A Methodology of Component Integration Testing

Hong Zhu[1] and Xudong He[2]

[1] Department of Computing
Oxford Brookes University
Wheatley campus
Oxford OX33 1HX, UK
hzhu@brookes.ac.uk

[2] School of Computer Science
Florida International University
University Park
Miami, FL 33199, USA
hex@cs.fiu.edu

Summary. Integration testing plays a crucial role in component-based software development. It is also very difficult due to the common problem of lack of information about the design of the components and the unavailability of source code of commercial off-the-shelf (COTS) components. Addressing this problem, we investigate how to observe system's dynamic behavior in component integration testing. Based on a theory of behavioral observation developed in our previous work, this chapter proposes a formal model of component integration testing methods and a hierarchy of behavioral observation schemes suitable for component integration testing. Their properties and interrelations are studied. Incremental integration testing strategies are also investigated. The requirements for proper uses of test drivers and component stubs in incremental integration are analyzed.

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, 2, 3]. Ensuring the correct integration of software components is a critical problem in component-based software development(CBSD). Industrial practices in CBSD have shown a clear shift of development focus from design and coding to requirements analysis, testing, and integration, especially from unit testing to integration testing [4, 5, 6]. However, traditional testing methods have to be adapted to meet the new requirements of CBSD.

Generally speaking, software integration testing can be based on the requirements specification, the design or the code of the system under test, or, ideally, a combination of these. Most existing methods are based on the functional requirement specifications, e.g., [7, 8]. Their major weakness is that program structure and design information are not utilized in the testing. Design-based methods have been proposed to utilize the information contained in design documents, such as in UML models [9, 10], software architecture descriptions [11, 12], and structural design diagrams [13, 14]. However, in the context of CBSD, the design information of the components is usually not available in testing when components are commercial off-the-shelf (COTS) packages. Among code-based methods are inter-procedural data flow testing methods [15, 16, 17, 18], their extensions to coupling-based methods [19], and, more recently, interface mutation testing methods [20]. A weakness of code-based methods is that they rely on the availability of the source code of the components. This weakness becomes a serious problem when the component is a COTS package because source code is not usually available for component users. Moreover, these methods do not support incremental integration strategies. Consequently, even if the source code is available, analyzing the complete set of code becomes impractical when the software system is large. Therefore, their applicability to CBSD is also limited.

It is widely recognized that the lack of information flows between component developers and component users is one of the main causes of the difficulties of software testing in CBSD [21]. In particular, as discussed above, component users have limited access to information about component design and implementation details in the integration of components into their systems. This implies that testers as component users have very limited ability to observe the internal behavior of the components. On the other hand, component developers, in the design and implementation as well as in the testing of components, have very limited knowledge about their uses. Hence, they have limited knowledge about what should be provided to the component's users to observe the behaviors of the components. Therefore, how to observe the behaviors of components becomes a crucial problem of testing in CBSD.

This chapter addresses this problem based on a general theory of behavioral observation in software testing that we proposed in [22, 23, 24, 25]. We study the aspect of software testing in which a system's dynamic behaviors are observed and recorded so that the system's properties can be inferred. The focus of behavioral observation is one of the most important characteristics that distinguishes software testing activities at different development stages. For example, integration testing focuses its observation on the interactions between the components of the system, while unit testing focuses on the internal behaviors of the components. The former represents component users' view toward testing software components and the latter represents component developers' view. This chapter will take the component users' view toward testing in the integration of components into a system. However, the theory can be applied equally to the testing of components from the developers' view if the components are compositions of other components. It also sheds new light on what component developers should provide to component users in order to help integration testing of components. Based on our formal theory of behavioral observation, especially the 'design patterns' of software testing methods obtained in the study of existing testing methods [24], we can derive a number of testing methods suitable for component integration testing. Moreover, a collection of guidelines for the proper uses of incremental integration testing strategies can also be developed from the theory.

The remainder of the chapter is organized as follows. Section 2 briefly reviews our theory of behavioral observation and presents a set of design patterns of software testing methods. Section 3 applies the theory to component integration testing. We propose a formal model of white-box integration testing in which components can be treated as black boxes while the code that glues components together are treated as a white box. We then apply design patterns to derive a collection of testing methods that are suitable for white-box component integration testing. The effective uses of test drivers and component stubs in incremental integration testing will also be investigated. Section 4 concludes the paper with a summary of the work reported in this chapter and a brief discussion of the directions for further work.

2 Overview of the Observation Theory

In software testing practices, observations on a system's dynamic behavior can be made on a number of different aspects of the execution of the system. For example, in addition to the correctness of the output, one can also observe the following:

(1) The set of executed statements, as in statement testing method;
(2) The set of exercised branches, as in branch testing method;
(3) The set of executed paths, as in path testing methods;
(6) The set of dead mutants, as in mutation testing;
(4) The sequences of communications between processes, as in the testing of communication protocols;
(5) The sequences of synchronization events, as in testing concurrent systems [26].

To render observed and recorded information meaningful, we require that observations be systematic and consistent. For example, it is not acceptable if sometimes we record executed statements, sometimes don't. We use the term 'observation schemes' to denote a systematic and consistent way of observing and recording a system's dynamic behavior in software testing. Here, a distinction must be made between an observed phenomenon and the dynamic behavior itself. A phenomenon is the result of an observation on a certain aspect of the dynamic behavior during a specific test execution of a software system. Employing different observation methods may result in observing different phenomena of the same behavior. Therefore, each observation method determines a universe of phenomena of the system's dynamic behavior observable from testing.

In this section, we review the theory that we developed in [22, 23, 24, 25] about the mathematical structure of the universe of observable phenomena, the mathematical properties of observation schemes, and the common structures of existing observation schemes in various software testing methods.

2.1 The Structure of Observable Phenomena

As argued in [22], the observable phenomena in testing a particular software system p using a well-defined testing method constitute an algebraic structure called Complete Partially Ordered set (CPO set). Formally, CPO sets are defined as follows.

Definition 1. *(CPO sets)*
A CPO set $\langle D, \leq \rangle$ consists of a nonempty set D and a binary relation $\leq$ on D, such that $\leq$ is a partial ordering and satisfies the following conditions:

(1) D has a least element, written $\bot$, i.e., for all $x \in D$, $\bot \leq x$;
(2) For all directed subsets $S \subseteq D$, S has a least upper bound, written as $\bigsqcup S$, i.e., for all upper bound u of S, $\bigsqcup S \leq u$;

where a subset $S \subseteq D$ is directed if for all $s_1, s_2 \in S$ there is $s \in S$ such that $s_1 \leq s$ and $s_2 \leq s$; an element $u \in D$ is called an upper bound of S if $s \leq u$ for all $s \in S$. □

The partial ordering $\leq$ on observable phenomena represents the fact that different observed phenomena contain different information of the dynamic behavior of the system. The relation $\alpha \leq_p \beta$ means that phenomenon α contains less information than phenomenon β in testing system p.

Example 1. (The universe of observable phenomena in statement testing)
For example, in statement testing, the set of statements in the program executed during testing is observed in addition to the correctness of the output. A phenomenon of the dynamic behavior of a system is the set of statements executed during testing. All such sets constitute a universe of phenomena and the partial ordering relation on the universe is the set inclusion relation $\subseteq$. The more a program is tested, the larger the set of statements executed. □

Within a CPO set, for all directed subsets S of elements, there is the least upper bound $\bigsqcup S$. In the context of software testing, the least upper bound of a set of observed phenomena serves as an operation that summarizes the observations and draws a general conclusion about the testing. The result is a phenomenon that puts information from all the independent observations together. It contains all the information in the individual observations and nothing more. For example, in statement testing, the union of the sets of executed statements is the summation operation on observable phenomena.

The set union operation is the least upper bound of two sets with respect to the set inclusion relation. This is consistent with the intuition that if two sets of statements are executed in two independent tests, the union of the sets of statements contains exactly all the tested statements.

The least element $\bot_p$ in the universe of observable phenomena in testing program p represents the phenomenon that can be observed if the system is not executed at all. Therefore, it usually is the phenomenon that contains no information about the dynamic behavior of a system. For example, in statement testing, the least element is the empty set, which means no statement is executed. Of course, any set of statements observed in a testing includes the empty set.

CPO sets have been well studied in denotational semantics of programming languages, and constitute domain theory. Readers are referred to [27] for a concise treatment of domain theory and its uses in the studies of the semantics of programming languages.

Note that, in many software testing methods, there is also a greatest element of observable phenomena in testing a software system p, written as $\top_p$. That is, for all phenomena α, we have that $\alpha \leq_p \top_p$. For example, the set of all feasible statements in a program is the greatest element in the observable phenomena in statement testing. In many cases, the greatest element can be observed only by testing on an infinite number of test cases. For example, the set of all feasible paths in a program is the greatest observable phenomenon in path testing. Usually, it can be covered only by infinite tests if the program contains loops. While such a greatest element might be useful, it does not necessarily exist in all testing methods. For the sake of generality, our theory does not assume its existence. The results obtained subsequently can also be applied to all phenomenon spaces, including those having the greatest elements.

2.2 The Notion of Observation Schemes

The notion of observation schemes is formally defined as a mathematical structure to represent systematic behavioral observation methods of software testing.

Definition 2. *(Observation scheme)*
A scheme $\mathcal{B}$ of behavioral observation and recording, or simply an observation scheme, is a mapping from software systems p to ordered pairs $\langle \mathbf{B}_p, \mu_p \rangle$, where $\mathbf{B}_p = \langle B_p, \leq_p \rangle$ is a CPO set that represents the universe of observable phenomena of p. Function μ_p is called the recording function, which is a mapping from test sets T to nonempty subsets of B_p.$\square$

Informally, $\mu_p(T)$ is the set of all possible phenomena observable by testing p on test set T. In other words, $\sigma \in \mu_p(T)$ means that σ is a phenomenon that is observable by an execution of p on test set T. Note that, in testing a concurrent system, two executions of the same system p on the same test set

T may demonstrate two different behaviors due to nondeterminism. Consequently, one can observe more than one phenomenon in two executions of p on the same test set, say, σ and σ'. Both σ and σ' belong to the set of such observable phenomena for testing p on test set T, which is denoted by $\mu_p(T)$.

Note also that, to test concurrent systems, a test set T can be a multiple set (or bag) so that multiple executions of the system on the same test case can be described. In this chapter, the set of all multiple sets on a set X is denoted by $\mathbf{bag}(X)$. We write $\widehat{Y}$ to denote the set obtained by removing duplicated elements in a multiple set Y. The traditional set operators are used to denote their multiple set variants as well. Moreover, in the test of interactive or process control systems, a test case can be a sequence of input data without an upper limit on the lengths. In such cases, a partial ordering on the test cases of the system can be defined and the input space of the system forms a CPO set. Therefore, test sets can be elements of a power domain of the input CPO set. For the sake of generality, we assume that there is only a partial ordering on test sets and the collection of all test sets of a program forms a complete partially ordered set. The least element of the test sets means that the software is not dynamically tested. The greatest test set is the exhaust test, if it exists. For the sake of readability, we use the set inclusion symbol $\subseteq$ as the partial ordering relation between test sets, the empty set $\emptyset$ as the least element of the test sets, and the set union $\cup$ and intersection $\cap$ symbols as the least upper bound and the greatest lower bound of test sets, respectively. We also use the set membership symbol $\in$ to denote that a test case is contained in a test set. We will use P to denote the set of all software systems. The input domain of a software system p is denoted by D_p.

The following examples illustrate the notion of observation schemes.

Example 2. (Input/output observation scheme)
Let $IO_p = \{\langle x, y\rangle | x \in D_p \wedge y \in p(x)\}$, where $y \in p(x)$ means that y is a possible output of p when executed on input data x. The universe of observable phenomena is defined to be the power set of IO_p, and the partial ordering is set inclusion. The recording function $\mu_p(T)$ is defined to be the collection of sets of input/output pairs observable from testing p on T.

For instance, assume that $D_p = \{0, 1\}$, $p(1) = \{1\}$, and $p(0) = \{0, 1\}$ due to non-determinism. Let test data $t = 0$ and test set $T_1 = \{t\}$; then, $\mu_p(T_1) = \{\{\langle 0, 0\rangle\}, \{\langle 0, 1\rangle\}\}$, i.e., one may observe either $\{\langle 0, 0\rangle\}$ or $\{\langle 0, 1\rangle\}$ by executing p on input 0 once. Let test set $T_2 = \{2t\}$; then, $\mu_p(T_2) = \{\{\langle 0, 0\rangle\}, \{\langle 0, 1\rangle\}, \{\langle 0, 0\rangle, \langle 0, 1\rangle\}\}$, i.e., one of the following three different phenomena can be observed by executing p twice on the same input 0:

$\{\langle 0, 0\rangle\}$ / p outputs 0 in two executions on input 0;
$\{\langle 0, 1\rangle\}$ / p outputs 1 in two executions on input 0;
$\{\langle 0, 0\rangle, \langle 0, 1\rangle\}$ / p outputs 0 in one execution and outputs 1 in another execution on the same input 0. □

Example 3. (Dead mutant observation scheme)
Consider the observation scheme for mutation testing [28, 29, 30]. Let Φ be a set of mutation operators. The application of Φ to a program p produces a set of mutants of p. Let $\Phi(p)$ be the set of such mutants that are not equivalent to p. Define the universe of phenomena to be the power set of $\Phi(p)$. The partial ordering is defined to be the set inclusion relation. For all test sets T, the recording function $\mu_p(T)$ is defined to be the collection of sets of mutants. Each element in $\mu_p(T)$ is a set of mutants that can be killed by one test of p on T. □

Example 4. (Output diversity observation scheme)
The observation scheme in this example records the number of different outputs on each input data. A phenomenon observable from testing a concurrent system on a set of test cases consists of a set of records. Each record has two parts, $\langle t, n\rangle$, where t is a valid input, and n is the number of different outputs on the input data observed from the testing. Formally, an element of the universe of phenomena is a set in the form of $\{\langle t_i, n_i\rangle | t_i \in D_p, n_i > 0, i \in I\}$. The partial ordering relation on phenomena is defined as follows:

$$\sigma \leq \sigma' \Leftrightarrow \forall\langle t, n\rangle \in \sigma.\exists\langle t', n'\rangle \in \sigma'.(t = t' \wedge n \leq n').$$

The least upper bound of σ_1 and σ_2 is a set, written as $\sigma_1 + \sigma_2$, which contains elements in the form of $\langle t, n\rangle$ and satisfies the following conditions.

(a) $\langle t, max(n_1, n_2)\rangle \in \sigma_1 + \sigma_2$, if $\exists n_1, n_2 > 0.(\langle t, n_1\rangle \in \sigma_1 \wedge \langle t, n_2\rangle \in \sigma_2)$;
(b) $\langle t, n_1\rangle \in \sigma_1 + \sigma_2$, if $\exists n_1 > 0.(\langle t, n_1\rangle \in \sigma_1) \wedge \neg\exists n_2 > 0.(\langle t, n_2\rangle \in \sigma_2)$;
(c) $\langle t, n_2\rangle \in \sigma_1 + \sigma_2$, if $\exists n_2 > 0.(\langle t, n_2\rangle \in \sigma_2) \wedge \neg\exists n_1 > 0.(\langle t, n_1\rangle \in \sigma_1)$.

□

2.3 Test Adequacy Criteria

One of the most important elements of all software testing methods is the concept of test adequacy criteria. Since the introduction of the concept in 1970s [28], a large amount of research on test adequacy criteria has been reported in the literature; see, e.g., [38] for a survey.

Software test adequacy criteria can play at least two significant roles in software testing [38]. First, a test adequacy criterion provides an objective guideline to select test cases so that adequate testing can be performed. Many software test criteria have been proposed as such guidelines or test case generation and selection methods. A test set generated according to such a guideline or by using such a method is therefore adequate according to the criterion. In the research on the theories of software testing, test adequacy criteria are, therefore, usually formally defined as predicates on the space $\mathbf{T}$ of test sets and software systems P, i.e., as mappings $\mathcal{C} : \mathbf{T} \times P \rightarrow Bool$, (cf. [29, **?**, 31, 32, 33, 34, 35, 36]). Second, a test adequacy criterion also provides a stop rule to determine whether a testing is adequate and can stop. Such a stop rule is often used together with a metric to determine how much has been achieved by the testing so far. For example, the percentage of statements covered by testing is a metric for statement coverage criterion. Theoretically speaking, such a metric is a function that gives a mapping from test sets and software systems to a numerical scale such as the unit interval, i.e., $\mathcal{C} : \mathbf{T} \times P \to [0, 1]$ (see, e.g., [37]).

A common feature of existing theories of test adequacy criteria is that they consider test adequacy as a property of test sets. However, as discussed in the previous sections, in the testing of concurrent and non-deterministic systems, the behavior of the system under test is not uniquely determined by the input test cases. Test adequacy is a property of the dynamic behavior demonstrated in the testing process. Therefore, we redefine the notion of test adequacy criteria as predicates of observed phenomena or as measurements on observed phenomena. Formally, let $\mathcal{B} : p \to \langle \mathbf{B}_p, \mu_p \rangle$ be an observation scheme. An adequacy criterion $\mathcal{C}$ as a stop rule is a mapping from software system p to a predicate C_p defined on B_p, such that, for any phenomenon, $\sigma \in \mu_p$, $C_p(\sigma) = \mathit{true}$ means that the testing is adequate if phenomenon σ is observed. An adequacy criterion $\mathcal{C}$ as a measurement is a mapping from software system p to a function M_p from B_p to the unit interval $[0, 1]$ of real numbers. For any phenomenon, $\sigma \in \mu_p$, $M_p(\sigma) = \rho \in [0, 1]$ means that the adequacy measurement of the testing is ρ if the phenomenon observed is σ.

In this framework, a software test method can be defined as an ordered pair $\langle \mathcal{B}, \mathcal{C} \rangle$ of a behavioral observation scheme $\mathcal{B}$ and a test adequacy criterion $\mathcal{C}$. In this chapter, we will focus on the observation schema because it is a difficult issue of testing in CBSD.

2.4 Properties and Axioms of Behavioral Observations

Having recognized that observation schemes are an essential part of all testing methods, we now discuss what a good observation scheme is. In [22], we proposed a set of desirable properties of observation schemes and studied the interrelationships between the properties. We now list the axioms. Their rationales can be found in [22].

Axiom 1. *(Empty set property)*
Nothing can be observed from the empty testing. Formally,

$$\forall p \in P, (\mu_p(\emptyset) = \{\bot_p\}) \,. \tag{1}$$

□

Axiom 2. *(Observability)*
If a software system is tested on at least one valid input, some nontrivial phenomenon of the system's behavior can always be observed. Formally,

$$\forall p \in P, (T \cap D_p \neq \emptyset \Rightarrow \bot_p \notin \mu_p(T)) \,. \tag{2}$$

□

A testing process is often incremental in the sense that more and more test cases are executed and observations are made cumulatively. Suppose that a system p is tested on test set T, and a phenomenon σ_1 is observed. Later on, some additional test cases are executed and a new observation σ_2 is made as the result of testing on T', where $T \subseteq T'$. The following axioms state the required properties for an observation scheme to be used in incremental testing:

Axiom 3. *(Extensibility)*
Every phenomenon observable from testing a system on a test set is part of a phenomenon observable from testing on any of its supersets. Formally,

$$\forall p \in P, (\sigma \in \mu_p(T) \wedge T \subseteq T' \Rightarrow \exists \sigma' \in \mu_p(T'), (\sigma \leq_p \sigma')) . \tag{3}$$

□

Axiom 4. *(Tractability)*
Every phenomenon observable from testing a system on a test set T contains a phenomenon observable from testing on any subset T'. Formally,

$$\forall p \in P, (\sigma \in \mu_p(T) \wedge T \supseteq T' \Rightarrow \exists \sigma' \in \mu_p(T'), (\sigma \geq_p \sigma')) . \tag{4}$$

□

A special case of incremental testing is to repeatedly execute a system on the same test cases. For testing concurrent systems, such repetition often reveals new behavior. However, the same phenomenon should be observable when repeating a test. Hence, we have the following axiom:

Axiom 5. *(Repeatability)*
Every phenomenon observable from testing a system p on a test set T can be observed from repeating the test of p on the same test set T. Formally,

$$\forall p \in P, (\sigma \in \mu_p(T) \Rightarrow \sigma \in \mu_p(T \cup T)). \tag{5}$$

□

Note that, in the above axiom, when the test set T is a multiple set of test cases, $T \cup T$ represents that all test cases in T are executed twice; hence, we may have that $T \cup T \neq T$.

Axiom 6. *(Consistency)*
For any given system p, any two phenomena observed from two tests of the system must be consistent. Formally,

$$\forall p \in P, (\mu_p(T) \uparrow \mu_p(T')), \tag{6}$$

where $\sigma_1 \uparrow \sigma_2$ means that the phenomena σ_1 and σ_2 are consistent, i.e., they have a common upper bound; $\Gamma_1 \uparrow \Gamma_2$ means that the sets Γ_1 and Γ_2 are consistent, i.e., for all $\sigma_1 \in \Gamma_1$ and $\sigma_2 \in \Gamma_2$, $\sigma_1 \uparrow \sigma_2$. □

In software testing practices, a testing task is often divided into several subtasks and performed separately. Such a testing strategy can be considered as testing on several subsets of test cases, and observations are made independently by executing on the subsets. These observations are then put together as the result of the whole testing effort. The following axioms are concerned with such testing processes:

Axiom 7. *(Completeness)*
Every phenomenon observable from testing a system on a subset is contained in a phenomenon observable from testing on the superset. Formally,

$$\forall p \in P, \left(\underset{i\in I}{\forall} \sigma_i \in \mu_p(T_i), (\exists \sigma \in \mu_p(T), (\sigma \geq_p \sigma_i)) \right), \tag{7}$$

where $T = \bigcup_{i \in I} T_i$. □

Axiom 8. *(Composability)*
The phenomena observable by testing a system p on a number of test sets can be put together to form a phenomenon that is observable by executing p on the union of the test sets. Formally,

$$\forall p \in P, \left(\underset{i\in I}{\forall} \sigma_i \in \mu_p(T_i), \left(\bigsqcup_{i\in I} \sigma_i \in \mu_p(\bigcup_{i\in I} T_i) \right) \right). \tag{8}$$

□

Axiom 9. *(Decomposability)*
For all test sets T and its partitions into subsets, every phenomenon observable from testing a system p on the test set T can be decomposed into the summation of the phenomena observable from testing on the subsets of the partition. Formally, let $T = \bigcup_{i\in I} T_i$*; we have,*

$$\forall p \in P, \left(\sigma \in \mu_p(T) \Rightarrow \underset{i\in I}{\exists} \sigma_i \in \mu_p(T_i), (\sigma \bigsqcup_{i\in I} \sigma_i) \right). \tag{9}$$

□

Figure 1 below summarizes the relationships between the axioms, where arrows are logic implications. Proofs of these relationships can be found in [22].

2.5 Extraction Relation Between Schemes

From a phenomenon observed under a given scheme, one can often derive what is observable under another scheme. For example, we can derive the set of executed statements from the set of executed paths. The following extraction relation formally defines such relationships between observation schemes:

Let $\mathcal{A} : p \to \langle \mathbf{A}_p, \mu_p^A \rangle$ and $\mathcal{B} : p \to \langle \mathbf{B}_p, \mu_p^B \rangle$ be two schemes.

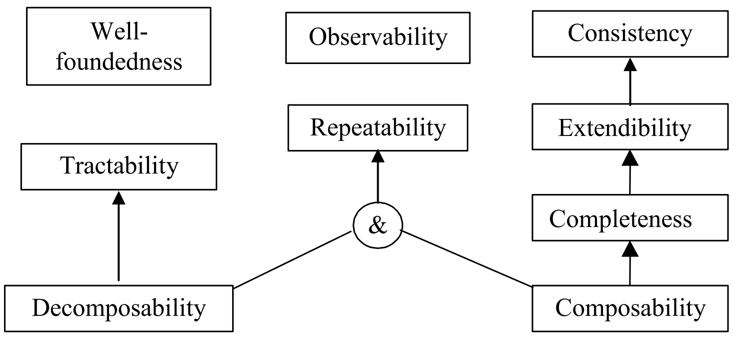

Fig. 1. Relationships between the axioms

Definition 3. *(Extraction relation between schemes)*
Scheme $\mathcal{A}$ is an extraction of scheme $\mathcal{B}$, written $\mathcal{A} \lhd \mathcal{B}$, if for all $p \in P$, there is a homomorphism φ_p from $\langle B_p, \leq_{B,p} \rangle$ to $\langle A_p, \leq_{A,p} \rangle$, such that

(1) $\varphi_p(\sigma) = \bot_p$ if and only if $\sigma = \bot_{B,p}$, and
(2) for all test sets T, $\mu_p^A(T) = \varphi_p(\mu_p^B(T))$.

□

The extraction relation is a partial ordering on observation schemes.

Informally, scheme $\mathcal{A}$ being an extraction of scheme $\mathcal{B}$ means that scheme $\mathcal{B}$ observes and records more detailed information about dynamic behaviors than scheme $\mathcal{A}$. The phenomena that scheme $\mathcal{A}$ observes can be extracted from the phenomena that scheme $\mathcal{B}$ observes. Consequently, if a fault in the system under test can be detected according to an observed phenomenon using $\mathcal{A}$, the same fault can also be detected by using observation scheme $\mathcal{B}$. In other words, $\mathcal{A}$ being an extraction of $\mathcal{B}$ implies that $\mathcal{B}$ has better fault detection ability than $\mathcal{A}$.

Note that, first, extraction relations between observation schemas are similar to the subsumption relations on test adequacy criteria. A test adequacy criterion $\mathcal{C}_1$ subsumes criterion $\mathcal{C}_2$ if for all tests T, T is adequate according to $\mathcal{C}_1$ implies that T is also adequate according to $\mathcal{C}_2$ (cf. [36, 38, 39]). However, the schema of testing method M_1 is an extraction of the schema of method M_2 does not imply that there is a subsumption relation between their adequacy criteria, or vice versa. Counterexamples can be found in [23]. In fact, a number of different test adequacy criteria can be defined on one observation schema. Second, subsumption relations between test adequacy criteria have been intensively investigated to compare testing methods, and are considered an indication of better fault detection ability. In posterior uses of test adequacy criteria, it does guarantee better fault detection ability [38]. However, it was proven that a subsumption relation cannot alone always guarantee better fault detection ability if the observation method used during testing is not taken into consideration [36]. In contrast, as discussed above, an extraction relation between two observation schemas can guarantee that one test method has a better fault detection ability than the other. Finally, the extraction relation between observation schemas allows us to compare the test methods' fault detection abilities without assuming that the tests are adequate.

2.6 Design Patterns of Observation Schemes

A great number of software testing methods have been proposed and investigated in the literature (see, e.g., [39] for a survey of unit testing methods). Our investigation of the observation schemes of existing testing methods has shown that there are a number of common structures occurring repeatedly [24]. These common structures can be considered as design patterns of observation schemes. They enable us to understand the strengths and weaknesses of testing methods from a very high level of abstraction, and to develop testing methods according to the desired features. They, therefore, provide guidelines for the design of testing methods. This section summarizes the common constructions of observation schemes and their properties. Readers are referred to [24] for details.

A. Set Construction

In statement testing, software testers observe and record the subset of statements in the software source code that are executed (see, e.g., [40, 41]). In this observation scheme, the execution of a statement is an atomic event to be observed. An observable phenomenon is a set that consists of such events that happened during testing. The partial ordering on observable phenomena is set inclusion. Such construction of a scheme is common to many testing methods. The following is a formal definition of this construction:

Definition 4. *(Regular set scheme)*
An observation scheme $\mathcal{B} : p \rightarrow \langle \mathbf{B}_p, \mu_p \rangle$ is said to be a regular set scheme (or simply a set scheme) with base $U_{p \in P}$ if, for all software systems $p \in P$, the elements in the CPO set $\langle B_p, \leq_p \rangle$ are subsets of U_p and the partial ordering $\leq_p$ is the set inclusion relation $\subseteq$. Moreover, the following conditions hold for the mapping μ_p:

(1) $U_p = \bigcup_{t \in D_p} (\bigcup \mu_p(\{t\}))$,
(2) $\mu_p(\emptyset) = \{\emptyset\}$,
(3) $T \cap D_p \neq \emptyset \Rightarrow \emptyset \notin \mu_p(T)$,
(4) $\mu_p(T) = \mu_p(T \cap D_p)$,
(5) $\mu_p(\bigcup_{i \in I} T_i) = \{\bigcup_{i \in I} \sigma_i | \sigma_i \in \mu_p(T_i), i \in I\}$.

□

The following theorem about the extraction relations between regular set observation schemes will be used later in the study of integration testing:

Theorem 1. *(Extraction theorem for regular set schemes)*
Let $\mathcal{B} : p \to \langle \mathbf{B}_p, \mu_p^B \rangle$ be a regular scheme. Let $\mathcal{A} : p \to \langle \mathbf{A}_p, \mu_p^A \rangle$. Assume that, for all software systems $p \in P$, there is a set U_p^A such that $\langle A_p, \leq_p \rangle$ is a CPO set of subsets of U_p^A with set inclusion relation $\subseteq$. If, for all $p \in P$, there is a surjection f_p from U_p^B to U_p^A such that $\sigma_A \in A_p \Leftrightarrow \exists \sigma_B \in B_p, (\sigma_A = \{f_p(x) | x \in \sigma_B\})$, and, for all test sets T, $\mu_p^A(T) = \{f_p(\sigma) | \sigma \in \mu_p^B(T)\}$, then we have that

(1) $\mathcal{A}$ is a regular scheme with base U_p^A, and
(2) $\mathcal{A}$ is an extraction of $\mathcal{B}$.

We say that $\mathcal{A}$ is the regular scheme extracted from $\mathcal{B}$ by the extraction mapping f_p. □

In particular, observation scheme $\mathcal{A}$ is an extraction of scheme $\mathcal{B}$ if, for all programs p, $U_p^A \subseteq U_p^B$.

B. Partially Ordered Set Construction

In the set construction, there is no ordering relationship between the basic events to be observed. However, in some testing methods such as path testing, the basic events are ordered by a partial ordering.

Let X be a nonempty set and $\ll$ be a partial ordering on X. A subset $S \subseteq X$ is said to be downward closed if, for all $x \in S$, $y \ll x \Rightarrow y \in S$. Let $p \in P$. Given a partially ordered set (also called poset) $\langle A_p, \ll_p \rangle$, we define the universe B_p of phenomena to be the set of downward closed subsets of A_p. The binary relation $\leq_{B,p}$ on phenomena is defined as follows:

$$\sigma_1 \leq_{B,p} \sigma_2 \Leftrightarrow \forall x \in \sigma_1, \exists y \in \sigma_2, (x \ll_p y) \tag{10}$$

It is easy to prove that $\leq_{B,p}$ is a partial ordering. Moreover, if the poset $\langle A_p, \ll_p \rangle$ has a least element $\bot_p$, the poset $\langle B_p, \leq_{B,p} \rangle$ is a CPO set with the least element $\{\bot_p\}$. The least upper bound of σ_1 and σ_2 is $\sigma_1 \cup \sigma_2$.

Definition 5. *(Partially ordered set scheme)*
An observation scheme $B : p \to \langle B_p, \leq_p^B \rangle$ is said to be a partially ordered set scheme (or poset scheme) with base $\langle A_p, \ll_p \rangle$ if its universe of phenomena is defined as above and the recording function has the following properties:

(1) $\mu_p(\emptyset) = \{\{\bot_p\}\}$,
(2) $T \cap D_p \neq \emptyset \Rightarrow \{\bot_p\} \notin \mu_p(T)$,
(3) $\mu_p(T) = \mu_p(T \cap D_p)$,
(4) $\mu_p(\bigcup_{i \in I} T_i) = \{\bigcup_{i \in I} \sigma_i | \sigma_i \in \mu_p(T_i), i \in I\}$. □

Example 5. (Observation scheme for path testing [29, 30, 42])
Let p be any given program. A path in p is a sequence of statements in p executed in the order. Let A_p be the set of paths in p, and the partial ordering $\ll_p$ be the sub-path relation. Let s be a set of paths in p. The downward closure of s is the set of sub-paths covered by s, written as $\bar{s}$. Let T be a test set. We define

$$\mu_p(T) = \{\bar{s}_{T,p} | s_{T,p}\},$$

where $s_{T,p}$ is a set of execution paths in p that may be executed on T.

It is easy to see that the function defined above satisfies conditions (1) through (4) in the definition of the poset scheme. $\square$

As in Example 5, we can define observation schemes that observe the sequences of a type of events that happened during test executions of a system, such as the sequences of communication and synchronization events. Such schemes have the same property as the scheme for path testing.

C. Product Construction

Given two observation schemes $\mathcal{A}$ and $\mathcal{B}$, we can define a new scheme from them by including the information observed by both schemes. The following defines the product scheme of $\mathcal{A}$ and $\mathcal{B}$:

Definition 6. *(Product construction)*
Let $\mathcal{A} : p \to \langle \mathbf{A}_p, \mu_p^A \rangle$ and $\mathcal{B} : p \to \langle \mathbf{B}_p, \mu_p^B \rangle$. The scheme $\mathcal{C} : p \to \langle \mathbf{C}_p, \mu_p^C \rangle$ is said to be the product of $\mathcal{A}$ and $\mathcal{B}$, written $\mathcal{C} = \mathcal{A} \times \mathcal{B}$, if for all software systems $p \in P$,

(1) $\mathbf{C}_p = \langle C_p, \leq_{C,p} \rangle$, where

$$C_p = \{\langle \sigma_A, \sigma_B \rangle | \sigma_A \in \mathbf{A}_p, \sigma_B \in \mathbf{B}_p\},$$

$$(\langle \sigma_A, \sigma_B \rangle \leq_{C,p} \langle \sigma'_A, \sigma'_B \rangle) \Leftrightarrow (\sigma_A \leq_{A,p} \sigma'_A) \wedge (\sigma_B \leq_{B,p} \sigma'_B);$$

(2) for all test sets T, $\mu_p^C(T) = \mu_p^A(T) \times \mu_p^B(T)$. $\square$

Example 6. (Typed dead mutant observation scheme)
In Example 3, an observation scheme is defined for mutation testing. In software testing tools, mutation operators are often divided into a number of classes to generate different types of mutants (see, e.g., [43]). Dead mutants of different types are then recorded separately to provide more detailed information. To define the observation scheme for this, let Φ_1, Φ_2,$\cdots$, Φ_n be sets of mutation operators. Each $\Phi_i, i = 1, 2, \cdots, n$, defines a dead mutant observation scheme $\mathcal{M}_i$, as in Example 3. We define the typed dead mutant observation scheme $\mathcal{M}_{Typed} = \mathcal{M}_1 \times \mathcal{M}_2 \times \cdots \mathcal{M}_n$. $\square$

D. Statistical Constructions

An observation scheme in the set construction or partially ordered set construction observes and records whether certain types of events happen during the testing process. Another type of observation often used in software testing is the statistics of the number or frequency of certain events that happened in testing. Let $\mathcal{B} : p \rightarrow \langle \mathbf{B}_p, \mu_p^B \rangle$ be an observation scheme and N be any given set of numbers. Then, $\langle N, \leq \rangle$ is a totally ordered set under the less than or equal to relation $\leq$ on numbers. We can define a scheme $\mathcal{A} : p \rightarrow \langle \mathbf{A}_p, \mu_p^A \rangle$ as follows.

Definition 7. *(Statistical construction)*
An observation scheme $\mathcal{A} : p \rightarrow \langle \mathbf{A}_p, \mu_p^A \rangle$ is said to be a statistical observation scheme based on $\mathcal{B} : p \rightarrow \langle \mathbf{B}_p, \mu_p^B \rangle$ if there exists a set N of numbers and a collection of mappings $s_{p \in P} : B_p \rightarrow N$ such that, for all software systems $p \in P$,

(1) $A_p = N$, and $\leq_{A,p}$ is the less than or equal to relation $\leq$ on N;
(2) The mapping s_p from B_p to the set N preserves the orders in B_p, i.e., $\sigma \leq_{B,p} \sigma' \Rightarrow s_p(\sigma) \leq s_p(\sigma')$;
(3) For all test sets T, $\mu_p^A(T) = \{s_p(\sigma) | \sigma \in \mu_p^B(T)\}$. □

Informally, the observable phenomena in a statistical construction are numerical values ordered as numbers. The mapping s_p can be considered as the measurement of the sizes of the phenomena observed by the base scheme. This size measurement must be consistent with the ordering on the phenomena in the base scheme. In other words, the more the information contained in a phenomenon observed by the base scheme, the larger the size of the phenomenon. For example, statement coverage is a statistical construction based on statement testing:

Example 7. (Statement coverage)
Let $\mathcal{B} : p \rightarrow \langle \mathbf{B}_p, \mu_p^B \rangle$ be the regular scheme for statement testing, where $\mathbf{B}_p$ is defined in Example 1. Define $s_p(\sigma) = \|\sigma\|/n_p$, where n_p is the number of statements in program p and $\|\sigma\|$ is the size of the set σ. We thus define a statistical observation scheme for statement coverage. The phenomena observed by the scheme are the percentages of statements executed during testing. □

Example 8. (Mutation score)
In mutation testing, mutation score is defined by the following equation and used as an adequacy degree of a test set [44, 45]:

$$MutationScore = \frac{DM}{NEM} \tag{11}$$

where DM is the number of dead mutants and NEM is the total number of non-equivalent mutants.

The mutation score can be defined as a statistical observation scheme based on the dead mutant observation scheme, defined in Example 3 with the mapping $s_p(\sigma)\|\sigma\|/m_p$, where $\|\sigma\|$ is the size of the set σ and m_p is the number of non-equivalent mutants of p generated by mutation operators. □

Note that the statement coverage scheme defined above is not decomposable, although the observation scheme for statement testing is a regular set construction that has decomposability according to Theorem 1. Similarly, the mutation score scheme does not have decomposability, while the dead mutation scheme has decomposability.

In software testing, statistics can also be made on the phenomena observed from testing on each test case. The following defines the construction of such schemes:

Definition 8. *(Case-wise statistical construction)*
An observation scheme $\mathcal{A} : p \rightarrow \langle \mathbf{A}_p, \mu_p^A \rangle$ is said to be a case-wise statistical observation scheme based on $\mathcal{B} : p \rightarrow \langle \mathbf{B}_p, \mu_p^B \rangle$ if there exists a set N of numbers and a collection of mappings $s_{p\in P} : B_p \rightarrow N$ such that, for all systems $p \in P$,

(1) $A_p = D_p \rightarrow N$, where $D_p \rightarrow N$ is the set of partial functions from D_p to N, and $\leq_{A,p}$ is defined by the equation

$$\sigma_1 \leq_{A,p} \sigma_2 \Leftrightarrow \forall t \in D_p.(\sigma_1(t) = undefined \vee \sigma_1(t) \leq \sigma_2(t)),$$

where $\leq$ is the less than or equal to relation on N;
(2) the mapping s_p from B_p to the set N preserves the order in B_p, i.e., $\sigma \leq_{B,p} \sigma' \Rightarrow s_p(\sigma) \leq s_p(\sigma')$;
(3) for all test sets $T = \{n_i t_i | t_i \in D_p, n_i > 0, i \in I, i \neq j \Rightarrow t_i \neq t_j\}$, we have that $\sigma_A \in \mu_p^A(T)$ iff
(a) $\forall i \in I.\exists \sigma_i \in \mu_p^B(\{n_i t_i\}).(\sigma_A(t_i) = s_p(\sigma_i))$, and
(b) $t \notin T \Rightarrow \sigma_A(t) = undefined$. □

Informally, a phenomenon in the universe A_p consists of a sequence of records. Each record represents the size of the phenomenon observed using the base scheme from the execution(s) of the concurrent system p on one test case. As in the statistical construction, the size function s_p must be consistent with the partial ordering relation defined on the base scheme.

Example 9. (Output diversity scheme)
The output diversity observation scheme defined in Example 4 is the case-wise statistical observation scheme based on the input/output observation scheme with the mapping s_p being the set size function. □

In [25], we studied the properties of the observation schemes defined above. Table 1 below gives the properties of the above constructions in terms of the axioms that they satisfy. It also gives a typical example of observation schemes in existing testing methods. Proofs of these properties can be found in [25].

Table 1. Properties of the constructions of observation schemes

		Properties (Axioms)								
Construction	Typical Examples	1	2	3	4	5	6	7	8	9
Regular set	Statement and branch testing Strong/weak mutation testing Def/Use dataflow testing Decision/condition testing Partition testing	√	√	√	√	√	√	√	√	√
Partially ordered set	Path testing Interaction chain of dataflow Definition context of dataflow Def/Use dataflow path testing	√	√	√	√	√	√	√	√	√
Product	Typed mutation testing	√	√	√	√	√	√	√	√	√
Statistics	Mutation score Statement/branch coverage Path coverage	√	√	√	√	√	√	√	×	×
Case-wise statistics		√	√	√	√	√	√	√	×	×

Table 1 presented the properties of product, and statistical and case-wise statistical constructions proved under the assumption that the base schemes satisfy the axioms.

3 Behavioral Observation in Component Integration Testing

In this section, we apply the theory to the integration testing in CBSD. We study the axioms of behavioral observation for component integration testing, propose a set of observation schemes inspired by the design patterns of observation schemes, and investigate how test drivers and component stubs should be used properly in incremental integrations.

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 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.

In this chapter, we are concerned with white-box integration testing (WIT) methods in which the code of glueware is available and used in testing. Using a WIT method, the tester observes the internal dynamic behavior 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.

A. Formal Model of White-Box Integration Testing

White-box integration testing methods can be formally defined using the theory of observation schemes as follows:

Definition 9. *(White-box integration testing methods)*
A white-box integration testing method contains an observation scheme $\mathcal{B} : p \rightarrow \langle \mathbf{B}_p, \mu_p \rangle$*. 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.* □

Note that the universe of observable phenomena of a component determined by a WIT method should also be a CPO set, which may have a different structure from the whole system. This is because in integration testing we usually focus on the interaction between the components and their environment instead of the details of the behavior of the component.

It is worth noting that in Def. 9 we have not assumed whether observations on the internal behavior of a component are available. As we will see later, the approach is applicable to situations both when the internal behavior is observable and when the inside information of the components is hidden.

By a well-defined WIT method, we not only require that the observation scheme B satisfy the axioms listed in the previous section, but also that the partial ordering $\leq_{c,p}$ on $B_{c,p}$ and the filter φ_c satisfy the following axioms:

Axiom 10. *(Filter's well-foundedness)*
If no observation on the whole system is made, nothing is known for the component. Formally,

$$\varphi_c(\perp_p) = \perp_{c,p}, \tag{12}$$

where $\perp_p$ *and* $\perp_{c,p}$ *are the least elements of* $\langle B_p, \leq_p \rangle$ *and* $\langle B_{c,p}, \leq_{c,p} \rangle$ *, respectively.* □

Axiom 11. *(Filter monotonicity)*
The more the behavior observed of the whole system, the more one knows about the component based on the observation. Formally,

$$\forall \sigma_1, \sigma_2 \in B_p, (\sigma_1 \leq_p \sigma_2 \Rightarrow \varphi(\sigma_1) \leq_{c,p} \varphi(\sigma_2)). \tag{13}$$

□

Axiom 12. *(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,

$$\forall \Theta \subseteq B_p, \left(\varphi_c(\bigsqcup_{\sigma \in \Theta} \sigma) \bigsqcup_{\sigma \in \Theta} \varphi_c(\sigma) \right). \tag{14}$$

□

In white-box integration testing, we usually integrate a number of components into the system at the same time. Therefore, we generalize the notion of filter to a set of components. Let C be a set of components. A filter φ_C for a set C of components is therefore a mapping from the universe of the observable phenomena of the whole system p to the universes of observable phenomena of the component set C. We require that φ_C also satisfy the extended version of the above axioms, which are obtained by replacing φ_c with φ_C. Moreover, we require that for each c in C, there be a function ϑ_c such that $\varphi_c = \vartheta_c \circ \varphi_C$, where φ_c is the filter for the component c.

B. Some Basic WIT Observation Schemes

The following defines a hierarchy of observation schemes for component integration testing. Note that, although components may have structures, we will treat components as black boxes at the moment. We will discuss how the structure of components can be utilized in integration testing in Sect. 3.2.

(a) Interaction Statement Testing

This method is based on the regular set construction of observation schemes. We define the atomic events to be observed as executions of the statements in the glueware. Therefore, the set of statements executed during integration testing is observed and recorded. These sets of statements include activities related to interactions with the components, such as

(a) Initiating the execution of a component as a process or thread,
(b) Establishing or destroying a communication channel with a component,
(c) Creating or destroying an instance of a class defined in a component,
(d) Registering or unregistering a component into a system,
(e) Subscribing the data produced by a component,
(f) Publishing data that is subscribed,
(g) Invoking a component, as a function or procedure defined in a component,
(h) Sending a message to a process or thread of a component,
(i) Receiving a message from a process or thread of a component.

As in statement testing in unit testing, the details of the executed statement and their sequences of executions are not recorded. The method does not require observing and recording the execution of the statements inside a component. Therefore, the components are treated as black boxes. Formally, the interaction statement testing method can be defined as follows:

Definition 10. *(Interaction statement testing)*
Let U_p be the set of statements in the glueware of a component-based software system p. For each component c in the system, $U_{c,p} \subseteq U_p$, where $U_{c,p}$ is the subset of statements in the glueware that are related to the component c. The observation scheme $\mathcal{IS}_p$ of interaction statement testing is the regular set construction based on U_p. The observation scheme $\mathcal{IS}_{c,p}$ for a component c in p is the regular set construction based on $U_{c,p}$. The filter function φ_c for component c removes the statements not related to the component. That is, for all $S \subseteq U_p$, $\varphi_c(S) = \{s | s \in S, s \in U_{c,p}\}$. An adequacy criterion $\mathcal{ISC}_c$ for statement coverage of interaction with component c can be defined as follows:

$$\mathcal{ISC}_c = \frac{\|\varphi_c(S)\|}{\|U_{c,p}\|}, \tag{15}$$

where S is the set of statements in p executed during testing, ISC_c is the interaction statement coverage with respect to c. □

It is easy to see that this testing method satisfies all the axioms of behavioral observation, as well as the axioms of filters in integration testing.

(b) Parameter Testing

The parameter testing method improves the observation on the dynamic behavior of a system by recording the set of component-related events with more details about the interactions with a component. Atomic activities in the interactions with a component often have parameters. For example, a call of a function/procedure defined in a component usually has parameters such as the values passed to the component in value parameters and values received from the component in variable parameters. Similarly, for a message passing event, the message also has contents. The values and contents passed across the interface of a component are not observed and recorded in interaction statement testing, but rather in parameter testing. In particular, in addition to what is observed in statement integration testing, parameter testing also observes and records the following information, and associates the information with the statements:

(a) The parameters used for initiating the execution of a component as a process or thread, if any, such as the name of the process, the value used for initialization of the process, etc.
(b) The parameters used for establishing or destroying a communication channel with a component, if any, such as the name, the identity number

of the communication channel, and the parameters used to set up the communication channel.

(c) The parameters used for creating or destroying an instance of a class defined in a component, such as the initial values used as parameters for the constructor.

(d) The parameters used for registering or unregistering from a component to a system, such as the name, network address, and any parameters of the component.

(e) The parameters used to subscribing for the data produced by a component, such as the name and/or format of the data.

(f) The details of publishing data that is subscribed to such as the value of the data, and any meta data such as format.

(g) Parameters used for invoking a component, such as the values of parameters and the names of the variable parameters in the invocations of functions or procedures defined in a component.

(h) The contents of messages sent to or received from a process or thread of a component, as well as any meta data associated with the message.

As with to interaction statement testing, this method itself does not require that the events happening inside a component be observed. It also treats components as a black box.

This scheme also has a set construction, but the base set is slightly more complicated here than in interaction statement testing. An element in the base set can be in the form of $\langle statement\ label, parameters\rangle$, which indicates that a statement is executed with its parameters. The observation scheme can be formally defined as follows:

Definition 11. *(Parameter testing)*
Let V_p be the set of statement-parameter pairs in the glueware of a component-based software system p, and $V_{c,p} \subseteq V_p$ be the subset of statement-parameter pairs where the statements are related to component c. The observation scheme $\mathcal{PT}_p$ of parameter testing is the regular set construction based on the set V_p. For each component c in the system p, the observation scheme $\mathcal{PT}_{c,p}$ for c is the regular set construction based on $V_{c,p}$. The filter function φ_c for component c removes the statement-parameter pairs that are not related to component c. That is, for all $S \subseteq V_p$,

$$\varphi_c(S) = \{s | s \in S \wedge s \in V_{c,p}\}.$$

□

An adequacy criterion for parameter testing cannot be defined as easily as interaction statement coverage because the set of statement-parameter pairs can be infinite if the parameter is, for example, a real number. Various methods to simplify the adequacy measurement can be applied to define practically usable adequacy criteria. For example, the domain of a parameter can be divided into a finite number of sub-domains so that each sub-domain is covered by at least one test case.

Note that, first, the observation scheme of statement testing is an extraction of the parameter observation scheme. This directly follows from Theorem 1. Second, the observation scheme of parameter testing is also a set construction. Therefore, it satisfies all the axioms of observation schemes. Finally, it is easy to prove that the filter satisfies the axioms of filters.

(c) Interaction Sequence Testing

Now, let us apply the partially ordered set construction to define a testing method similar to path testing.

The interaction sequence testing method for component integration testing observes and records the execution sequences of the statements in the glueware. Note that a component is still regarded as a black box. The base set is the set of paths in the glueware. Each path is a sequence of statements in the base set of interaction statement testing. There is a partial ordering between execution paths, which is the sub-path/super-path relation.

It is worth noting that interaction sequence testing in component integration can be applied not only to observe the sequences of interactions between a component and its environment (i.e., the glueware), but also to observe the interactions among a set of components. Therefore, the following defines the method in its most general form for integrating a set of components.

Definition 12. *(Interaction sequence testing)*
Let C be a set of components integrated into a component-based software system p, U_p be the set of interaction statements in the glueware of the system as defined in Def. 10 of interaction statement testing, and $U_{C,p}$ be the subset of U_p that is related to components in C as also defined in Def. 10. The observation scheme $\mathcal{ISQ}_p$ of interaction sequence testing is the partially ordered set construction based on $\langle Seq(U_p), \sqsubseteq \rangle$, where $Seq(X)$ is the set of finite sequences of elements in set X, and $\sqsubseteq$ is the sub-sequence/super-sequence relation between the sequences. The observation scheme $\mathcal{ISQ}_{C,p}$ of the component set C in p is the partially ordered set construction based on $\langle Seq(U_{C,p}), \sqsubseteq \rangle$. The filter function φ_C for component set C removes the statements not related to the components in C from the sequences of statements. That is, for each sequence q in $Seq(U_p)$, φ_C produces a sequence q' in $Seq(U_{C,p})$ by removing all the statements not in the set $U_{C,p}$. □

With this observation scheme, a variety of adequacy criteria can be defined, such as the coverage of simple sequences, which have no elements that appear more than once, and various path coverage criteria for loops (see, e.g., [46]).

Note that a finite sequence in the set $Seq(U_p)$ is a sequence of statements in p that represents an execution of the system. A finite sequence in the set $Seq(U_{C,p})$ removes any statement that is not related to interaction with the components in C. Therefore, it focuses on the interaction process between the glueware and the components.

From the properties of partially ordered constructions of observation schemes, we can prove that the observation scheme of interaction sequence testing satisfies all the axioms discussed in Sect. 2. It is also easy to prove that the filter function φ_C satisfies all the axioms of filters.

(d) Information Flow Testing

Similar to the interaction sequence testing method, information flow testing method is a generalization of parameter testing by applying the partially ordered set construction. The basic elements of observable phenomena are sequences of statement-parameter pairs. Each sequence records the execution of the system with detailed information about what has been passed across the interface to and/or from the components.

Definition 13. *(Information flow testing)*
Let C be a set of components integrated into a component-based software system p, V_p be the set of statement-parameter pairs in the glueware of the system as defined in Def. 11 of parameter testing, and $V_{C,p}$ be the subset of V_p that are related to components in C as defined also in Def. 11. The observation scheme $\mathcal{IFL}_p$ of information flow testing is the partially ordered set construction based on $\langle Seq(V_p), \sqsubseteq \rangle$. The observation scheme $\mathcal{IFL}_{C,p}$ of the component set C in p is the partially ordered set construction based on $\langle Seq(V_{C,p}), \sqsubseteq \rangle$. The filter function φ_C for component set C removes the statements not related to the components in C from the sequences. That is, for each sequence q in $Seq(V_p)$, φ_C produces a sequence q' in $Seq(V_{C,p})$ by removing all the elements not in the set $V_{C,p}$. $\Box$

The following theorem states the extraction relationships between the observation schemes defined above. They are also show in Fig. 2.

Theorem 2. *(Extraction theorem of basic WIT methods)*
The following extraction relations hold between the observation schemes defined above.

(1) $\mathcal{IS} \triangleleft \mathcal{PT}$; (2) $\mathcal{IS} \triangleleft \mathcal{ISQ}$; (3) $\mathcal{ISQ} \triangleleft \mathcal{IFL}$; (4) $\mathcal{PT} \triangleleft \mathcal{ISQ}$. $\Box$

The proof of Theorem 2 is straightforward.

3.2 Hierarchical Integration Testing

In large-scale and complicated component-based software systems, a component may also be a composition of other components. We call the components that directly constitute a software system the 1st order components. We call the components in a 1st order component the 2nd order components. Similarly, we define 3rd order components as components of 2nd order components,

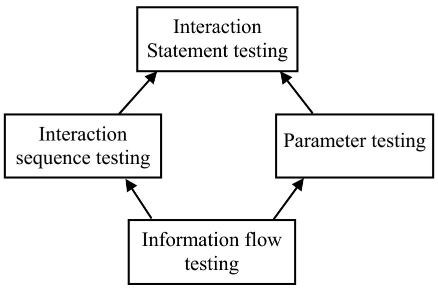

Fig. 2. The extraction relations between the testing methods

and so on. We use high order components to denote all the components of any order. Therefore, each component can be modeled as a software constructor that consists of glueware and a number of higher order components. A component is called atomic if it does not contain any higher order components. In this case, the glueware is just the code of the component. Figure 3 below illustrates this view of a software system's structure. In the diagram, the overall system consists of three 1st order components B_1, B_2, and B_3. Each of these 1st order components is composed of some 2nd order components. For example, B_1 is composed of components $C_{1,1}$ and $C_{1,2}$. Component $C_{1,2}$ is composed of another smaller component, while $C_{1,1}$ is atomic.

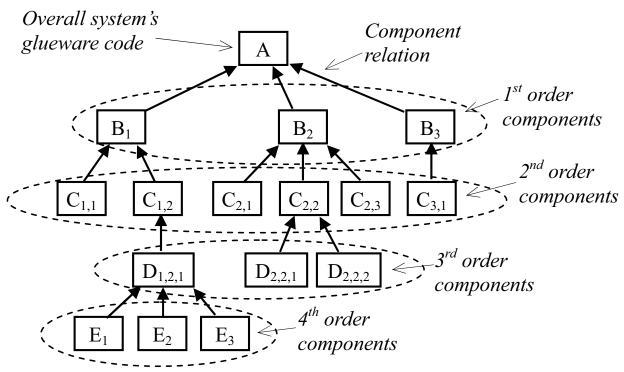

Fig. 3. Illustration of system structure from component view

The integration testing methods studied in the previous section treat glueware as a white box and all components as black boxes. We call such a testing method a ground order white box integration testing method. For each ground order method, we can generalize it to treat the 1st order components as white boxes and observe the same aspect of behavior inside the component. Meanwhile, the components of higher order than the 1st order are still treated as black boxes. We call such a generalized method a 1st order WIT method.

For example, the method obtained by generalizing the ground order interaction statement testing to the 1st order observes the statements at the architectural level glueware executed during testing, as well as the statements inside the 1st order components. All the statements of the glueware of the 1st order components are considered as related to the component. Therefore, the interaction statement coverage for integrating the component requires not only executing all the statements in the glueware of the whole system, but also all the statements of the glueware of the component.

For the interaction sequence testing of component integration, the generalization of the method from ground order to the 1st order requires testers to observe the execution paths that cross the interface of the 1st order components while treating higher order components as black boxes. Similarly, when the information flow testing method is generalized from ground order to the 1st order, testers are required to observe the execution paths that represent information flows between the glueware and the 1st order components. Such a path may start from the glueware and flow into the body of the 1st order components, then flow inside the body of the 1st order components, and finally come out the body, with the information being received and processed in the glueware. Before the execution path finishes in the glueware, it may well flow into other 1st order components.

A 1st order WIT method will not observe the same detail in the behavior of components of the 2nd and higher orders. It can be further generalized to kth order for any given natural number $k > 1$ by observing the same detail in the kth order components, while treating components of $(k+1)$th order as black boxes. The most powerful method is to treat all high order components equally, as white boxes. Such a method is called an infinite order WIT method. Figure 4 below illustrates the 2nd order WIT testing method, where all 2nd order components are treated as white boxes and higher order components as black boxes, shaded in the diagram.

Let $\mathcal{Z}$ be any given ground order WIT testing method. We write $\mathcal{Z}^{(k)}$ to denote the kth order generalization of $\mathcal{Z}$, and $\mathcal{Z}^{(\infty)}$ to denote the generalization of $\mathcal{Z}$ to an infinite order. It is easy to see that these observation schemes have the following extraction relationship. Its proof is omitted.

Theorem 3. *(Extraction relations on generalizations)*
For all WIT testing methods $\mathcal{Z}$, we have $\mathcal{Z} \lhd \mathcal{Z}^{(1)} \lhd \cdots \lhd \mathcal{Z}^{(K)} \lhd \mathcal{Z}^{(K+1)} \lhd \cdots \lhd \mathcal{Z}^{(\infty)}$, for all $K > 0$. $\Box$

The generalization of ground order WIT testing methods also preserves the extraction relations. Formally, we have the following theorem [47].

Theorem 4. *(Preservation of extraction relations by generalizations)*
For all WIT testing methods $\mathcal{X}$ and $\mathcal{Y}$, we have that for all $n = 1, 2, \cdots, \infty$, $\mathcal{X} \lhd \mathcal{Y} \Rightarrow \mathcal{X}^{(n)} \lhd \mathcal{Y}^{(n)}$. $\Box$

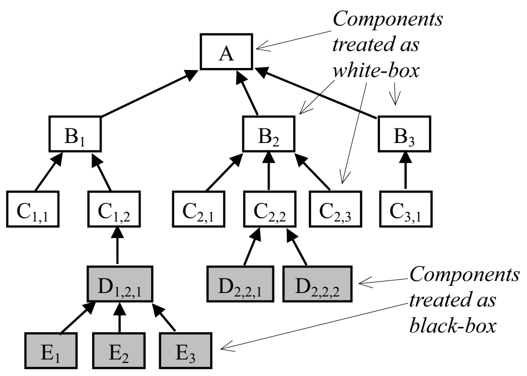
Components treated as white-box
A
B1
B2
B3
C1,1
C1,2
C2,1
C2,2
C2,3
C3,1
D1,2,1
D2,2,1
D2,2,2
Components treated as black-box
E1
E2
E3

Fig. 4. Illustration of 2nd order WIT testing method

A ground order WIT testing method can also be generalized heterogeneously so that some kth order components are treated as white boxes and some as black boxes. Figure 5 illustrates a situation in heterogeneous higher order WIT testing, where shaded components in the diagram are treated as black boxes and the others are treated as white boxes.

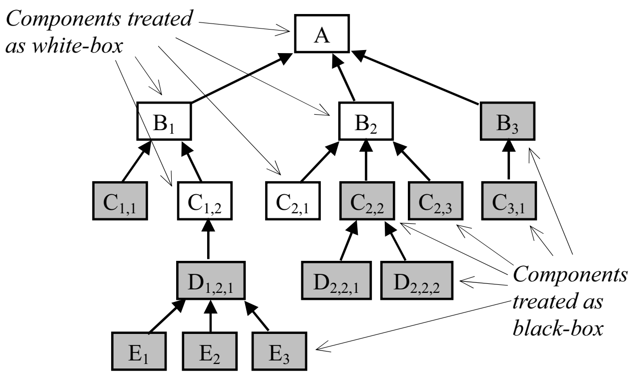
Components treated as white-box
A
B1
B2
B3
C1,1
C1,2
C2,1
C2,2
C2,3
C3,1
D1,2,1
D2,2,1
D2,2,2
Components treated as black-box
E1
E2
E3

Fig. 5. Illustration of heterogeneous higher order WIT testing

Let C be a subset of a system p's components that is to be treated as a white box in testing.

Definition 14. *(Consistent subset of components for WIT testing)*
A subset C of a system p's components is said to be consistent for WIT testing if a component of p being treated as a black box implies that all its subcomponents are also treated as black boxes. Formally, $\forall c \in C, (c \notin C \wedge (x \textit{ is a subcomponent of } c) \Rightarrow (x \notin C))$. □

We write $\mathcal{Z}^{(C)}$ to denote the application of WIT testing method $\mathcal{Z}$ on software system p with a consistent collection C of components as white boxes and all other components as black boxes. Let C_1 and C_2 be two collections of components in a software system p. From Theorem 1, we can prove the following:

Theorem 5. *(Extraction relation on heterogeneous WIT testing)*
For all consistent collections C_1 and C_2 of components of any given software system p, $C_1 \subseteq C_2$ implies $\mathcal{Z}^{(C_1)} \triangleleft \mathcal{Z}^{(C_2)}$, i.e., $\mathcal{Z}^{(C_1)}$ is an extraction of $\mathcal{Z}^{(C_2)}$.
$\square$

3.3 Incremental Integration Testing

In practice, integration testing is often carried out incrementally as components are gradually integrated into the system. Integration strategies such as top-down, bottom-up, and their combinations 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 behavioral observation theory.

For the sake of simplicity, we subsequently assume that program constructors are binary, i.e., they take two components as parameters. The result can be easily generalized to constructors of any number of components. Let $\otimes$ be a binary program constructor; $p = c_1 \otimes c_2$, where c_1 and c_2 are components. A component itself may be a composition of some other components, and formed by applying a program constructor, say, $c_1 = c_{1,1} \oplus c_{1,2}$.

A. Bottom-Up Integration Strategy

By applying the bottom-up strategy, we first put $c_{1,1}$ and $c_{1,2}$ together to form $c_1 = c_{1,1} \otimes c_{1,2}$ and test c_1 with a test driver to replace the constructor $\otimes$. After successfully testing c_1 and c_2 in this way, they are put together to form $p = c_1 \otimes c_2$ and tested. A test driver is in fact a program constructor, $\otimes'$, which, when applied to c_1, forms an executable program p'. During this testing process, we would like the test driver to act like the environment of c_1, as would be the case in the real program p. This means that if we can observe the behavior of component c_1 in the context of $\otimes$, we should be able to observe the same behavior in the context of the test driver $\otimes'$. Suppose that we use a WIT method with observation scheme $\mathcal{B} : p \rightarrow \langle \mathbf{B}_p, \mu_p \rangle$. Hence, there is a filter φ from p to c_1 and a filter φ' from p' to c_1. The requirements for a well developed test driver can be specified by the following axiom:

Axiom 13. *(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 exist test suites T' for p' such that the same phenomena σ can be observed by executing p' on test suite T'. Formally, $\forall T \in \mathbf{T}_p$,

$$\forall \sigma \in \mu_p(T), \exists T' \in \mathbf{T}_{p'}, \exists \sigma' \in \mu_{p'}(T'), (\varphi(\sigma) \leq_{c_1} \varphi'(\sigma')), \tag{16}$$

where $\leq_{c_1}$ is the partial ordering on B_{c_1}. □

Note that Eq. 16 can be equivalently expressed as follows:

$$\forall T \in \mathbf{T}_{c_1 \otimes c_2}, \exists T' \in \mathbf{T}_{\otimes'(c_1)}, \left(\varphi(\mu_{c_1 \otimes c_2}(T)) \sqsubseteq_{c_1} \varphi'(\mu_{\otimes'(c_1)}(T'))\right),$$

where $\varphi(X) = \{\varphi(x) | x \in X\}$, and $X \sqsubseteq_{c_1} Y$ if and only if $\forall x \in X, \exists y \in Y, (x \leq_{c_1} y)$.

A test driver 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 observes only the functional aspect of behavior, such a test driver satisfies the representativeness axiom if it can pass all valid inputs to the component and pass out the result of the component's execution.

Sometimes, test drivers are written to combine other testing tasks, such as automatic generation of test cases, or select test cases from a repository of test data. Such a test driver usually does not have representativeness, because it generates only input data in a sub-domain. Therefore, it limits the space of observable phenomena. For all the test cases in the sub-domain, we require that the test driver be representative. Hence, we have the following weak form of Axiom 13.

Axiom 14. *(Representativeness on a sub-domain)*
For all test suites T in a sub-domain $S \subseteq D_p$ of the valid input of a system p, and all observable phenomena of the component from executing p on T, there is a test suite T' for the test driver for which the same phenomena can be observed in the context of the test driver. Formally, for all $T \in \mathbf{T}_S \subseteq \mathbf{T}_p$,

$$\forall \sigma \in \mu_p(T), \exists T' \in \mathbf{T}_{p'}, \exists \sigma' \in \mu_{p'}(T'), (\varphi(\sigma) \leq_{c_1} \varphi'(\sigma')) \tag{17}$$

□

B. Top-down Integration Strategy

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

Axiom 15. *(Faithfulness of component stubs)*
For all test suites 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 a component c with a stub c'. Formally,

$$\forall T \in \mathbf{T}_p, (\mu_p(T) = \mu_{p'}(T)). \tag{18}$$

□

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 is concerned only with the functional aspect of a system. In that case, the stub is required to produce functionally correct outputs. Therefore, if a kth order WIT method is used, a component of $(k+1)$th or higher order can be replaced by a stub.

In software testing practices, stubs tend to provide only partial functionality of the component, and they faithfully represent the components' behavior only on a sub-domain of the component. This sub-domain is called the designated sub-domain of the stub. We require the stub to be faithful on the sub-domain. Hence, we have a weak form of Axiom 15. Assume that c is a component in system p, c' is a stub of c, and p' is obtained by replacing c in p with c'. We say that c' is faithful on a designated sub-domain S, if it satisfies the following axiom:

Axiom 16. *(Faithfulness of stubs on a 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 c with a stub c' if, during the executions of p on T, the component is executed only on the stub's designated sub-domain. Formally,

$$\forall T \in \mathbf{T}_p \swarrow (c, S), (\mu_p(T) = \mu_{p'}(T)),$$

where $\mathbf{T}_p \swarrow (c, S)$ is the subset of $\mathbf{T}_p$ on which p calls the component only c on the designated sub-domain S. □

The axioms of test stubs can also be extended for replacing a number of components by their corresponding stubs.

4 Conclusion

In this chapter, we reviewed the theory of behavioral observation in software testing and applied the theory to integration testing in component-based software development. We formalized the notion of white-box integration testing methods (WIT methods), in which the components can be treated as black boxes while the glueware of the system is treated as a white box. The basic properties of such testing methods are studied and a set of axioms is proposed to characterize well-defined observation schemes for such white box integration testing. We also proposed four basic observation schemes for WIT, proved their satisfaction of the axioms, and investigated their interrelationships. These methods of component integration testing are

(a) interaction statement testing, which focuses on the statements in the glueware that interact with the components,
(b) interaction sequence testing, which observes the execution sequences of the statements in the glueware that interact with the components,
(c) parameter testing, which focuses on the interactions between the glueware and the components by observing the parameters of each interactive action,
(d) information flow testing, which focuses on the information flow in the glueware in the context of interaction with the components by observing the sequences of interactive actions and their parameters in each test execution of the system.

When details of components are available, our white box integration testing methods also allow us to treat components as white boxes. In such cases, a component consists also of a glueware and a number of smaller components, which are called 2nd order or higher order components. In general, the components that directly comprise a software system are called 1st order components. The components that directly comprise a kth order component are called $(k+1)$th order components. The basic WIT methods can be generalized to a kth order testing method, which treats components up to kth order as white box, while any higher order components are treated as black boxes. These testing methods and their generalizations fall into a nice hierarchical structure according to the extraction relation. Therefore, according to the availability of the code of components, appropriate testing methods can be used to achieve required test adequacy.

Integration testing of complicated large-scale software systems must use appropriate integration strategies. These involve writing and using test drivers and/or component stubs to enable the integration strategy to be applied. In this chapter, we also investigated and analyzed the requirements of test drivers and component stubs in bottom-up and top-down integration strategies.

There are several directions for future work. First, there are 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 our understanding of the desirable properties of behavioral observation in integration, we will further investigate the algebraic structures of observable phenomena and their corresponding recording functions that satisfy these properties. The constructions of observation schemes that we proposed and investigated in [24] will also be further studied with regard to the axioms for integration testing. Finally, in [47] we have studied integration testing of software systems where components are integrated by applying parallel system constructors. We are further investigating some concrete system constructors that integrate components. In particular, we are applying the theory to specific component techniques.

5 Acknowledgements

This work was jointly funded by the NSF of USA under grant INT-0096143 and the NSF of China under grant 69811120643. X. He was also partially supported by the NSF of USA under grant HRD-0317692 and the NASA of USA under grant NAG2-1440. H. Zhu was also partially supported by China High-Technology Program under grant 2002AA116070.

References

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 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. Zhu, H., A note on test oracles and semantics of algebraic specifications, Proc. of QSIC'03, Oct. 2003, Dallas, USA, pp91-98.
9. Abdurazik, A. and Offutt, J., Using UML collaboration diagrams for static checking and test generation, Proc. UML'00, York, UK, Oct. 2000, pp383-395.
10. Barnett, M., Grieskamp, W., Schulte, W., Tillmann, N. and Veanes, M., Validating use-cases with the AsmL test tool, Proc. QSIC03, Oct. 2003, Dallas, USA, pp238-246.
11. 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.
12. Bertolino, A., Corradini, F., Inverardi, P. and Muccini, H., Deriving test plans from architectural descriptions, Proc. ICSE'2000, June 2000, Limerick, Ireland, pp220-229.
13. 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.
14. Zhu, H., Jin, L., Diaper, D. and Bai, G., Software requirements validation via task analysis, Journal of System and Software, March 2002, Vol 61, Issue 2, pp145-169.
15. Harrold, M,J., and Soffa, M.L., Selecting and Using Data for integration testing, IEEE Software, March 1991, pp58-65.
16. Frankl, P.G. and Weyuker, J.E., An applicable family of data flow testing criteria, IEEE TSE, Vol.SE_14, No.10, October 1988, pp1483-1498.

17. Ural, H. and Yang , B., Modeling software for accurate data flow representation, Proc. ICSE'93, May 1993, pp277-286.
18. Pandi, H. D., Ryder, B. G., Landi, W., Interprocedural Def-Use associations in C programs, Proc. TAV4, Oct. 1991, pp139-153.
19. Jin, Z. and Offutt, J., Integration testing based on software couplings, Proc. COMPASS'95, Gaithersburg, Maryland, June 1995, pp13-23.
20. 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.
21. Beydeda, S. and Gruhn, V., State of art in testing components, Proc. of QSIC03, Dallas, USA, Oct. 2003, IEEE Computer Society, pp146-153.
22. 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.
23. Zhu H. and He X., A Theory of Testing High-Level Petri Nets, Proc. of the IFIP 16th World Computer Congress, Beijing, China, Aug. 2000, pp443-450.
24. Zhu, H. and He, X., A methodology of testing high-level Petri nets, Information and Software Technology, Volume 44, Issue 8, June 2002, Pages 473-489.
25. Zhu H. and He X., Constructions of Behaviour Observation Schemes in Software Testing, Proc. HASE'00, New Mexico, Nov. 2000, pp2-12.
26. Taylor, R. N. Levine, D. L. and Kelly, C. D., Structural Testing of Concurrent Programs, IEEE Transaction on Software Engineering, Vol. 18, No. 3, Mar. 1992, pp206-215.
27. 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.
28. Goodenough, J. B. and Gerhart, S. L., Toward a theory of test data selection, IEEE TSE, Vol.SE_3, June 1975.
29. Budd, T. A. and Angluin, D., Two notions of correctness and their relation to testing, Acta Informatica, Vol. 18, 1982, pp31-45.
30. Weyuker, E. J., Axiomatizing software test data adequacy, IEEE TSE, Vol.SE_12, No.12, December 1986, pp1128-1138.
31. Cherniavsky, J. C. and Smith, C. H., A recursion theoretic approach to program testing, IEEE TSE, Vol. SE_13,No.7, July 1987, pp777-784.
32. Davis, M. and Weyuker E., Metric space-based test-data adequacy criteria, The Computer Journal, Vol.13, No.1, February 1988, pp17-24.
33. Weyuker, E.J., The evaluation of program-based software test data adequacy criteria, Communications of the ACM, Vol.31, No.6, June 1988, pp668-675.
34. Parrish, A. S. and Zweben, S. H., Analysis and refinement of software test data adequacy properties, IEEE TSE, Vol. SE_17, No. 6, June 1991, pp565-581.
35. Parrish, A. S. and Zweben, S. H., 1993, Clarifying Some fundamental Concepts in Software Testing, IEEE TSE, Vol. 19, No.7, July 1993, pp742-746.
36. Frankl, P. G. and Weyuker, J. E., A formal analysis of the fault-detecting ability of testing methods, IEEE TSE, Vol. 19, No. 3, March 1993, pp202- 213.
37. Zhu, H. and Hall, P., Test data adequacy measurement, SEJ, Vol. 8, No.1, Jan. 1993, pp21-30.
38. Zhu, H., A formal analysis of the subsume relation between software test adequacy criteria, IEEE Transactions on Software Engineering, Vol. 22, No. 4, April 1996, pp248-255.

39. Zhu, H. Hall, P. and May, J., Software Unit Test Coverage and Adequacy, ACM Computing Survey, Vol. 29, No. 4, Dec. 1997, pp366-427.
40. Myers, G. J. , The Art of Software Testing, John Wiley and Sons, New York, 1979.
41. Beizer, B., Software Testing Techniques, 2nd Edition, New York, Van Nostrand Reinhold, 1990.
42. Howden, W. E., Reliability of The Path Analysis Testing Strategy, IEEE Transaction on Software Engineering, Vol. SE-2, No. 9, Sept. 1976, pp208-215.
43. King, K. N. and Offutt, A. J., A FORTRAN Language System for Mutation-Based Software Testing, Software–Practice and Experience, Vol. 21, No. 7, Jul. 1991, pp685-718.
44. DeMillo, R. A., Lipton, R. J. and Sayward, F. G., Hints on Test Data Selection: Help for The Practising Programmer, IEEE Computer, Vol. 11, Apr. 1978, pp34-41.
45. Budd, T. A., Mutation Analysis: Ideas, Examples, Problems and Prospects, In Computer Program Testing, Chandrasekaran and Radicchi (eds.), North Holland, 1981, pp129-149.
46. Zhu, H., Axiomatic assessment of control flow based software test adequacy criteria, Software Engineering Journal, UK, Sept. 1995.
47. Zhu, H. and He, X., An Observational Theory of Integration Testing for Component-Based Software Development, Proc. of COMPSAC2001, Oct. 2001, Chicago, Illinois, USA, pp363-368.