData EJBs. They both try valid and invalid transactions (commit, rollback and raise exceptions) and security access (valid and invalid user and access to remote object functions). Resource usage testing e.g. memory, disk, CPU and so on, requires validation

agents to monitor system state vital signs as components are tested by other validation agents. As an example, in Figure 7 a JSP conformance validation agent determines the range of arguments a shopping cart JSP page can take to perform different functions. It then deploys a simple pseudo-web browser agent to interact with the JSP, as if it were a customer's web browser, to check the JSP works.

In the example from Figure 7, the persistency validation agent locates persistency aspect information for the Order entity EJB component. It extracts all specified persistency-impacting methods encoded in the component's aspect information. For each of these methods it obtains example order data values and method argument values to set from a specified AspectTestData EJB and sets the Order object's attributes to these values. It then constructs method calls and invokes each of the Order EJB persistency methods to store the data, change the data and reload the data. It then checks that the reloaded Order component attribute values (state) equal those expected.

The conformance validation agent locates user interface aspect information for the CartJSP Java Server Page component. It extracts all URL requests encoded in the component's aspect information. For each of these URL requests it obtains example URL argument data values from a specified AspectTestData EJB. It then constructs URL requests and invokes each of the CartJSP requests. It obtains expected response content data values from the AspectTestData EJB and checks to see that this data is contained in the HTML returned from the URL request.

Some kinds of deployed component validation require quite sophisticated testing. For example, to determine if a deployed component's response time (performance) will be adequate under "realistic" client, server and network loading we need to run "realistic" loading tests. We have developed a performance test bed generator, SoftArch/MTE, that allows software architects to generate realistic performance testing frameworks from high-level architecture descriptions [7]. We have reused this system to allow validation agents to configure SoftArch/MTE to run realistic loading tests for deployed J2EE components.

Figure 8 illustrates this process. A validation agent wanting to determine if deployed component response times to requests will meet aspect-encoded requirements generates an XML-encoded software architecture description (1). This includes references to the deployed component name/URL (for look-up) and methods (if an EJB) or arguments (if a JSP), along with test harness client and server applications, objects, services, requests, middleware and database table information. SoftArch/MTE takes this architecture configuration and generates source code and deployment scripts sufficient to automatically run realistic performance tests on the architecture [7] (2). However, unlike our previous work with SoftArch/MTE, this code now incorporates references to real, deployed software components rather than only automatically generated test bed code. The generated code is compiled then deployed by SoftArch/MTE to multiple available client and server host machines (3). A deployment agent initialises clients and servers (4) and then performance tests are run, results being captured in text files (5). The performance validation agent retrieves the results (6)and visualises them for the developer (7).

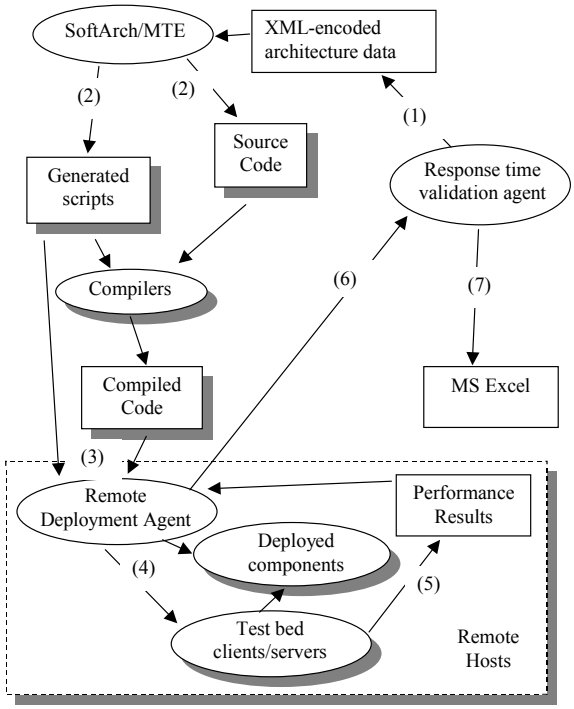


**Figure 8. Example of a using SoftArch/MTE.**

## 7. Evaluation

Our aspect-oriented component characterisation technique supports both functional and non-functional provided and required service specification. This is richer than most other component characterisation techniques we have found [13, 26, 29] and is sufficient to automate a range of quite different component validations. As long as sufficient detail is provided to allow agents to generate required test cases then many kinds of validation are automatable. The functional component service characterisations are similar to those of other techniques in terms of identifying methods and types but we group these by the aspects that impact this service (whether provided or required by the component). This has the advantage of identifying agents that make use of particular aspect detail or aspect detail property constraints and ensuring sufficient information is specified for these agents to automate validation tests. Our use of agents to carry out the testing, including using agents to drive 3rd party testing tools,

contrasts to many other testing and performance validation approaches that require substantially tester-written code [2, 9, 14, 23]. Our approach allows developers to deploy components then be given proactive feedback on components' aspect-encoded constraint compliance. Our testing of JSP-based components is not as comprehensive as some [20], but we do allow for parsing of returned data and compare keywords to aspect-encoded URL calls.

Validation tests can be automatically re-run as more components are deployed or component configurations changed to provide developers with on-going feedback. Our approach of agents using aspect-encoded information to formulate operation invocations on deployed components to validate their configurations is similar to behaviour-based component repository work [12, 27]. However, we use this information to validate deployed components rather than locate components by behaviour for deployment. We have found developing most of our validation agents to be low-effort – key steps are to encode and extract required test case generation information, generate and run test cases, and compare results to encoded constraints. Use of third-party testing tools requires suitable tool integration though to date we have used data files predominantly to integrate tools.

To assess the effectiveness of our validation agents we have used hand-written test scripts and third-party testing tools to compare with our validation agent's testing results. We performed over a dozen hand-crafted tests on the Furniture System components and compared these results to those of four of our validation agents. This showed that as long as the validation agents had access to sufficient aspect-codified test methods, test data and required constraints, they can provide as accurate validation of deployed components as hand-built tests. However, analysing why a component doesn't meet its aspect-codified constraints can be quite difficult for developers if the agents have generated the tests. This implies that improved test oracle support is necessary in future agents. An additional problem occurs when aspect information was deficient or inaccurate - component deployments flagged as invalid are not necessarily correct.

The main disadvantages of our approach are the necessity to specify sufficiently detailed and consistent aspect information for component implementations and effort in building validation agents. While we have developed prototype tool support to generate aspect encodings from design models [8], this is still primitive and insufficient for many of our prototype validation agents needs. Some validation tests are extremely difficult to perform even with component aspect information e.g. does a component deployment situation satisfy reliability, non-functional user interface and data integrity constraint specifications? We have currently only used our agents to validate snap-shots of a system's deployed components and not done continuous validation of a system of components with a dynamic evolving architecture. We currently have not investigated the issue of validating interacting aspects i.e. validation agents only test specific aspect details and not multiple, interacting component aspects. We have so far only applied our technique to J2EE-based software components although have developed a number of quite different validation agents.

We plan to record testing results and use these to feedback into the component development process to help guide enhancements to component development. We plan to allow for some aspect constraints to be specified in more general ways in aspect encodings, and specific constraint values to be provided by separate AspectTestData components. This will allow aspect constraints to be tuned to a component's deployment situation. Improved tool support will allow developers to have aspect information generated, encoded and made accessible for deployed components automatically. We plan to make use of other 3rd party testing tools with our agents, and to look at validating other kinds of middleware using these tools (e.g. message-oriented middleware like SOAP).

## 8. Summary

We have developed a technique of characterising software components via the cross-cutting system concerns that impact component methods, called aspects. Component aspect information encoded with J2EE components is accessible after component deployment by validation agents which perform tests on deployed component configurations. These agents determine if aspect-specified functional and non-functional properties of the components are met in their current deployment scenario. Developers are proactively informed of invalid component deployment and configuration and can reconfigure systems to ensure required individual component and system constraints are met.

## Acknowledgements

Support from the New Zealand Foundation for Research, Science and Technology and the University of Auckland Research Committee for this research is gratefully acknowledged, as are the helpful comments on an earlier version of this paper from the anonymous referees.

## References

1. Allen, P. and Frost, S. *Component-Based Development for Enterprise Systems: Apply the Select Perspective™,* SIGS Books/Cambridge University Press, 1998.
2. Baudry, B., Vu Le, H., Le Traon, Y. Testing-for-trust: the genetic selection model applied to component qualification. In *Proceedings of the 33rd International Conference on*

*Technology of Object-Oriented Languages and Systems*, IEEE CS Press, 2000, pp.108-119.

3. Beugnard, A., Jezequel, J.-M., Plouzeau, N., Watkins, D., Making components contract aware, IEEE Computer, vol.32, no.7, July 1999, pp.38-45.
4. D'Souza, D.F. and Wills, A., *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.
5. Feather, M.S., Smith, B. Test oracle automation for V&V of an autonomous Spacecraft's planner, Model-Based Validation of Intelligence - Papers from the 2001 AAAI Symposium, AAAI Press, 2001.
6. Grundy, J.C. Multi-perspective specification, design and implementation of software components using aspects, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 10, No. 6, December 2000.
7. Grundy, J.C., Cai, Y., Liu, A. Generation of Distributed System Test-beds from High-level Software Architecture Descriptions, In *Proceedings of the 2001 IEEE International Conference on Automated Software Engineering,* San Diego, CA., Nov 26-28 2001, IEEE CS Press.
8. Grundy, J.C. and Patel, R. Developing Software Components with the UML, Enterprise Java Beans and Aspects, In Proceedings of the 2001 Australian Software Engineering Conference, Canberra, Australia 26-29 August 2001, IEEE CS Press.
9. Gorton, I. And Liu, A. Evaluating Enterprise Java Bean Technology, In *Proceedings of Software - Methods and Tools*, Wollongong, Australia, Nov 6-9 2000, IEEE CS Press.
10. Haddox, J.M., Kapfhammer, G.M. An approach for understanding and testing third party software components, In Proceedings of 2002 Annual Reliability and Maintainability Symposium, Seattle, WA, 28-31 Jan. 2002, IEEE CS Press.
11. Han, J. Zheng, Y. Security characterisation and integrity assurance for component-based software. In *Proceedings of the 2000 International Conference on Software Methods and Tools*, IEEE CS Press.
12. Henninger, S. Supporting the Construction and Evolution of Component Repositories, In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996, IEEE CS Press, pp. 279-288.
13. Ho, W.M., Pennaneach, F., Jezequel, J.M., and Plouzeau, N. Aspect-Oriented Design with the UML, In *Proceedings of the ICSE2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, Limerick, Ireland, June 6 2000.
14. Hoffman, D., Strooper, P. Tools and techniques for Java API testing. In *Proceedings of the 2000 Australian Software Engineering Conference,*IEEE CS Press, pp.235-245.
15. Hoffman, D., Strooper, P., White, L. Boundary values and automated component testing. *Software Testing Verification & Reliability*, vol. 9, no. 1 (March 1999), Wiley, pp.3-26.
16. Hu L., Gorton, I. A performance prototyping approach to designing concurrent software architectures, In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, IEEE, pp. 270 – 276.
17. Jurie, M.R., Rozman, I., Nash, S. Java 2 distributed object middleware performance analysis and optimization, *SIGPLAN Notices* 35(8), Aug. 2000, ACM, pp.31-40.
18. Kaiya, H. and Kaijiri, K., Specifying Runtime Environments and Functionalities of Downloadable Components under the Sandbox Model, In *Proceedings of the International Symposium on Principles of Software Evolution*, Kanazawa, Japan, Nov 2000, IEEE CS Press, pp. 138-142.
19. Khan, K.M. Han, J., Composing security-aware software, IEEE Software, vol.19, no.1, Jan.-Feb. 2002, pp.34-41.
20. Lee, S.C., Offutt, J. Generating test cases for XML-based Web component interactions using mutation analysis, In Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong, China, 27-30 Nov 2001, IEEE CS Press.
21. Ma, Y-S. Oh, S-U. Bae, D-H., Kwon, K-R. Framework for third party testing of component software. In *Proceedings of the Eighth Asia-Pacific Software Engineering Conference*, IEEE CS Press, 2001, pp.431-434.
22. McCann, J.A., Manning, K.J. Tool to evaluate performance in distributed heterogeneous processing. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing*, IEEE, 1998, pp.180-185.
23. McGregor, J.D. Parallel Architecture for Component Testing. Journal of Object-Oriented Programming, vol. 10, no. 2 (May 1997), SIGS Publications, pp.10-14.
24. Mezini, M. and Lieberherr, K. Adaptive Plug-and-Play Components for Evolutionary Software Development, In *Proceedings of OOPSLA'98*, Vancouver, WA, October 1998, ACM Press, pp. 97-116.
25. Motta, E., Fensel, D., Gaspari, M., Benjamins, R. Specifications of Knowledge Components for Reuse, In *Proceedings of 11th International Conference on Software Engineering and Knowledge Engineering*, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 36-43.
26. Orso, A., Harrold, M.J., Rosenblum, D., Rothermel, G., Soffa, M.L., Do, H. Using component metacontent to support the regression testing of component-based software, In Proceedings of the IEEE International Conference on Software Maintenance, Florence, Italy, 7-9 Nov 2001, IEEE CS Press.
27. Pai, Y. and Bai, P. Retrieving software components by execution, In *Proceedings of the. 1st Component Users Conference,* Munich, July 1996, SIGS Books, pp. 39-48.
28. Perronel,P. Chaganti, K. *Building Java Enterprise Systems with J2EE*, Sams, June 2000.
29. Qiong, W., Jichuan, C., Hong, M., and Fuqing, Y. JBCDL: an object-oriented component description language, Proc. of the 24th Conf. on Technology of Object-Oriented Languages, (September 1997), IEEE CS Press, pp. 198 – 205.
30. Rakotonirainy, A. and Bond, A. A Simple Architecture Description Model, In *Proceedings of TOOLS Pacific'98*, Melbourne, Australia (Nov 24-26, 1998), IEEE CS Press.
31. Szyperski, C.A., *Component Software: Beyond OO Programming*, Addison-Wesley, 1997.
32. Tarr, P., Ossher, H., Harrison, W. and Sutton, S.M. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE 21),* May 1999.