An Observational Theory of Integration Testing for Component-Based Software Development

Hong Zhu
School of Computing and Mathematical Sciences
Oxford Brookes University
Gipsy Lane, Headington, Oxford OX3 0BP, UK
Email: hzhu@brookes.ac.uk

Xudong He
School of Computer Science
Florida International University
University Park, Miami, FL 33199, U.S.A.
Email: hex@cs.fiu.edu

Abstract

Integration testing plays a crucial role in component-based software development. In complementary to the existing works on the selection of test cases and measurement of test adequacy in integration testing, this paper focuses on questions about how to observe the behaviours of a large and complicated system during dynamic testing. We first analyse the structure of white-box integration testing and propose a family of integration testing methods. We then discuss and formalise the requirements of proper uses of test drivers and component stubs in incremental integration. Finally, we propose a set of axioms for integration testing of concurrent systems.

1. Introduction

In recent years, software component technology has emerged as a key element of modularity in the development of large and complicated systems [1~3]. Ensuring the correct integration of system components is a critical problem. Industrial practice of component-based development has shown a clear shift of development focus from design and code to requirements analysis, test and integration, especially from unit testing to integration testing [4~6].

Theoretically speaking, integration testing can be based on either the requirements specification, the design or the code of a system, or a combination of these. Most existing methods are based on the functional requirement specifications, e.g. [7]. Their major weakness is that the structure and design information is not utilised in the testing. Design-based methods consider the interactions between designed components of a system. Methods have been proposed to utilise the design information contained in UML [8], software architecture descriptions [9,10], and structured design diagrams [11,12]. Among code-based methods are inter-procedural data-flow testing methods [13~16], their extension to coupling-based methods [17], and more recently, interface mutation testing method [18]. A weakness of code-based methods is that they rely on the availability of the source code of the components. This is usually not the case when the component is a commercial off-the-shelf (COTS) package. They do not benefit from incremental integration strategies such as top-down or bottom-up integration. When the software system is large, analysing the complete set of code becomes impractical.

We regard software testing as a process in which a system's dynamic behaviours are observed and recorded so that the system's properties can be inferred. Integration testing distinguishes from testing at other development stages by observing the interaction between the components of the system, while unit testing focuses on the correctness of the components. In [19] we proposed a behaviour observation theory of software testing. This paper extends the above theory to integration testing.

The remainder of the paper is organised as follows. Section 2 briefly reviews our theory of behaviour observation. Section 3 discusses the uses of test drivers and component stubs in integration testing. Section 4 presents a set of axioms for integration testing of concurrent systems. Section 5 is the conclusion of the paper. Further work is discussed.

2. The observation theory

The theory is concerned with the behaviour observations in software testing [19~21]. In the testing of large-scale software systems, testers can only observe certain aspects of the system's dynamic behaviour. Such observations must be made systematically and consistently. In [19~21], we argued that the universe of observable behaviours of a software system p by a well-defined testing method constitutes an algebraic structure of complete partially ordered (CPO) set [22]. This universe has a least element $\perp_p$, which contains the minimum information about the system. It is partially ordered by a binary relation $\leq_p$. Informally, $\alpha \leq_p \beta$ means that the information contained in phenomenon β subsumes that in α. A consistent subset of phenomena $\alpha_1, \alpha_2, ..., \alpha_n$ has a least upper bound, denoted by $\alpha_1 + \alpha_2 + \cdots + \alpha_n$ or equivalently $\sum_{i=1}^{n} \alpha_i$. It is the summation of information contained in the phenomena.

The notion of observation schemes is an abstraction of systematic methods of behaviour observation.

Definition 1. (Observation schemes)

An *observation scheme* $\mathscr{B}$ is a mapping from software systems p to an order pair $<\mathscr{B}_p, \mu_p>$, where $\mathscr{B}_p$ is a CPO set; μ_p is the recording function that associates each test suite t with a collection of phenomena in $\mathscr{B}_p$. □

Notice that, first, each element in the collection of phenomena associated to a test suite represents one observation that can be made on a number of possible non-deterministic executions on the test suite. Second, a test suite is assumed to be a multi-set of test cases. The multiple appearances of a test case mean multiple executions of the system on the test case. Third, a test adequacy criterion as a stop rule is defined as a predicate C on the CPO set $\mathscr{B}_p$. $C(\sigma)$ being true means that a testing is adequate if the phenomenon σ is observed. An adequacy measurement criterion is defined as a mapping M from $\mathscr{B}_p$ to the unit interval of real numbers. $M(\sigma)$ is the testing adequacy of the observed phenomenon σ. A well-established software testing method can be defined as a triple $<\mathscr{B}_p, \mu_p, \mathscr{A}_p>$, where $\mathscr{A}_p$ is a test adequacy criterion.

A well-defined observation scheme must satisfy some desirable properties, i.e. axioms. Table 1 gives the formal definitions of the axioms proposed in [19]. Their inter-relationships are shown in Figure 1.

Table 1. Axioms of observation schemes

Axiom	Formal definition
Well-foundedness	$t \cap D_p = \varnothing \Rightarrow \mu_p(t) = \{\perp_p\}$
Observability	$t \cap D_p \neq \varnothing \Rightarrow \perp_p \notin \mu_p(t)$
Extendibility	$\sigma \in \mu_p(t) \wedge t \subseteq t' \Rightarrow \exists \sigma' \in \mu_p(t').(\sigma \leq_p \sigma'))$
Tractability	$\sigma \in \mu_p(t) \wedge t \supseteq t' \Rightarrow \exists \sigma' \in \mu_p(t').(\sigma' \leq_p \sigma))$
Repeatability	$\sigma \in \mu_p(t) \wedge t' \in \mathbf{bag}(\overleftarrow{t}) \wedge t' \supseteq t \Rightarrow \sigma \in \mu_p(t')$
Consistency	$\mu_p(t) \uparrow \mu_p(t')$
Completeness	$\underset{i \in I}{\forall} \sigma_i \in \mu_p(t_i).\ (\exists \sigma \in \mu_p.(\bigcup_{i \in I} t_i)\ (\sigma_i \leq_p \sigma))$
Composability	$\underset{i \in I}{\forall} \sigma_i \in \mu_p(t_i)\ .(\sum_{i \in I} \sigma_i \in \mu_p(\bigcup_{i \in I} t_i))$
Decompos-ability	$\sigma \in \mu_p(\bigcup_{i \in I} T_i) \Rightarrow \underset{i \in I}{\exists} \sigma_i \in \mu_p(T_i)\ .(\sigma = \sum_{i \in I} \sigma_i)$

These axioms have been validated against a large variety of testing methods. They hold for statement, branch, and path testing, mutation testing, data flow testing, and so on. It was also proved that statistical testing methods are neither composable nor decomposable, although they are consistent and complete [19].

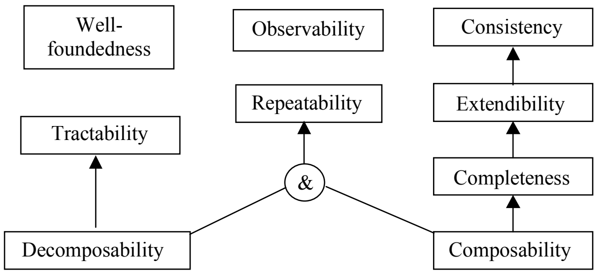

Figure 1. Relationships between the axioms

Different observation schemes have different fault detecting ability especially when one is an extraction of another.

Definition 2. (Extraction Relation between Schemes)

Scheme $\mathscr{A}$ is an *extraction* of scheme $\mathscr{B}$, written $\mathscr{A} \lhd \mathscr{B}$, if for all $p \in P$, there is a homomorphism φ_p from $<B_p, \leq_{B,p}>$ to $<A_p, \leq_{A,p}>$, such that (1) $\varphi_p(\sigma) = \perp_{A,p}$ if and only if $\sigma = \perp_{B,p}$, and (2) for all test sets t, $\mu_p^A(t) = \varphi_p(\mu_p^B(t))$. □

Informally, scheme $\mathscr{A}$ is an *extraction* of scheme $\mathscr{B}$ means what scheme $\mathscr{A}$ observes can be derived from the phenomena that $\mathscr{B}$ observes. For example, we can extract the set of executed statements from the set of executed paths. Let $\mathscr{A} = <A, \mu^A>$ and $\mathscr{B} = <B, \mu^B>$ be two schemes.

It is proved in [19] that extraction is a partial ordering. It also preserves the axioms of schemes discussed in the previous section. Formally, the extraction relation *preserves* property P, if $\mathscr{A} \lhd \mathscr{B}$ and $\mathscr{B}$ has property P implies that $\mathscr{A}$ also has property P.

The above theory is applied to the study of existing testing methods and the development of new methods for Petri nets [20]. We found that although there are a great number of testing methods proposed in the literature, the ways that these methods observe software behaviour have certain common structures, which determines their main properties. A number of constructions were defined and their properties were analysed. Such a construction can be applied to existing observation schemes to generate new schemes. Therefore, they constitute a sort of calculus of observation schemes [21].

3. Observation in integration testing

In this section, we further develop the theory by studying the axioms of behaviour observation for integration testing.

3.1. White-box integration testing

At a high level of abstraction, a component-based software system can be regarded as a number of software components plugged into an architecture. Such an architecture can be considered as a program constructor. In practice, it appears in the form of program code called *glueware*, while components may be in a number of forms, such as a module, a class, or a library, etc.

We are concerned with white-box integration testing (WIT) methods, which means that the code of glueware is available and used in testing. Using a WIT method, the tester observes the internal dynamic behaviour of the system rather than just the input/output. Moreover, the tester should be able to identify which part of the observation is about the components and to separate such information from the rest.

Definition 3. (White-box integration testing methods)

A *white-box integration* testing (WIT) method contains an observation scheme $\mathscr{B}$: $p \rightarrow <B_p, \mu_p>$, and for each component C in the system p under test, there exists a mapping φ_C from observable phenomena of the system in B_p to a universe $B_{C,p}$ of observable phenomena of the component C in the context of p. The mapping φ_C is called the *filter* for component C. □

Notice that, the universe of observable phenomena of a component determined by a WIT method should also be a CPO set, which may not have the same structure as that of the whole system. This is simply because that in integration testing we usually focus on the interaction between the components and their environment instead of the details of the behaviour of the component. In addition, we require that the partial ordering $\leq_{C,p}$ on $B_{C,p}$ must satisfy the following axioms.

Filter's Well-foundedness: if no observations on the whole system, nothing is known for the component. Formally, $\varphi_C(\perp_p) = \perp_{C,p}$, where $\perp_p$ and $\perp_{C,p}$ are the least elements of $<B_{C,p}, \leq_{C,p}>$ and $<B_p, \leq_p>$, respectively.

Filter Monotonicity: the more one observed the behaviour of the whole system, the more one knew about the component based on the observation. Formally,

$$\forall \sigma_1, \sigma_2 \in B_p.\left(\sigma_1 \leq_p \sigma_2 \Rightarrow \varphi(\sigma_1) \leq_{C,p} \varphi(\sigma_2)\right)$$

Filter Continuity: the information about a component contained in the sum of a number of global observations is equal to the sum of the information about the component contained in each individual global observation. Formally,

$$\forall \Theta \subseteq B_{p.}\left(\varphi(\sum_{\sigma \in \Theta} \sigma) = \sum_{\sigma \in \Theta} \varphi(\sigma)\right)$$

The following defines some observation schemes for integration testing methods.

Method I. The first method records the set of statements executed during integration testing. The invocation of a component such as the invocation of a function or procedure defined in a component is considered as a statement. Initiating the execution of a component as a process or thread, sending or receiving a message to such a process or thread are all considered as a statement. As in statement testing, details of the executed statement and their sequences of executions are not recorded. The method itself does not require observing and recording the execution of the statements inside a component. Therefore, the components are treated as black boxes.

This scheme has a set construction [21]. Its base set consists of the statements in the glueware. The filter for a component filter out the statements related to the component.

Method II. The second method not only records the same information for non-component-related statements as method I, it also records the set of component-related events with details about its parameters. For example, for a call of a function/procedure defined in a component, it will observe and record the name of the function/procedure invoked, the values of the parameters and the return values if any. For a message passing event, it will observe and record the receiver (or sender) component of the message and the contents in the message. The method itself does not require the events happened inside a component to be observed. It also treats components as black-box.

This scheme also has a set construction, but the base set is slightly more complicated. The element in the base set can be in one of two forms, a statement label indicating a non-component-related statement, and a tuple in the form of <event, parameter>, which indicates a component related event and the details of the parameter. The filters in this scheme are similar to those of method I.

Methods III. The third method observes even more information than method II. It records the execution sequences of the statements and component-related events. Notice that, a component is still regarded as a black box.

This scheme has a *poset* construction [21]. The base set is the set of paths in the glueware. Each path is a sequence of elements in the base set of method II. The filters filter out the component related events from each path.

Method VI. The fourth method records the same sequence of statements and component-related events as in method III. However, for each component-related event, it is annotated with the location in the component where the event is processed. This information enables testers to identify whether two events of the same type but with different parameters are handled differently inside a component. With additional information about how many different locations in the component where such an event is handled, software testers can measure the adequacy of integration testing.

In this method, components are not treated completely as black-boxes. However, with appropriate instrumentation of the components, the testing method can be applied without the source code of the components. The scheme is also a poset construction. The filters are similar to those in method III.

The above methods treat glueware as white-box and components as black-boxes. We call them *ground order* WIT methods. For each ground order method, we can generalise it to treat the component as white-box and observe the same aspect of behaviour inside the component. We call such a generalised method a *1st order* WIT method. For example, the generalisation of method I observes the statements at architectural level executed during testing as well as the statements inside the components. Notice that, a component may also be a composition of other components. We call a component in a component a *2nd order component*. Similarly, we define *3rd order components*, and so on. We will also use *high order components* to denote all the components of any order. A 1st order WIT method will not observe the same detail of the behaviour of components of 2nd order or higher. It can be further generalised to K'th order for any given natural number $K>1$ by observing the same detail of the K'th order components, but treating components of $K+1$'th order as black-box. The most powerful method is to treat all high order components equally as white-box. Such a method is called an *infinite order* WIT method. These observation schemes have the following extraction relationship. Its proof is omitted for the sake of space.

Proposition.

(1) For each method $Z\in\{I, II, III, VI\}$, we have that for all natural numbers $k\geq 1$,

$Z \lhd Z^{(1)} \lhd \ldots \lhd Z^{(k)} \lhd Z^{(k+1)} \lhd \ldots \lhd Z^{\infty}$;

(2) For all $n = 0, 1, 2, \ldots, \infty$, $I^{(n)} \lhd II^{(n)} \lhd III^{(n)} \lhd VI^{(n)}$,

where $Z^{(k)}$ is the k'th order generalisation of Z. □

3.2. Incremental integration

In practice, integration testing is often carried out piece-by-piece as each component is integrated. Integration strategies such as top-down, bottom-up and combinations of them are employed. Applications of such strategies involve writing and using test drivers and component stubs. This section investigates the requirements on test drivers and component stubs in the light of behaviour observation theory.

For the sake of simplicity, subsequently, we assume program constructors are binary, i.e. they take two components as parameters. The results can be easily generalised to constructors of any number of components. Let $\otimes$ be a binary program constructor, $p = p_1\otimes p_2$, where p_1 and p_2 are components. A component itself may well be a composition of some other components and formed by applying a program constructor, say $p_1=p_{1,1}\oplus p_{1,2}$.

By a bottom-up strategy, we first put $p_{1,1}$ and $p_{1,2}$ together to form $p_1=p_{1,1}\oplus p_{1,2}$ and test p_1 with a test driver to replace the constructor $\otimes$. After successfully testing p_1 and p_2 in this way, they are then put together to form $p = p_1\otimes p_2$ and tested. A test driver is in fact a program constructor $\otimes'$ which when applied to p_1 forms an executable program p'. During this testing process, we would like the test driver to act like the environment of p_1 as it would be in the real program p. This means that if we can observe the behaviour of component p_1 in the context of $\otimes$, we should be able to observe the same behaviour in the context of the test driver $\otimes'$. Suppose that we use a WIT method with observation scheme $\mathscr{B}$: $p \rightarrow (B_p, \mu_p)$. Hence, there is a filter φ from p to p_1 and a filter φ' from p' to p_1.

Representativeness of test drivers: For all test suites t and all phenomena σ of the component that can be observed by executing the system p on t, there exists a test suite t' for p' such that the same phenomenon σ can be observed by executing p' on test suite t'. Formally, $\forall t \in T_p$,

$$\forall \sigma \in \mu_p(t).\exists t' \in T_{p'}.\exists \sigma' \in \mu_{p'}(t').(\varphi(\sigma) \leq_{p_1} \varphi'(\sigma')) \qquad (1)$$

where $\leq_{p_1}$ is the partial ordering on B_{p_1}.

Equation (1) can be equivalently expressed as $\varphi(B_{p_1\otimes p_2}) \sqsubseteq_{p_1} \varphi(B_{\otimes'(p_1)})$, where $\varphi(X)=\{\varphi(x)|x\in X\}$, $X\sqsubseteq_{p_1} Y$ if and only if $\forall x \in X.\exists y \in Y.(x \leq_{p_1} y)$.

Test drivers may simply pass input data to the component under test and then execute the component. Such test drivers serve as an interface between the tester and the component. For an observation scheme that only observes the functional aspect of behaviour, such a test driver satisfies the representativeness axiom if it can pass all valid inputs to the component and pass the internal state of the component as the result of one execution to the next.

Sometimes, test drivers are written to combine other testing tasks, such as automatic generation of test cases. Such a test driver usually does not have representativeness, because it only generates input data in a sub-domain.

Representativeness on a sub-domain S: For all test suite t in a sub-domain S of the valid input of a system p, and all observable phenomena of the component by executing p on t, there is a test suite t' for the test driver that the same phenomena can be observed in the context of test driver. Formally, for all $t \in T_S \subseteq T_p$,

$$\forall \sigma \in \mu_p(t).\exists t' \in T_{p'}.\exists \sigma' \in \mu_{p'}(t').(\varphi(\sigma) \leq_{p_1} \varphi'(\sigma')) \qquad (2)$$

A top-down strategy starts with testing the program constructor $\otimes$ by replacing components p_n with stubs p'_n, n=1, 2. The difference between a real component p_n and a stub p'_n, is that we would not be able to observe the internal behaviour of p_n by executing p'_n. In fact, the internal behaviour of p_n is not the focus of observation in integration testing. However, we would like that the interaction between the component p_n and its environment in p is faithfully represented by the stub p'_n.

Faithfulness of component stubs: For all test suite t and all phenomena σ observable by executing the system p on t, the same observation can be obtained by executing the system p' obtained by replacing the component with the stub. Formally, $\forall t\in T_p.(\mu_p(t)=\mu_{p'}(t))$.

An implication of the faithfulness axiom is that a stub can replace a component, if the observation scheme treats the component as a black-box and if the observation scheme only concerns with the functional aspect of a system. In that case, the stub is required to produce functionally correct outputs. Therefore, if a k'th order WIT method is used, a component of k+1'th or higher order can be replaced by a stub. However, in practice, stubs tend to only provide partial functions of the component and they faithfully represent the components' behaviour only on a sub-domain of the component. This sub-domain is called the designated sub-domain of the stub.

Faithfulness of stubs on designated sub-domain: For all phenomena σ observable by executing the system p on a test suite t, σ can also be observed by executing the system p' obtained by replacing the component with a stub if the component is only executed on the stub's designated sub-domain. Formally, $\forall t \in T_p \lrcorner(C,S)\,.(\,\mu_p(t) = \mu_{p'}(t)\,)$, where $T_p \lrcorner(C,S)$ is the subset of T_p on which p only calls the component C on the designated sub-domain of stub S.

4. Integration of concurrent systems

A key question about integration testing methods is the relationships between the observations on the components (B_{p1}, μ_{p1}), (B_{p2}, μ_{p2}) and the observations on the composite ($B_{p1\otimes p2}$, $\mu_{p1\otimes p2}$). A desirable relationship represents the requirements on integration testing methods. Hence, we call such a relationship a *fitness condition*, or a *fitness axiom* of integration testing. The axioms discussed in the previous subsections are fitness conditions. They are independent of the program constructor. This section further investigates fitness axioms by studying the constructions of concurrent systems and proposes axioms for each constructor.

Non-deterministic choice. Given two components p_1 and p_2, the non-deterministic choice $p_1|p_2$ of p_1 and p_2 is a system that either behaves like p_1 or p_2 each time the composite system is called. For this construction, one of the desirable properties is that if a phenomenon σ is observable in testing p_1, it should also be observable in testing $p_1|p_2$. Let φ_1 and φ_1 be the filters for p_1 and p_2, respectively. We require that for n=1, 2,

$$\forall t.\left(\sigma \in \mu_{p_n}(t) \Rightarrow \exists \sigma' \in \mu_{p_1|p_2}(t).\left(\varphi_n(\sigma') = \sigma\right)\right); \qquad (1)$$

Another desirable property for non-deterministic choice is that any observation on the whole system consists of two disjoint parts: one for each component. Formally, for all $\sigma \in B_{p_1|p_2}$ there are $\sigma_1, \sigma_2 \in B_{p_1|p_2}$ such that

$$\sigma = \sigma_1 + \sigma_2, \sigma_1 \cap \sigma_2 = \varnothing, \varphi_2(\sigma_1) = \perp_2, \varphi_1(\sigma_2) = \perp_1 \qquad (2)$$

Moreover, the execution of p_1 | p_2 on each test case selects either p_1 or p_2 to execute, but not both. Formally, $\forall t \in T_{p_1|p_2}.\forall \sigma \in \mu_{p_1|p_2}(t).\exists t_1, t_2 \in T_{p_1|p_2}$,

$$t = t_1 \cup t_2, t_1 \cap t_2 = \varnothing, \varphi_1(\sigma) \in \mu_{p_1}(t_1), \varphi_2(\sigma) \in \mu_{p_2}(t_2) \qquad (3)$$

Parallel composition. In a parallel composition of two components, the components are executed in parallel. Therefore, a desirable property is that if phenomena σ_1 and σ_2 are observable in testing p_1 and p_2 independently, a combination of σ_1 and σ_2 should be observable when testing their parallel composition. Let φ_1 and φ_1 be the filters for p_1 and p_2, respectively. We require that for all t, $\sigma_1 \in \mu_{p_1}(t), \sigma_2 \in \mu_{p_2}(t)$ imply that

$$\exists \sigma \in \mu_{p_1 \| p_2}(t).\left(\varphi_1(\sigma) = \sigma_1 \wedge \varphi_2(\sigma) = \sigma_2\right). \qquad (4)$$

On the other hand, every observable phenomenon of the parallel composition of p_1 and p_2 is a combination of the behaviour of the components. Formally, $\forall t \in T_{p_1 \| p_2}$,

$$\forall \sigma \in \mu_{p_1 \| p_2}(t).\left(\varphi_1(\sigma) \in \mu_{p_1}(t) \wedge \varphi_2(\sigma) \in \mu_{p_2}(t)\right). \qquad (5)$$

Synchronised parallel composition. In a synchronised parallel composition, components may engage in synchronisation events. The composition of them may restrict the behaviour of each component to obey synchronisation rules. A phenomenon observable by testing the component alone may not be observable by executing the component in a given environment. Therefore, property (4) does not hold for synchronised parallel composition, while (5) is still true. We define a predicate *Syn* on the set $B_{p_1} \times B_{p_2}$. For all $\sigma_1 \in \mu_{p_1}(t)$ and $\sigma_2 \in \mu_{p_2}(t)$, $Syn(\sigma_1, \sigma_2)$ means that the behaviours σ_1 and σ_2 of p_1 and p_2 respectively are synchronised. $\forall t, \forall \sigma_1 \in \mu_{p_1}(t),\ \sigma_2 \in \mu_{p_2}(t),\ Syn(\sigma_1, \sigma_2)$ implies that $\exists \sigma \in \mu_{p_1 \| p_2}(t).\left(\varphi_1(\sigma) = \sigma_1 \wedge \varphi_2(\sigma) = \sigma_2\right)$.

Guarded command. A guard command $G \rightarrow C$ is a program that the command C is executed if the predicate G is true. We would like an observation scheme to be able to tell if the guard is satisfied or not. If the guard is not satisfied, the command will not be executed and hence no behaviour of the command part should be observed. Otherwise, a dynamic behaviour of command part should be observable if the guard is satisfied. Let φ_G and φ_C be the filters for the guard and command, respectively. For all test suites t, $\varphi_G\left(\mu_{G\rightarrow C}(t)\right) = \mu_G(t)$, $\varphi_C\left(\mu_{G\rightarrow C}(t)\right) = \mu_C\left(t \downarrow G\right)$, and $\varphi_C\left(\mu_{G\rightarrow C}\left(t - t \downarrow G\right)\right) = \mu_C(\varnothing)$, where $t \downarrow G$ is the subset of the test suite t that satisfies the guard G.

Sequential composition. For sequential composition p_1;p_2, the scheme should be able to observe both behaviours of p_1 and p_2. The part of p_1 should be the same as testing p_1 independently. The behaviours observable in testing p_2 in the context of executing p_1 first should be the same as using the output of p_1 as the input to p_2. Let φ_1 and φ_1 be the filters for p_1 and p_2, respectively. We require that $\forall t$, $\mu_{p_1}(t) = \varphi_1\left(\mu_{p_1;p_2}(t)\right)$, and $\varphi_2\left(\mu_{p_1;p_2}(t)\right) = \mu_{p_2}\left(p_1(t)\right)$, where $p_1(t)$ is the set of output of p_1 on the test set t.

Loops. For a loop structure, such as **'while** p **do** q', the loop control condition will be executed on all test cases in t. When the condition is true, the loop body will be executed and the condition will be evaluated again on the output of the loop body. Such executions will continue until the condition becomes false. Similarly, the loop body will be executed not only on those test cases that the control condition is true, but also on the output of the first loop that the conditions are also true, and so on. Let φ_C and φ_B be the filters for the loop condition p and the loop body q, respectively. We require that for all test suites t,

$$\varphi_C\left(\mu_{while\ p\ do\ q}(t)\right) = \mu_p(t) \cup \mu_p\left(q(t) \downarrow p\right) \cup$$

$$\mu_p\left(q(q(t)) \downarrow p \circ q\right) \cup \cdots \cup \mu_p\left(q^n(t) \downarrow p \circ q^{n-1}\right) \cup \cdots$$

i.e. $\varphi_C\left(\mu_{while\ p\ do\ q}(t)\right) = \mu_p(t) \cup \bigcup_{n=1}^{\infty} \mu_p\left(q^n(t) \downarrow p \circ q^{n-1}\right)$.

Similarly, $\varphi_B\left(\mu_{while\ p\ do\ q}(t)\right) = \bigcup_{n=0}^{\infty} \mu_q\left(q^n(t) \downarrow p \circ q^{n-1}\right)$,

where f^0 is the identity function.

5. Concluding remarks

In this paper, we applied the theory of behaviour observation to integration testing in component-based software development. We formalised the notion of WIT methods and analysed the requirements of test drivers and component stubs. We also proposed a set of axioms of observation schemes for integration testing of concurrent systems. There are a few directions for further work.

First, there is a number of testing methods proposed in the literature to support integration testing. We will examine whether these testing methods satisfy the axioms proposed in the paper.

Second, based on the understanding of the desirable properties of behaviour observation in integration, we will further investigate the algebraic structures of observable phenomenon and their corresponding recording functions that satisfy the properties. The constructions of observation schemes that we proposed and investigated in [21] will also be further studied with regard to the axioms for integration testing.

Acknowledgements

This work is jointly funded by the NSF of USA under grant INT-9731620 and the NSF of China under grant 69811120643. X. He was also partially supported by the Office of Naval Research of the USA under grant N00014-98-1-0591.

Reference

[1] Hopkins, J., Component primer, C. ACM, Vol. 43, No. 10, Oct. 2000, pp27~30.

[2] Szyperski, C., Component Software: Beyond Object-Oriented Programming, Addison Wesley, 1998.

[3] D'Souza, D. and Wills, A. C., Objects, Components and Frameworks with UML: The Catalysis Approach, Addison Wesley, Reading, MA, 1999.

[4] Sparling, M., Lessons learned through six years of component-based development, C.ACM, Vol. 43, No. 10, Oct. 2000, pp47~53.

[5] Crnkovic, I., Larsson, M., A case study: demands on component-based development, Proc. ICSE'2000, June 4~11, 2000, Limerick, Ireland, pp22~30.

[6] Morisio, M., Seaman, C. B., Parra, A. T., Basili, V. R., Kraft, S. E. and Condon, S. E., Investigating and improving a COTS-based software development, Proc. ICSE'2000, June 4~11, 2000, Limerick, Ireland, pp32~ 41.

[7] Chen, H. Y. Tse, T. H. and Chen, T. Y., TACCLE: a methodology for object-oriented software testing at the class and cluster levels, ACM Transactions on Software Engineering and Methodology**,** Vol. 10, No. 1**,** 2001.

[8] Abdurazik, A. and Offutt, J., Using UML collaboration diagrams for static checking and test generation, Proc. UML'00, York, UK, Oct. 2000, pp383~395.

[9] Richardson, D. and Wolf, A., Software Testing at the Architectural Level, Proc. of the 2nd International Software Architecture Workshop, San Francisco, California, October 1996, ACM Press, pp68~71.

[10] Bertolino, A., Corradini, F., Inverardi, P. and Muccini, H., Deriving test plans from architectural descriptions, Proc. ICSE'2000, June 4~11, 2000, Limerick, Ireland, pp220~229.

[11] Zhu, H., Jin, L., and Diaper, D., Application of Task Analysis to the Validation of Software Requirements, Proc. SEKE'99, Kaiserslautern, Germany, June, 1999, pp239~245.

[12] Zhu, H., Jin, L., Diaper, D. and Bai, G., Testing Software Requirements via Task Analysis, Technical Report CMS-TR-99-02, CMS, Oxford Brookes University, Jan. 1999. (*to appear in the Journal of Systems and Software*)

[13] Harrold, M,J., and Soffa, M.L., Selecting and Using Data for integration testing, IEEE Software, March 1991, pp58-65.

[14] Frankl, P.G. & Weyuker, J.E., An applicable family of data flow testing criteria, IEEE TSE, Vol.SE_14, No.10, October 1988, pp1483-1498.

[15] Ural, H. and Yang , B., Modeling software for accurate data flow representation, Proc. ICSE'93, May 1993, pp277~286.

[16] Pandi, H. D., Ryder, B. G., Landi, W., Interprocedural Def-Use associations in C programs, Proc. TAV4, Oct. 1991, pp139~153.

[17] Jin, Z. and Offutt, J., Integration testing based on software couplings, Proc. COMPASS'95, Gaithersburg, Maryland, June 1995, pp13~23.

[18] Delamaro, M. E., Maldonado, J. C., and Mathur, A. P., Interface Mutation: an approach to integration testing, IEEE TSE, Vol. 27, No. 3, March 2001, pp228~247.

[19] Zhu, H. and He, X., A theory of behaviour observation in software testing, Technical Report, CMS-TR-99-05, School of Computing and Mathematical Sciences, Oxford Brookes University, Sept. 1999.

[20] Zhu H. and He X., A Theory of Testing High-Level Petri Nets, Proc. of the IFIP 16th World Computer Congress, Beijing, China, August, 2000, pp443-450.

[21] Zhu H. and He X., Constructions of Behaviour Observation Schemes in Software Testing, Proc. HASE'00, New Mexico, Nov. 2000, pp2~12.

[22] Gunter, C. A., Scott, D. S., Semantic domains, In Handbook of Theoretical Computer Science, Vol. B., Formal Models and Semantics, Ed. J. van Leeuwen, The MIT Press/Elsevier, 1990, pp633~674.