Preprint of paper published in Journal of Systems and Software © Elsevier 2004.

Deployed Software Component Testing using Dynamic Validation Agents

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

Software component run-time characteristics are largely dependent on their actual deployment situation. Validating software components i.e., confirming that they meet functional and non-functional property requirements, is time-consuming and for some properties quite challenging. We describe the use of "validation agents" to automate the testing of deployed software components to verify that they have the non-functional properties required. Our validation agents utilise "component aspects" that describe functional and non-functional cross-cutting concerns impacting on software components. Aspect information is queried by our validation agents and these construct and run automated tests on the deployed software components. The agents then determine if the deployed components meet their aspect-described requirements. Some agents deploy existing performance test-bed generation tools to run realistic loading tests on components. We describe the motivation for our work, how component aspects are designed and encoded, our automated agent-based testing process, the architecture and implementation of our validation agents, and our experience in using them.

1. Introduction

There are three key times at which software components can be validated to ensure that they havew the required functional and non-functional characteristics: static, design-time analysis; when testing components during implementation; and when validating components in actual deployment scenarios. Design-time specification checking requires formally codifying component characteristics and reasoning about proposed component compositions. Many approaches have been used to describe software components formally (Grundy, 2000; Motta et al, 1999; Oiong et al, 1997). During component implementation developers can run tests on partially- or fully-implemented components to verify that a component's behaviour meets its specification, using specially developed tools (Baudry et al, 2000; Hoffman et al, 1999; McGregor, 1997).

However, software components are run on un-verified hardware, networks, operating systems and in conjunction with unverified third-party COTS components (Gorton and Liu, 2000; Ma et al, 2001). It is thus currently not possible to ensure through static component specification checking or by the use of "laboratory situation" testing that components will comply with both functional and non-functional constraints when actually deployed. We describe the use of "validation agents" that use characterisations of software components to test if these components comply with functional and non-functional constraints in an actual deployment situation. We use an approach we have developed called "component aspects" as the codification mechanism to capture cross-cutting concerns impacting on software components and thus to describe the functional and non-functional constraints to be checked. We describe our validation agent architecture, examples of simple validation agents, and a complex performance validation agent that makes use of an existing performance test-bed generation tool. We conclude by presenting an evaluation of our work, a comparison with related component

validation approaches, and summarise the contributions of our research to automated component engineering.

2. Motivation

Component-based software development is an approach that composes self-describing, reusable and tailorable units of functionality (components), sometimes dynamically (Szyperski, 1997; Grundy, 2000). Components reused for a particular application may have been designed and built by developers without knowledge of the eventual component deployment scenario (Szyperski, 1997). Components are deployed and must operate with other components, databases, application servers, networks and hardware from a variety of sources, mostly unverified, and with functional and non-functional constraints that are un-codified or unobtainable. The problem then is to assure developers and users of component-based systems that the components they use will meet their specifications (Hu and Gorton, 1997).

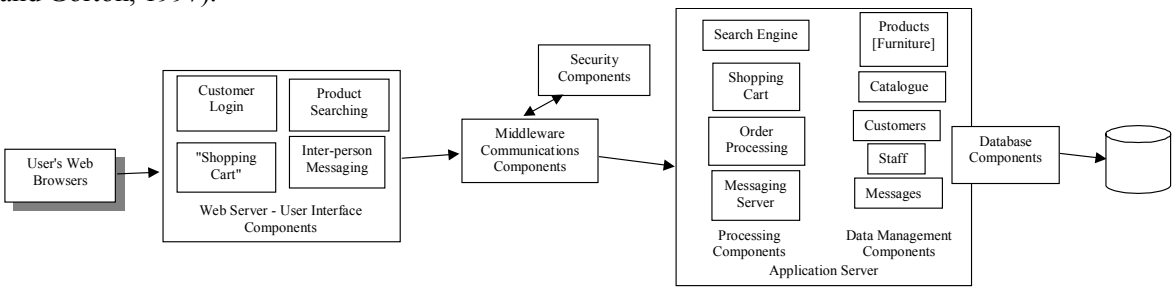


Figure 1. An example component-based application.

Consider an example of component-based application, an on-line furniture store. This application consists of a number of software components, as shown in Figure 1. Components include those realising its user interfaces (customer login, product search, shopping cart, messaging) its business processing logic (product catalogue, carts and orders, messages); and data management (customers, products, orders and order lines, messages). Both individual components and the components they are deployed with have functional and non-functional constraints that "cross-cut" the components that make up the system (Grundy, 2000). Examples of constraints include: performance (e.g. execution or response time; number of concurrent users; data transfer rates), which is notoriously difficult to predict (Gorton and Liu, 2000; Jurie et al, 2000); resource usage (e.g. memory and disk space consumption, network bandwidth, web server CPU time), where components can adversely affect each other's resource utilisation in unpredictable ways (Jurie et al, 2000); transaction support (e.g. recovery, concurrency control, distributed transaction support); security (e.g. authentication of users, access control to server-side functions, data encryption and decryption); and data persistency and distribution (e.g. data storage, location and retrieval, data transmission, event subscription and notification). Verifying that component compositions comply with overall system constraints, and that individual component constraints are satisfied, is very difficult (Gorton and Liu, 2000; Grundy et al, 2001).

In order to support adequately the validation of deployed software component validation in an automated way we need to: (1) encode information about component constraints; (2) allow this information to be accessed at run-time; and (3) use this information to run extensive, realistic tests on deployed components. Where possible, we want to leverage existing testing frameworks and tools. Figure 2 illustrates how we make use of these "component aspects" to validate deployed software components. Component designs produced from our aspect-oriented component engineering (AOCE) method (Grundy, 2000) include characterisations of functional and non-functional systemic properties of components e.g. persistency, distribution, security, transaction processing, provided and required services and associated constraints. When implemented from these designs, encodings of component aspects in XML are constructed and associated with component implementations. Validation agents

can query parts of these XML-encoded component descriptions, formulating and running tests to check the components' actual deployment-time adherence to their aspect-codified constraints.

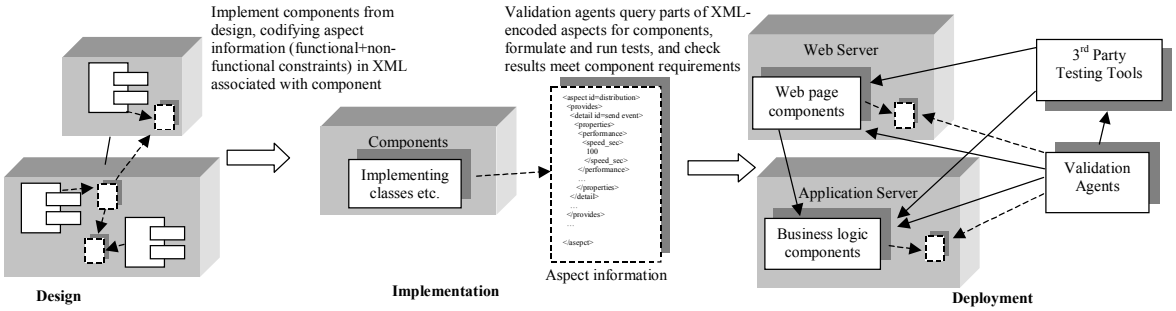


Figure 2. An overview of our aspect-oriented validation agent approach.

3. Aspect-oriented Component Description

We developed aspect-oriented component engineering (AOCE) to help developers design, implement and reuse their software components more effectively (Grundy, 2000). AOCE introduces the concept of *component aspects* which are used to group common cross-cutting component functional and non-functional behaviour (e.g. user interfaces, distribution, persistency, security, transaction processing, configuration support). Components typically *provide* and *require* a number of *aspect details*. For example, a "shopping cart" component might provide a user interface window and menu bar, a data structure, and event generation, but might require data persistency, event transmission and security management. 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. Components designed and built using AOCE are composed to form a network of inter-related components, some providing aspect-characterised services, some requiring these services. After designing software components using aspects a developer implements them using an appropriate technology. We have used our own JViews component architecture as well as Java 2 Enterprise Edition (J2EE) components to realise our aspect-oriented component designs. As well as implementing their software component designs, developers encode aspect-based information about these component implementations using XML, providing a description of the component's functional and non-functional properties that can be queried at run-time.

Parts of our Document Type Definition (DTD) describing the structure of component aspect XML documents are illustrated in Figure 3 (a). Component specifications include properties, operations, events and aspects. Component aspects, aspect details and aspect detail properties describe cross-cutting systemic concerns impacting the component's methods and state. Aspect details are provided or required, and properties include types and expressions constraining the property's value(s). We also support the specification of "validation methods" and URLs.

Figure 3 (b) shows examples of parts of the XML-encoded aspect information of two components, an OrderManager data management component and a ShoppingCart web server page component. Each component characterisation in XML encodes the interfaces of the components (properties, operations and events) as well as the aspect details provided and required by the component implementation. For the OrderManager component aspects include provided persistency management facilities and required transaction processing support. For the persistency management of this OrderManager component, additional constraints on its performance have been encoded indicating the minimum data storage and retrieval performance the component must support when deployed. Its actual persistency management performance will depend on the performance of the component(s) with which it is deployed that implement its data storage and retrieval. A testing agent will synthesise method calls

from the `ImpactedMethod` and `DetailTestMethod` aspect detail properties, and argument values from test data-providing EJBs e.g. `order_manager.ejbCreate();` `order_manager.ejbStore();` `order_data.setDate("12/02/2003");` etc. A conformance agent will test that the persistency of the EJB meets the target speed, as specified by the value of `StoreSpeed` property.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Component+>
<!ELEMENT Component (CompName, ComposedComp*,
  MappingName, CompProperties, CompMethods, CompEvents,
  CompAspects)>
<!ELEMENT CompProperties (CompProperty)+>
<!ELEMENT CompMethods (CompMethod)+>
...
<!ELEMENT CompAspects (CompAspect)+>
<!ELEMENT CompAspect (AspectName, AspectDetails)>
<!ELEMENT AspectDetails (AspectDetail)+>
<!ELEMENT AspectDetail (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="OrderManager">
  <MappingName>java:comp/env/ejb/orders</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>
   </Aspect AspectName="Transactions">
    <Detail DetailName="Strategy" DetailType="TransStrategy" Provided='false'>
   …
</Component>

<Component CompName="ShoppingCart.jsp">
  <MappingName>jsp/furniture/ShoppingCart.jsp</MappingName>
  <Aspects>
   <Aspect AspectName="UserInterface">
    … <Detail DetailName="Interface" DetailType="UIScreen" provided='true' …>
      … <DetailProperty DetailPropName="MinUsers" …>
        … <DetailPropConstraint><FixedValue>20</FixedValue>
     … <DetailProperty DetailPropName="ResponseTime" …>
        <ImpactedMethods><Method>GET</Method></ImpactedMethods>
      …. <DetailPropConstraint><Expr><LessThan>2500</LessThan>…
        <DetailTestMethod><URLCall URLName="ShoppingCart.jsp"
          URLArgumentData= java:comp/env/ejb/ShoppingCart_testdata1' />
  <Aspect AspectName="DataManagement" …>
   … <Detail DetailName="GetData" DetailType="DataQuery" Provided='false' …>
```

(b) Examples of component aspect descriptions.

Figure 3. (a) DTD for encoding component aspects; and (b) examples of component aspects.

The ShoppingCart web page provides user interface services (GET and POST HTTP requests for a web browser), requires data management (including OrderData), and data distribution support (provided by the web server in which it is eventually deployed). Additional constraints on these services encoded by the aspect detail properties include a minimum number of concurrent users the web page must support (different for different web servers that host it) and the performance of the web page when a GET is issued. Example data the web page should display when run is specified in an external testing component. This approach supports tailoring of example test data for different applications the ShoppingCart might be deployed in. An example URL invocation a simple performance testing agent may synthesise from the `ResponseTime` aspect detail and associated `ImpactedMethods` properties would be `http://localhost:8080/furniture/jsp/ShoppingCart.jsp?id=1&action=display`. A testing agent checking the number of concurrent users supported would attempt 20 or more connections, as indicated by the `MinUsers` detail property.

4. Validation Agent Architecture

After deploying software components developers use a set of "validation agents" to determine whether the components' aspect-encoded requirements are met in their current deployment and configuration scenarios. Figure 4 shows the architecture of our J2EE components augmented with aspect descriptions used by our validation agents. Each JSP hosted by a web server has an associated XML document that describes the component's aspect information, accessible via a URL. Aspect information for EJBs is accessed via either a URL or by "AspectInfo" EJBs providing meta-data

management. We have also used AspectTestData EJB components to provide updatable test data for use by our validation agents in different component deployment situations.

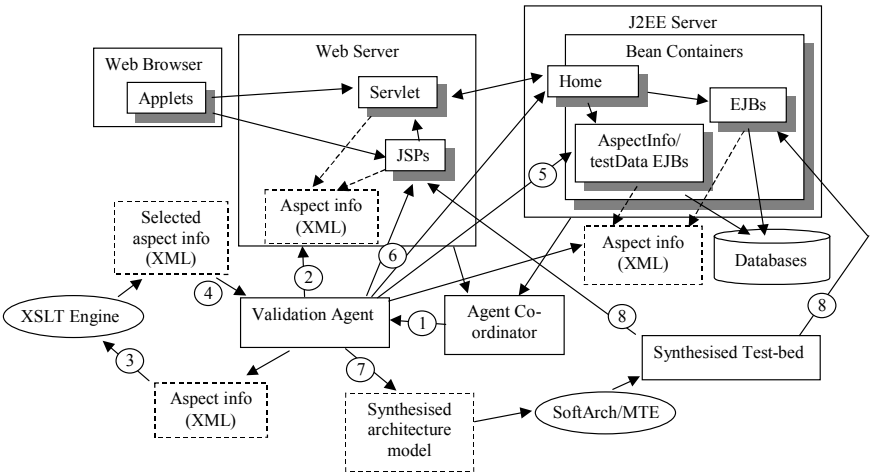


Figure 4. Our J2EE component validation agent architecture.

Validation agents are instructed to perform component validation tests by a co-ordinator or by the user (1). Some tests can only be run if the system is in a configuration in which the component being tested has other components it requires suitably deployed. The validation agent obtains component aspect information as an XML document (2), and uses Java's XML support to process component aspect-encoded specifications in order to determine appropriate tests to run on the deployed J2EE components (3). We use the XSLT transformation scripting language and the Xalan XSLT engine to query specific aspect details and properties encoded for a component, producing a set of aspect details and detail properties relevant to each agent (4). The validation agent then constructs tests for the component using the aspect information and test data for the component, obtained from a URL (hard-coded) or Enterprise JavaBean (accessing a test database) (5). Some agents are in external processes and run tests themselves, while others pass on information to others e.g. the pseudo-web browser, transaction testing agents or SoftArch/MTE performance testing tool (6). Validating agents run in a separate process and communicate with the deployed components via HTTP (if a JSP) or RMI (if an EJB). Others, such as transaction testing and resource measuring agents are deployed by the co-ordinator and run in the same process as the components under test, as they must take part in transactions or gain access to process information for these components. We used XSLT to synthesise a situation-specific architecture description from a template architecture and aspect information for SoftArch/MTE (7). These synthesised performance test-beds call deployed components (8) and return performance results to the validation agent for analysis.

5. Simple Validation Agents

We have prototyped a number of "simple" validation agents that check a variety of different software component characteristics after the components have been deployed. Table 1 summarises these validation agents, the aspect information they use and their purpose and basic operation. Some of these agents e.g. those checking single-client response time, functional operation and data storage/retrieval, perform their validation checks and analysis by themselves while others, e.g. transaction recovery checking, resource utilisation and security checking (authentication and access rights), need to deploy another agent to perform the testing and to notify the agent of the result. Validation agents compose method and URL calls using aspect information and example test data supplied by TestData components. They try valid and invalid transactions (e.g. commit, rollback raise exceptions, valid and invalid user, and access to remote object functions). Resource usage testing e.g.

memory, disk, CPU etc, requires validation agents to monitor system state attributes as tests are run. A co-ordinator ensures that validation tests are scheduled sequentially and while a system is not used.

Validation Agent	Aspect Detail(s) Used	Description
Data, Operation and Event Conformance Checking	Provided data/events; TestData providing component	Checks that component supports get/set of data. Calls operations to check they run and terminate. Subscribes to events and calls operations to generate events.
Web user interface conformance checking	Provided user interface; TestData providing component.	Runs GET/POST on web page component. Passes parameters from test data. Checks web page contains data in TestData component for given parameters by parsing HTML returned.
Persistency Checking	Provided persistency management; TestData component.	Sets properties of component from TestData, saves, changes and then reloads component. Checks raw performance of save/restore.
Resource Utilisation	Data, persistency requires CPU, memory properties. TestData component.	Calls operations specified in TestData and monitors resource utilisation (available CPU; memory consumption).
Transaction Recovery	Transaction or persistency details. TestData component.	Starts transactions and runs operations. Commits and rolls back transactions and determines component state changed/unwound.
Concurrent users supported	User interface and distribution properties.	Connects to component concurrently and checks specified number of users can be supported.
Simple performance checking	UI response time, persistency speed, distribution and transaction speed. TestData.	Does raw performance speed checks by calling operations with test data multiple times and checking total time taken/average time taken acceptable.
Security checking	Security properties. TestData.	Checks authentication (username, password) and access control rights by attempting valid and invalid operations.

Table 1. Examples of simple validation agents and the aspect information they use.

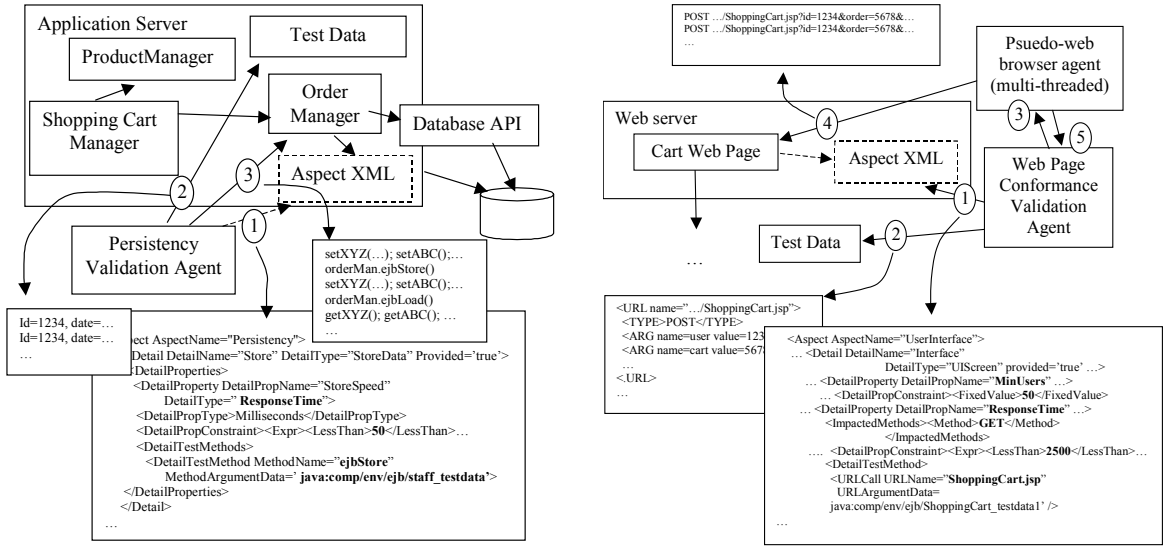


Figure 5. (a) Persistency validation agent operation; and (b) web page conformance validation agent.

Figure 5 (a) shows how a basic persistency validation agent tests if a component's persistency services function. In this example a Shopping Cart component makes use of ProductManager and OrderManager components. The persistency validation agent reads the OrderManager component's aspects and queries them to extract aspect detail information about persistency requirements from the components' aspect information (1); this includes methods that store data, component state to store, and methods to test restored component state. The aspects for this component also include the expected performance of the store/retrieve functionality. The persistency validation agent formulates requests to the component to create, update, store, retrieve the component state, and many requests may be generic for particular component implementations. Examples of data for method (and URL)

invocation are obtained from a TestData component (2), tailored to different deployment situations of the components. Invocations of methods or web page URLs (3) are then performed by the validation agent and the results analysed. The contents of the HTML returned by a URL invocation is parsed and example data items from the TestData components are searched for in this returned HTML.

In Figure 5 (b) a web page conformance validation agent obtains the aspects of a Shopping Cart web page implementation component (1). It then obtains test data from a TestData component (2) and determines the arguments the web page can use to perform different functions. The validation agent deploys a simple pseudo-web browser agent (3) to interact with the web page component (4), as if it were a customer's web browser. The delegate "pseudo-web browser" can check all functions of the web page component work with URLs generated from TestData example data. It can also check that the required number of concurrent users can be supported by the web page component, and can perform simple response time tests. Results are returned to the invoking validation agent (5).

6. Automated Performance Validation using SoftArch/MTE

Some kinds of deployed component validation require more sophisticated testing, e.g. to determine if a deployed component's response time (performance) will be adequate under "realistic" client, server and network loading conditions. In previous work 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 (Grundy et al, 1997). SoftArch/MTE takes a high-level architecture description specified by an architect, encodes this description in XML and then generates "test bed" client and server code, database configuration scripts and compilation and deployment scripts. These are uploaded to distributed hosts, run, the performance of selected client and server methods monitored, and results aggregated and sent back to SoftArch/MTE for visualisation. We have developed a validation agent to make use of this test-bed generator to test user loading, response and transaction processing times for realistically loaded EJB and JSP components.

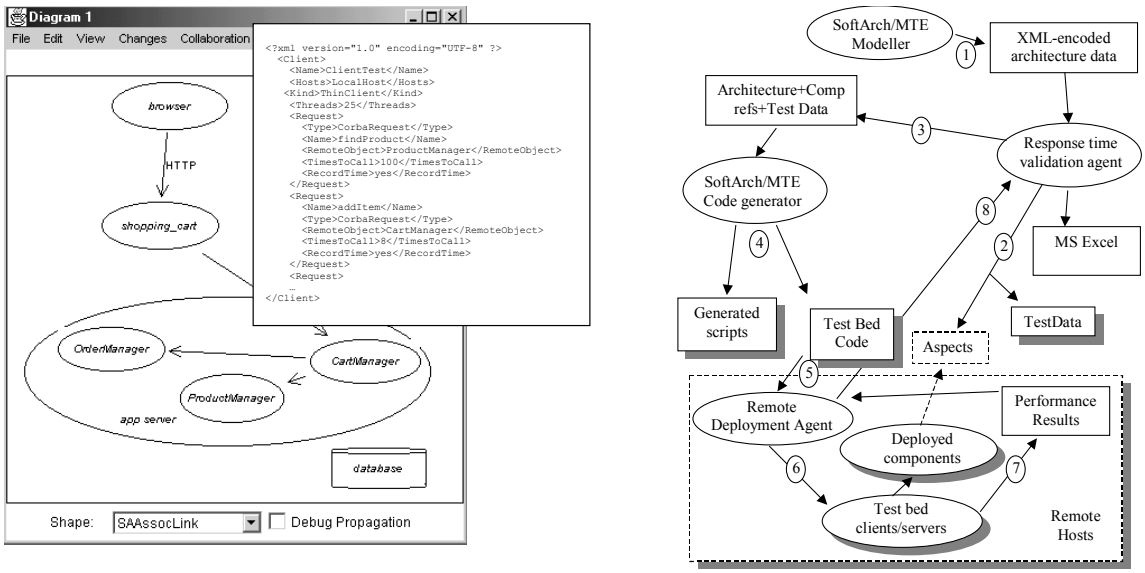


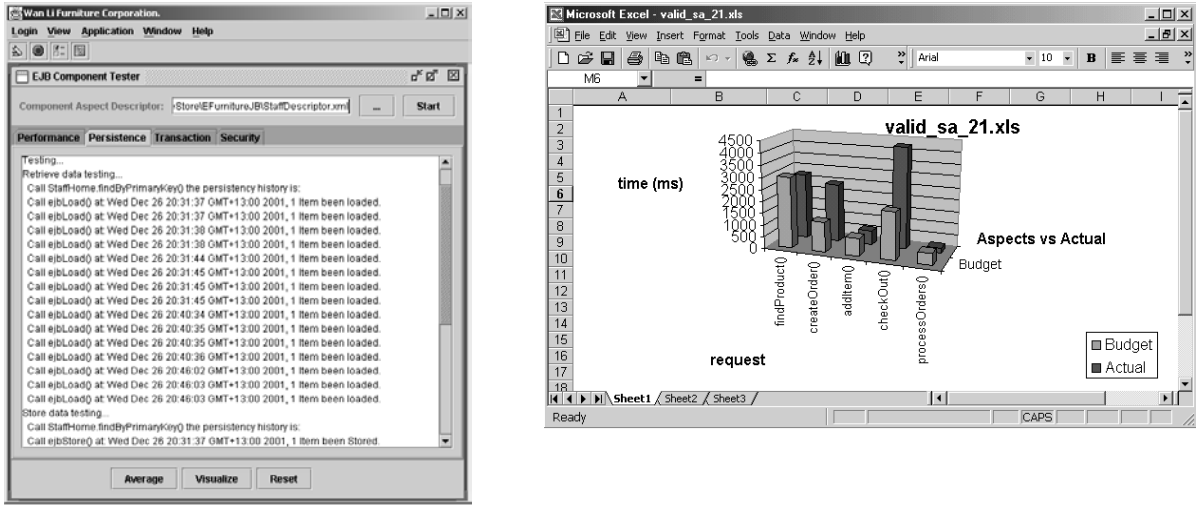
Figure 6. (a) A SoftArch/MTE architecture model; and (b) agent generating a performance test-bed.

Figure 6 (a) shows a SoftArch/MTE high-level architecture diagram for the furniture store system with part of the XML encoding this architecture description. Figure 6 (b) illustrates the validation agent operation: a software architect specifies the architecture component(s) will be deployed (1). A performance validation agent testing the response time of component methods to client requests detects deployed components, queries their aspect descriptions and obtains sample test data (2). It then

updates the SoftArch/MTE-generated architecture model with actual deployment information (e.g. JNDI name(s) of component(s), the number of tests to run, test data for clients/generated server objects) (3). This synthesised SoftArch/MTE architecture model is fed through the SoftArch/MTE's test-bed code generators (4), and the generated test-bed code and scripts are up-loaded to remote client and server hosts (5), test databases initialised and clients and server programs compiled, configured and initialised. When run, the clients generate requests to servers (6) and generated test-bed code instrumentation captures performance results (timing) which is recorded (7) and then sent back to the validation agent by SoftArch/MTE's remote deployment agents (8) for analysis.

7. Implementation

We have implemented our validation agents to check Java 2 Enterprise Edition applications and their Java Server Pages (JSPs) and Enterprise JavaBeans (EJBs) components. Key components in the J2EE software architecture include clients (thin clients via browsers and thick clients via applets and applications); middle-tier web servers (consisting of Java Server Page (JSP); JavaBean and Servlet "components"); enterprise servers (consisting of dynamically deployed Enterprise JavaBean (EJB) components); and databases.



(a) EJB storage validation agent results. (b) SoftArch performance results.

Figure 7. Examples of aspect-oriented component validation agent results visualised for analysis.

Validation agents provide user interfaces to display test results back to developers and to allow developers to run manually tests. Figure 7 (a) shows the output from an entity bean EJB persistency testing validation agent which has extracted persistency aspect information and ensures that the component's state can be stored and retrieved in its current deployment situation. Figure 7 (b) shows some output from a performance validation agent that has used SoftArch/MTE to execute performance tests under realistic loading for a deployed component, displaying the results adjacent to the required, aspect-codified performance benchmarks using an MS Excel™ chart.

8. Evaluation

To assess the effectiveness of our validation agents we have carried out two sets of evaluations: one comparing the results obtained from hand-testing deployed components with those from validation agent tests, and one using feedback from experienced J2EE component developers regarding the usefulness of some of our prototype agents. We evaluated EJB and JSP conformance validation

agents, EJB persistency testing agent, a memory usage monitoring agent and a SoftArch/MTE-based performance analysis agent.

For each agent-automated test of deployed J2EE components we built and ran a comparable test by hand, comparing results from each. We used HttpUnit and JUnit-style automated tests for conformance checking, hand-coded simple EJB persistency and memory usage tests, and specified SoftArch/MTE performance test bed configurations by hand. We also carried out performance tests of a fully-implemented furniture store system's EJBs and JSPs by hand, comparing these with the validation agent-synthesised performance tests and the hand-specified SoftArch/MTE performance tests. We had 4 experienced J2EE developers test the selected validation agents for feedback regarding their likely usefulness. These developers were asked to assemble part of the furniture store E-commerce application from a set of pre-developed components, checking deployed component behaviour as they went. Some available components failed to meet aspect-encoded performance constraints, while others needed parameter setting to meet conformance and persistency constraints.

Our evaluations showed that if validation agents have access to sufficient aspect-codified test methods, test data and required constraints, they can provide very accurate validation of deployed components. Agent-run conformance, memory and persistency tests are as accurate as equivalent hand-coded tests, while our performance tests via SoftArch/MTE were accurate to within approximately 15%. A major problem occurs when aspect data is incomplete so that an agent can not be used or is inaccurate. Developers considered that an HttpUnit-style automated testing approach would prove more effective for conformance checking JSP components but felt that performance testing JSP and EJB J2EE components via a validation agent using the SoftArch/MTE performance test bed generator provided a very efficient and accurate approach. Feedback indicated that the user interfaces for our validation agents could be improved to allow more flexible developer results analysis. A major current deficiency is the difficulty developers have in configuring test data.

9. Discussion

The majority of existing component testing approaches and tools have been used to apply tests to individual or small groups of components under what we describe as "laboratory" conditions (Ma et al 2001; McGregor, 1997; Hoffman and Strooper, 2000; Haddox and Kapfhammer, 2002). These allow developers to check that basic component functionality meets design specifications and some techniques enable developers to demonstrate that certain non-functional constraints are met under carefully controlled conditions. Some approaches use extracted component meta-data and wrappers to formulate automated component tests (Orso et al, 2001; Ma et al 2001).

Middleware performance testing work typically focuses on limited aspects of component functional and non-functional requirements (Gorton and Liu, 2000; McCann and Manning, 1998; Grundy et al, 2001). Some researchers and practitioners have carried out deployment-time evaluation of such system performance using realistic loading and usage conditions. However many of the current approaches to middleware evaluation suffer from a high degree of manual effort required of developers in building test harnesses with which to evaluate their components and middleware (Feather and Smith, 2001; Grundy et al 2001, Gorton and Liu 2000; Hoffman and Stooper, 2000). Most of the component validation approaches currently used also typically require extensive test bed prototyping to evaluate components (Gorton and Liu, 2000).

Most formal component description techniques that have been developed focus on specifying component functional properties (Beugnarrd et al, 1999; Motta et al 1999; Oiong et al, 1997). Few support non-functional constraint specifications, although some researchers have been developing the notion of component "trust" and also some specialised non-functional requirement description techniques, such as security, have been developed (Khan and Han, 2002; Kaiha and Kaijiri, 2000; Baudry et al, 2000). These support design-time composition reasoning but can not adequately validate component all deployments due to impossibility to specify 3rd party component characteristics.

Our aspect-oriented component characterisation technique has been sufficient to automate a range of quite different component validation tests for deployed software components. This approach supports a much greater range of automated deployed component testing to be performed than existing component description techniques. Our use of autonomous "software agents" to proactively carry out testing contrasts with many other approaches that require substantially tester-written code or manual testing tool configuration. The main disadvantages of our approach we have encountered to date are the necessity to specify sufficiently detailed aspect information for component implementations, multiple-aspect interactions and the difficulty in identifying why a deployed component does not comply with constraints. We have developed prototype tool support to generate aspect encodings from design models in previous work but this is still relatively primitive and insufficient for some of our prototype validation agents. Some validation tests are extremely difficult to perform even with component aspect information. For example, is a component deployment situation in accordance with reliability, non-functional user interface and data integrity constraint specifications? Some validation tests cut across aspects e.g. response time/performance. Providing developers with better analysis support of why a component doesn't comply with its constraints is needed. We have currently used our agents only to validate snap-shots of a system's deployed components and not in the continuous validation of a system of components for an in-use system or a dynamic, evolving architecture.

We plan to record testing results and use these to feedback into the component development process. In particular we want to provide developers with better analysis support for determining the causes of detected deployment problems. We plan to allow for some aspect constraints to be specified in more general ways in aspect encodings, and for specific constraint values to be provided by separate TestData components. We would like to make greater use of other 3rd party testing tools with our agents, and study validating other kinds of middleware (e.g. message-oriented middleware like SOAP in the .NET framework). We currently have not investigated validation of interacting aspects aspects. We have used agents only to test specific aspect details and not multiple component aspects.

10. Summary

We have developed a technique of characterising software components via the cross-cutting system concerns that impact component methods, that we call "component aspects". As we have encoded these aspect-based descriptions of software components using XML and made them available at run-time, we have been able to develop a number of "validation agents". These query the descriptions of deployed components and formulate tests to ensure that constraints applied to a component are complied with in its deployment scenario. Developers are informed of invalid component deployment and configuration. We implemented various prototype validation agents and have shown that these agents provide useful automated testing of deployed software components.

11. References

1. Baudry, B., Vu Le, H., Le Traon, Y. Testing-for-trust: the genetic selection model applied to component qualification. In *Proc. of the 33rd Int. Conf. on Tech. of Object-Oriented Languages and Systems*, IEEE CS Press, 2000, pp.108-119.
2. Beugnard, A., Jezequel, J.-M., Plouzeau, N., Watkins, D., Making components contract aware, IEEE Computer, vol.32, no.7, July 1999, pp.38-45.
3. Feather, M.S., Smith, B. Test oracle automation for V&V of an autonomous Spacecraft's planner, In *Proc. of the 2001 AAAI Symposium*, AAAI Press, 2001.
4. Grundy, J.C. Multi-perspective specification, design and implementation of software components using aspects, *Int. J. of Soft. Eng. and Knowledge Eng.*, Vol. 10, No. 6, December 2000.
5. Grundy, J.C., Cai, Y., Liu, A. Generation of Distributed System Test-beds from High-level Software Architecture Descriptions, In *Proc. of 2001 IEEE Int. Conf. on Automated Soft. Eng.,* San Diego, Nov 26-28 2001, IEEE CS Press.
6. Gorton, I. And Liu, A. Evaluating Enterprise Java Bean Technology, In *Proc. of Software - Methods and Tools*, Wollongong, Australia, Nov 6-9 2000, IEEE CS Press.
7. Haddox, J.M., Kapfhammer, G.M. An approach for understanding and testing third party software components, In *Proc. of 2002 Annual Reliability and Maintainability Symposium*, Seattle, WA, Jan 28-31 2002, IEEE CS Press.

8. Ho, W.M., Pennaneach, F., Jezequel, J.M., and Plouzeau, N. Aspect-Oriented Design with the UML, In *Proc. of the ICSE2000 Workshop on Multi-Dimensional Separation of Concerns in Soft. Eng.*, Limerick, Ireland, June 6 2000.
9. Hoffman, D., Strooper, P. Tools and techniques for Java API testing. In *Proc. of the 2000 Australian Soft. Eng. Conf.,*IEEE CS Press, pp.235-245.
10. 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.
11. Hu L., Gorton, I. A performance prototyping approach to designing concurrent software architectures, In *Proc. of the 2nd Int. Workshop on Soft. Eng. for Parallel and Distributed Systems*, IEEE CS Press, 1997, pp. 270 – 276.
12. 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.
13. Kaiya, H. and Kaijiri, K., Specifying Runtime Environments and Functionalities of Downloadable Components under the Sandbox Model, In *Proc. of the Int. Symposium on Principles of Software Evolution*, Kanazawa, Japan, Nov 2000, IEEE CS Press, pp. 138-142.
14. Khan, K.M. Han, J., Composing security-aware software, IEEE Software, vol.19, no.1, Jan.-Feb. 2002, pp.34-41.s
15. Ma, Y-S. Oh, S-U. Bae, D-H., Kwon, K-R. Framework for third party testing of component software. In *Proc. of the Eighth Asia-Pacific Soft. Eng. Conf.*, IEEE CS Press, 2001, pp.431-434.
16. McCann, J.A., Manning, K.J. Tool to evaluate performance in distributed heterogeneous processing. In *Proc. of the Sixth Euromicro Workshop on Parallel and Distributed Processing*, IEEE, 1998, pp.180-185.
17. McGregor, J.D. Parallel Architecture for Component Testing. J. of Object-Oriented Programming, vol. 10, no. 2 (May 1997), SIGS Publications, pp.10-14.
18. Motta, E., Fensel, D., Gaspari, M., Benjamins, R. Specifications of Knowledge Components for Reuse, In *Proc. of 11th Int. Conf. on Soft. Eng. and Knowledge Engineering*, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 36-43.
19. Orso, A., Harrold, M.J., Rosenblum, D., Rothermel, G., Soffa, M.L., Do, H. Using component meta-content to support the regression testing of component-based software, In Proc. of the IEEE Int. Conf. on Software Maintenance, Florence, Italy, 7-9 Nov 2001, IEEE CS Press.
20. Qiong, W., Jichuan, C., Hong, M., and Fuqing, Y. JBCDL: an object-oriented component description language, Proc. of the 24th Conf. on Tech. of Object-Oriented Languages, (September 1997), IEEE CS Press, pp. 198 – 205.
21. Szyperski, C.A., *Component Software: Beyond OO Programming*, Addison-Wesley, 1997.