Structural Testing of Component-Based Systems

Daniel Sundmark, Jan Carlson, Sasikumar Punnekkat, and Andreas Ermedahl

MRTC, Mälardalen University
Box 883, SE-721 23 Västerås, Sweden
daniel.sundmark@mdh.se

Abstract. Component based development of software systems needs to devise effective test management strategies in order fully achieve its perceived advantages of cost efficiency, flexibility, and quality in industrial contexts. In industrial systems with quality demands, while testing software, measures are employed to evaluate the thoroughness achieved by execution of a certain set of test cases. Typically, these measures are expressed in the form of *coverage* of different structural test criteria, e.g., statement coverage. However, such measures are traditionally applicable only on the lowest level of software integration (i.e., the component level). As components are assembled into subsystems and further into full systems, general measures of test thoroughness are no longer available. In this context, we formalize the added test effort and show to what extent the coverage of structural test criteria are maintained when components are integrated, in three representative component models. This enables focusing on testing the right aspects of the software at the right level of integration, and achieves cost reduction during testing — one of the most resource-consuming activities in software engineering.

1 Introduction

The component-based development paradigm has been quite successful in enterprise computing and is being explored as an attractive option in other domains with quality demands, e.g., embedded software systems. However, in order to gain wide acceptance in such domains, the component-based approach also needs to devise efficient testing strategies and test management approaches, since testing accounts for a lion's share of the development cost in such systems. Quality concerns also demand that the developer presents evidence of thoroughness of verification efforts performed. Structural coverage is a set of measures for evaluating the thoroughness of software testing, based on how exhaustively the tests exercise certain aspects of the structure of the software under test [1]. Test criteria based on structural coverage are well-defined for component-level testing [1, 2], but most of these definitions are not generally applicable for testing performed post-integration, where issues of, e.g., multi-tasking and shared resources come into play.

Building a system out of well-tested components does not necessarily result in a well-tested system. However, during integration testing, it is possible to make use of the information available on what aspects of the software that have already

been tested before integration. Ideally, during integration testing, testing should only focus on the correctness of the actual interaction between the integrated components.

The contribution of this paper is twofold. First, we describe the added test effort required by component integration, by introducing the concept of *compositionally introduced test items*. Second, we describe the impact of this concept for a number of common structural test criteria and three representative component models. We also outline how this concept can be used in order to maintain the quality assurance achieved by structural test coverage at an arbitrary level of integration in multi-level hierarchical development of component-based software.

The main motivation for this work is that it facilitates a less ad-hoc way of determining the adequacy of integration- and system-level testing in addition to the functional testing traditionally used at this level, while also clearly separating the test items that could be tested at component-level from those that need to be tested post-integration.

2 Background

Testing is the primary means for verification used in the software industry, and methods and strategies for testing come in many different shapes. Depending on the type of software system to be developed or maintained, hypotheses regarding the types of bugs suspected in the software, and many other aspects, the testing approach will be different. There is, however, a common ground for reasoning about test techniques.

2.1 Software Testing and Coverage

A *test criterion* is a specification for evaluating the test adequacy given by a certain set of test cases. Test criteria are defined in terms of *test items*, the "atoms" of test criteria. A test criterion is generally formulated such that test adequacy (with respect to that criterion) is attained when all test items are exercised during testing. For example, for the statement coverage criterion, statements are the test items. Test items are also called coverage items. *Coverage* is a generic term for a set of metrics used for expressing test adequacy (i.e., the thoroughness of testing or determining when to stop testing with respect to a specific test criterion [1]). A coverage measure is generally expressed as a real number between 0 and 1, describing the ratio between the number of test items exercised during testing and the overall number of test items. Hence, a statement coverage of 1 implies that all statements in the software under test are exercised. Rules for when to stop testing can be formulated in terms of coverage. For example, a statement coverage of 0.5 (indicating that half of the statements in the software are exercised) may be a valid, if not very practical, stopping rule.

Test criteria may be *structural* or *functional*, where structural test criteria are based on the actual software implementation, or on abstract representations of the software implementation (e.g., control flow graphs). As *structural* test

criteria are strictly based on the actual software implementation and different inherent aspects of its structure, these are possible to define formally. Examples of structural test criteria include exercising of all instructions, all execution paths, or all variable definition-use paths in the software. It should be noted here that a full coverage (i.e., a coverage of 1) is not generally achievable for structural test criteria. This is due to the fact that the control flow graph representations used to define these criteria typically contain infeasible paths or statements (i.e., paths or statements that are not actually exercisable when executing the code), and the exact determination of feasible paths and statements is undecidable [3, 4].

On the other hand, test case selection based on *functional* test criteria is, in the general case, ad-hoc in the sense that it depends on the quality, expressiveness, and the level of abstraction of the specification. Basically, a more detailed and thorough specification will result in a more ambitious and thorough functional test suite. Examples of functional test techniques are boundary value testing and equivalence class partitioning testing based on the software specification [5].

In the traditional view of the software engineering process, verification testing is performed at different levels. Throughout the literature, many such levels are discussed, but the most commonly reappearing levels of verification testing are *component, integration* and *system testing* [5, 6], see Fig. 1. **Component testing** (also known as unit testing) is performed at the "lowest" level of software development, where the smallest units of software are tested in isolation. Such units may be functions, classes or components. **Integration testing** can be performed whenever two or more components are assembled into a system or a subsystem. Specifically, integration testing focuses on finding failures that are caused by interaction between the different components in the (sub)system. **System testing** focuses on the failures that arise at the highest level of integration [7], where all parts of the system are incorporated and executed on the intended target hardware(s). The execution of a system-level test case is considered correct if its output and behaviour complies with what is stated in the system specification.

Generally, the lower the level of testing, the more likely that both structural and functional criteria will be considered. In traditional system-level testing, only functional criteria are considered [6], and integration testing poses several problems for structural criteria, e.g., definition of control- and data-flow structures over component boundaries. Hence, as we recognize that structural and functional techniques complement each other by focusing on different aspects of the same software, this paper aims at facilitating the additional use of structural criteria on higher levels of integration. It is our firm belief that, compared to the traditional testing performed at the higher levels of integration, a combination of structural and functional testing will provide a more thorough software verification.

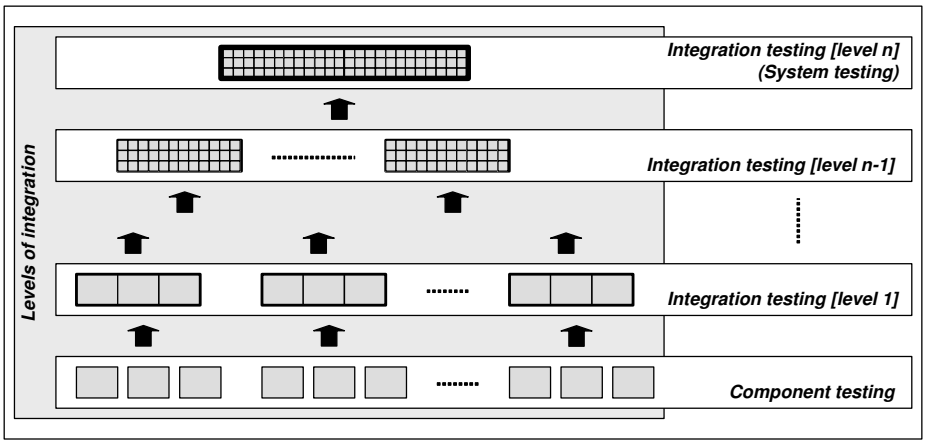

Fig. 1. Testing/integration levels in the software development process.

2.2 CBSE and Structural Software Testing

During component integration, the control- and data flow may be modified or compromised, and coverage based on these concepts may be invalidated. Considering the system-level counterparts of control- and data flow, there are no widely accepted general definitions of these concepts. Given unrestricted component interaction, the control and data flow of a component assembly may exhibit an unmanageable complexity. However, in practice the interaction between the software parts in an integration is restricted by several factors, e.g., the run-time system, and the architectural style used. To prevent an overwhelming complexity, it is often desirable to adopt some level of component or unit isolation.

In Component-Based Software Engineering (CBSE), software applications are built by composing software components into component assemblies [8, 9]. The main idea is that, by building systems out of well-tested components, an increase in the predictability of the behaviour of the software could be gained; provided that experience from component testing is taken into account.

Components are independent software units that interact via well-defined interfaces. According to the basic CBSE principles, there should be no hidden dependencies between components, except for those explicitly represented in the component interfaces. This facilitates reuse, allowing a component to be replaced without affecting the other components in the system.

In the context of structural testing at higher levels of integration, the additional information provided by component interfaces could be exploited while reasoning about the control and data flow in an assembly, and, in a later stage, when generating test cases. Thus, with a strong notion of component interface, the use of CBSE gives benefits during integration level testing, compared to traditional approaches where the corresponding information must be derived from low level code.

In general, system composition out of a set of components is guided by the architectural style chosen for the system. According to Shaw and Garlan [10], examples of such architectural styles include:

- **Dataflow systems**, which include systems based on *pipes and filters*, where components act as filters of data, and the interconnections between components act as data pipelines. This type of system typically manipulates sequential streams of data passed through components by pipelines.
- **Call-and-return systems**, e.g., object oriented systems, where the components (objects) of the system interact through the use of inter-component method calls.
- **Independent components**, e.g., event-based systems, uses an approach where the invocation of components and component methods are not triggered by explicit calls from other components, or on the explicitly stated data flow through the system, but rather on the occurrences of internal or external events.

In the following section, we will consider instances of these styles in order to see how component composition according to each instance affects component interaction, and the structural testing thereof. It should be noted that Shaw and Garlan [10] also mention **virtual machines** and **data centered systems (repositories)** as examples of architectural styles, but we will not consider them in this paper.

3 Structural Testing of Component-Based Systems

In this section we describe what is required to achieve structural test coverage for component assemblies, including whole component-based systems. In doing this, we aim at a software development process where the knowledge of what has already been tested, and the effects of component interaction, are used in order to perform a more conscious, non-ad-hoc integration testing. Ideally, we consider a process as the one described in Fig. 2, where the composition of a set of components (1) is followed by an analysis determining if any further testing is required to maintain the desired coverage (2). If such testing is required, we generate (3) a set of test cases required to achieve the desired coverage, whereafter test execution (4) and evaluation (5) follows, leading to an integrated component assembly (or system) with the desired coverage maintained (6).

In doing this, our primary goal is to find a set of test cases that are required in order to safely cover the aspects of the software added by component integration. The secondary goal would be to find the minimal set of test cases that fulfils this criterion. Unfortunately, since we are generally unable to determine exactly which test items are feasible (i.e., executable on the level of integration where the testing is performed), any analysis performed to safely determine feasible test items will be over-approximative, and potentially find false positives [11]. Once again, it should be noted that this problem is not unique to the higher levels of testing we consider in this paper (even though it is likely to be more severe),

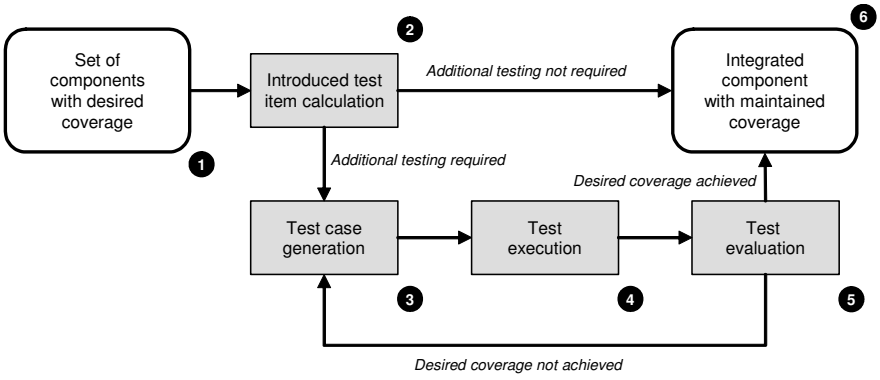

Fig. 2. An outline of the envisioned process.

since not all items are feasible on component-level, and the exact determination of feasible test items, even on component-level, is provably incalculable [3, 4].

Also, note that some test items that are feasible when testing a component in isolation might be made infeasible by system integration [12]. For example, a definition of a shared variable in one component may influence the flow of control and make paths considered feasible in other components infeasible [13–15]. This should be considered when performing testing of multi-tasking (and parallel) systems.

3.1 Impact of Architectural Style

The architecture of the software under test will affect how test criteria will be affected by component composition, since it, to a large extent, determines the means of inter-component communication. Further, the choice of architectural style to different degrees limits the component interaction, e.g., in terms of temporal perturbation.

Although the proposed approach is not limited to a particular component model or architectural style, it is exemplified by representatives of three different architectural styles:

- **CM1:** As an example of the dataflow style, we consider a component model (see Fig. 3a), similar to that of SaveCCM [16] or PECOS [17]. Contrasting, e.g., the UNIX pipes-and-filters architecture where the filters execute concurrently, we consider an interleaved model where components execute non-preemptively. When activated, a component consumes one set of input data and then executes to completion.
- **CM2:** Representing call-and-return systems, we consider a more traditional model (see Fig. 3b), where components are invoked by method calls. Examples of such models include Sun's JavaBeans [18], Microsoft's COM [19], and the Koala component model [20].

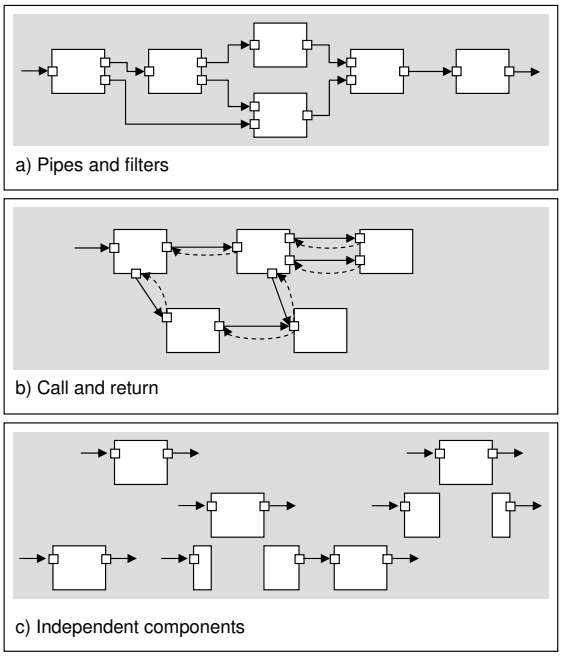

Fig. 3. Architectural Styles.

– **CM3:** As a third example, covering the architectural style of independent components, we consider a component-based preemptive real-time system where components correspond to individual tasks executing on an underlying real-time operating system (see Fig. 3c). Here, we consider components that are strictly periodic, inter-arrival and assume that execution is controlled by a system-level scheduler that distributes computation among the components, based on, e.g., priority levels. Moreover, components are preemptive, meaning that the scheduler is allowed to interrupt a component during its execution, should a component of higher priority level become available for execution. Examples of component models of this type include Rubus [21] and Autocomp [22].

It should be noted that the components in the third example are only independent in the sense that any transfer of control between components is initiated by the system-level scheduler, and not from within a component. In general, the components are not functionally independent, since they may communicate via shared memory. Also, since tasks compete for the same computational resources, they are clearly not independent with respect to timing.

It should also be noted that different architectural styles are sometimes adopted at different integration levels of the same system, as in COMDES [23] or ProCom [24]. For example, a large system might be built from a few indepen-

dent components, each of which in turn can be further decomposed into smaller components interacting in a pipes and filter fashion. The aim of our method is that the choice of architectural style at lower levels of integration should be transparent when determining the test coverage at a specific level of integration.

3.2 Compositionally Introduced Test Items

In this section, we describe what needs to be covered by testing on a certain level of component integration in order to maintain the coverage held by the components to be integrated. This is done by defining the concept of *compositionally introduced test items*. Simplified, these are the test items, given a certain test criterion, that only exist on the current level of integration and above. These additional test items particularly have two sources: The interaction between the integrated components, and extra code added merely for the integration of the components. Starting with the latter, during the development of systems based on components there could be "extra" code involved, depending on the architectural style followed. Particularly, this extra code could comprise of different categories such as:

- operating system code (e.g., driver routines, task switch routines and other system functions); and
- glue code, written or automatically generated to connect components, and for configuration and initialization.

For simplicity, we will consider the extra code as completely untested in the remainder of this paper.

Given the existence of previously non-covered additional code, one will have to pay special attention to ensure test coverage of this code during the higher level integration. Also, even if the additional code is already covered, it might give rise to additional test items for some test criteria and architectural styles, caused by its interaction with the components in the assembly.

Before formally defining the concept of compositionally introduced test items, some notation needs to be introduced. We consider an assembly $\mathcal{A}$ consisting of components $C_1, \ldots, C_\alpha$, composed in accordance with some component model (i.e., α denotes the number of components in the assembly). To simplify the presentation, we denote by C_0 all the extra code of the assembly. In all other aspects, C_0 is not to be considered as a component. Moreover, given a particular test criterion TC, the test items of C_i and $\mathcal{A}$ are denoted $TI_{C_i}^{TC}$ and $TI_{\mathcal{A}}^{TC}$, respectively.

Definition 1 *The set of* compositionally introduced test items *of a test criterion TC and an assembly $\mathcal{A}$, is defined as follows:*

$$CI_{\mathcal{A}}^{TC} = TI_{\mathcal{A}}^{TC} \setminus \bigcup_{i=1}^{\alpha} TI_{C_i}^{TC}$$

Thus, $CI_{\mathcal{A}}^{TC}$ denote the test items of $\mathcal{A}$ that are not present when the constituent components $C_1, \ldots, C_\alpha$ are considered in isolation. Since most structural test criteria are defined in terms of control flow paths or control flow graphs, these concepts must be defined on an assembly level. For this, we denote by S_i the statements of component C_i.

Definition 2 *The statements of an assembly $\mathcal{A}$ are denoted $S_{\mathcal{A}}$, and defined as*

$$S_{\mathcal{A}} = \bigcup_{i=0}^{\alpha} S_i.$$

Definition 3 *The* control flow graph *of an assembly $\mathcal{A}$ is a directed graph where the nodes are the statements in $S_{\mathcal{A}}$, and a directed edge $\langle s_1, s_2 \rangle$ represents a possible control flow from statement s_1 to s_2.*

Definition 4 *A* control flow path *of an assembly $\mathcal{A}$ is a finite path in the control flow graph of $\mathcal{A}$.*

The control flow graph of an assembly can be very complex, particularly for component models where transfer of control from one component to another is not related to explicit constructs in the component code (as, for example, in **CM3** where the scheduler may preempt a component at any point). Further, we note that each edge $\langle s_k, s_{k+1} \rangle$ in the control flow path of an assembly is either

1. part of the local control flow of a component C_i (i.e., $\langle s_k, s_{k+1} \rangle$ is in the control flow path of C_i and $1 \leq i \leq \alpha$);
2. part of the control flow of the additional code C_0; or
3. a transfer of control between two components, or between a component and additional code (i.e., $s_k \in S_i$ and $s_{k+1} \in S_j$, with $i \neq j$, $0 \leq i \leq \alpha$ and $0 \leq j \leq \alpha$).

Categories 2 and 3 are of particular interest, since they are the ones introduced as a result of the composition. The third category is the main source of complexity, and this is also where the choice of component model has the biggest impact.

For the usage proposed here, i.e., to measure test coverage and guide test case generation, it is preferrable if the transfer of control between components can be determined, or approximated, from the component interfaces. The impact of the component model on the control flow graph is further discussed in Section 4.2.

4 Test Criteria

This section lists a number of structural test criteria, and, for each criterion, defines its set of compositionally introduced test items. In the cases where the choice of component model (**CM1** – **CM3**) affects this set, this effect is described for each different choice. The structural test criteria we investigate in this section with respect to the set of compositionally introduced test items are:

- **Statement coverage** is chosen since it is the most basic, and in our experience, the most widely recognized structural test criterion, even to the point that it is sometimes used synonymously with *code coverage* or *coverage* in general.
- **Branch coverage** is chosen since it has a large similarity to statement coverage, but still differs with respect to compositionally introduced test items.
- **Path coverage** is chosen based on the fact that it, in its basic form, requires the execution of each path through the system. As such, it is one of the more exhaustive test criteria available.
- **Modified condition/decision coverage (MC/DC)** is chosen since it is required as a part of the de-facto standard process in software development of some safety critical software, e.g., avionics software [25].
- **All uses coverage** is chosen since it, when considering shared variables in multi-tasking environments (a typical integration or system-level testing concern), targets failures related to race conditions and similar interleaving problems [11, 26].

In our work, we make use of definitions of these criteria from [1, 27] in defining the compositionally introduced test items for each of the three representative component models.

4.1 Statement Coverage Criterion

According to Zhu et al. [1], "*a set P of execution paths satisfies the statement coverage criterion if and only if for all nodes n in the flow graph, there is at least one path $p \in P$ such that node n is on the path p*".

For **CM1**, **CM2**, and **CM3**, the set of compositionally introduced test items of the statement coverage criterion are just the statements of the additional code, i.e., $CI_{\mathcal{A}}^{\mathrm{SC}} = S_0$. This follows directly from Definitions 1, 2 and 4, since for each statement $s \in S_{\mathcal{A}}$ in the control flow graph of $\mathcal{A}$ we have $s \in S_i$, $0 \leq i \leq \alpha$. Thus, either $s \in S_0$ or s is among the test items of component C_i, in which case it should not be included in $CI_{\mathcal{A}}^{\mathrm{SC}}$.

4.2 Branch Coverage Criterion

Again, according to Zhu et al. [1], "*a set P of execution paths satisfies the branch coverage criterion if and only if for all edges e in the flow graph, there is at least one path $p \in P$ such that p contains the edge e*".

As discussed in Section 3.2, there are three categories of edges in the assembly control flow graph: (1) the control flow within the components, (2) the control flow within the additional code, and (3) the transfer of control between two components, or between a component and additional code. Edges from the first category are not included in the set of compositionally introduced test items, which thus can be described as $CI_{\mathcal{A}}^{\mathrm{BC}} = B_2 \cup B_3$, where B_2 and B_3 correspond to categories 2 and 3 above, respectively.

Regarding B_3, for **CM1**, these edges go from an exit statement of one component to the entry statement of another. Alternatively, if communication is carried out by glue code, from exit statements to some $s \in S_0$ and from some $s \in S_0$ to the entry statement of a component.

For **CM2**, B_3 consists of edges going from a method call statement in one component to an entry statement in the called component, and from the return statement of a method to the next statement of a caller, possibly linked by additional code statements.

For **CM3**, let C_i and C_i be two components, such that C_j has strictly higher priority than C_i. Then B_3 contains edges from all statements in S_i to the first statement of C_j, and from each final statement in S_j to all statements in S_i. Note that if the assembly constitutes the entire system, then additional information is available, such as periods, response times, etc. of all components in the system. This additional system-level information can be exploited to further reduce the number of edges in B_3.

4.3 Path Coverage Criterion

"*A set P of execution paths satisfies the path coverage criterion if and only if P contains all execution paths from the begin node to the end node in the flow graph*" [1].

For this criterion, the compositionally introduced test items are those paths in the control flow graph of $\mathcal{A}$ which includes a statement from S_0, or two statements $s_k \in S_i$ and $s_l \in S_j$, such that $i \neq j$.

For **CM1**, the paths in $CI_{\mathcal{A}}^{\mathrm{PC}}$ are sequential combinations of local control flow paths of the components, respecting the order in which they are connected in the pipes and filter scheme. For **CM2**, $CI_{\mathcal{A}}^{\mathrm{PC}}$ is the set of interleaved control flow paths, where the points of interleaving are constituted by the calls and returns of methods between components. For **CM3**, $CI_{\mathcal{A}}^{\mathrm{PC}}$ consists of interleaved control flow paths, where the points of interleaving are governed by component priority levels, similarly to the branch coverage criterion discussed above.

4.4 MC/DC Criterion

According to Chilenski and Miller [27], the Modified Condition/Decision Coverage (MC/DC) criterion is satisfied when "*every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions*".

For **CM1**, **CM2**, and **CM3**, the set of compositionally introduced test items is the set of test items introduced by the additional code. To show that these are the only test items introduced by the integration of components, we establish that no new points of entry and exit are introduced by composition, and that

the only conditions in the resulting assembly are the conditions residing in the assembled components.

4.5 All Uses Coverage Criterion

"*A set P of execution paths satisfies the all-uses criterion if and only if for all definition occurrences of a variable x and all use occurrences of x that the definition feasibly reaches, there is at least one path p in P such that p includes a subpath through which that definition reaches the use*" [1].

The compositionally introduced test items of the all uses criterion is given by $CI_{\mathcal{A}}^{\mathrm{US}} = D_1 \cup D_2$, where D_1 is the set of pairs of definiton and uses that fulfil the criterion in the control flow graph of the additional code; D_2 is the set of pairs of definitions and uses that fulfil the criterion, and where the definition of the variable is performed in a component C_i and the use of the variable is performed in a component C_j, such that $i \neq j$.

In **CM1**, all component communication is supposed to take place via the explicit component ports. If this can be ensured, e.g., by the development framework, D_2 is empty. For **CM2**, it is also the case that D_2 is empty if the underlying formalism does not permit shared variables. Since **CM3** allows shared variables, D_2 is simply the set of feasible inter-component shared variable definition and use pairs.

5 Discussion

Our research so far has been aimed at developing a general formal theory for identifying additional structural test efforts required under different models of component interactions. The consideration of three architectural styles and five test criteria yields fifteen possible architectural style/test criterion combinations, of which not all are reasonably applicable. Below, we reflect upon the most notable combinations.

The fact that statement coverage composes nicely might be no major surprise, since it is intuitive that no new statements are introduced during integration (besides the ones in the extra code). Branch coverage, however, is more interesting during integration testing, since the set of all compositionally introduced test items for branch coverage describes all transfers of control from one component in the assembly to another. For **CM3**, covering these transfers of control quickly becomes impracticable without rigid restrictions on the scheduling of components, but for **CM1** and **CM2**, covering these branches would be a feasible way of testing explicit component interactions.

Path coverage suffers severely from complexity issues even at the component level [1], and would at best be applicable for very small systems conforming to **CM1** and **CM2**, with a handful of branching statements. Furthermore, the fact that MC/DC coverage scales well for all architectural styles might be interesting to component-based software developers building systems that should conform to a standard that requires such coverage (e.g., [25]). Finally, as related work

shows [11, 26], data flow (e.g., all uses) coverage on integration level is useful for detecting interleaving failures such as race conditions and stale-value errors in systems conforming to **CM3**.

6 Related Work

Previous contributions in testing of component-based systems range from verification of execution time properties by evolutionary testing [28], through regression testing of components based on information provided regarding component changes [29], to model-based testing of component-based systems [30]. Despite this variety of contributions in this field, there exists, to our knowledge, no previous work discussing traditional structural test coverage in the integration testing of component-based systems. However, the fact that components need to be tested in the integrated setting in which they are intended to operate is recognized, e.g., by Weyker [31]. It is our firm belief that the quality of the verification would benefit by complementing the functional testing traditionally performed during integration with structural coverage as described in this paper.

Outside the component-based development community, the most notable efforts regarding structural testing on integration- or system level has been investigations of how to achieve structural coverage in concurrent systems of different flavours (typically focusing on definitions and uses of shared variables [11, 26]). For structural testing of concurrent systems, many approaches combine the internal control flow structure of concurrent threads with the possible synchronizations between the threads. By doing this, a system-level control flow representation for structural testing can be achieved. Also related to this work, are contributions describing specialized structural test criteria focusing on the control flow paths of concurrent programs [32–35]. In contrast to the above works, the contribution of this paper is an effort to generalize the problem by considering several architectural styles, and a variety of test criteria.

7 Conclusions and Future Work

Building a system out of well-tested components does not necessarily result in a well-tested system. Generally, after integration of well-tested components into a subsystem or a system, the interaction between components remain to be tested. Using functional testing, we will cover the functional aspects of the integration (if the specification used as the base for test case generation is of a sufficient quality), but in order to cover structural and non-functional aspects, structural testing is required. Furthermore, structural coverage measures are not perfect, but they constitute the main formal quality assurance measures available in software testing.

In this paper, we have described the added test effort required by component integration, by introducing the concept of *compositionally introduced test items*. Also, we have described the impact of this concept for a number of common structural test criteria considering common architectural styles. Second, we have

shown what is required in order to achieve structural test coverage at an arbitrary level of integration in multi-level hierarchical development of component-based software. By doing this, we have facilitated a less ad-hoc way of determining the adequacy of integration- and system-level testing than the functional testing traditionally used at this level.

Extending this approach to other component models and identifying impacts of relaxing some of our assumptions will be the immediate followups of this work. Several interesting questions also need to addressed such as a) what information we need to provide at the component interface level and b) what happens if we do not have access to source code. We have also assumed strong adherence to component model semantics at lower levels of implementation, which cannot be taken for granted in many systems where the underlying implementation could be based on languages such as C. Scenarios that are potentially capable of invalidating the results need to identified and appropriate additional test efforts need to be incorporated.

Furthermore, in the continuation of this work, a goal would be to, for different architectural styles, and different test criteria, find a set test cases that safely covers the set of compositionally introduced test items. A second goal would be to find the minimal set of test cases that fulfils this criterion.

References

1. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys (CSUR) **29**(4) (1997) 366–427
2. Juristo, N., Moreno, A.M., Vegas, S.: Reviewing 25 Years of Testing Technique Experiments. Journal of Empirical Software Engineering **9**(1-2) (2004) 7–44
3. Frankl, P.G., Weyuker, E.J.: An Applicable Family of Data Flow Testing Criteria. IEEE Transactions of Software Engineering **14**(10) (1988) 1483–1498
4. Pavlopoulou, C., Young, M.: Residual test coverage monitoring. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 277–284
5. Craig, R.D., Jaskiel, S.P.: Systematic Software Testing. Artech House Publishers (2002)
6. van Veenendaal, E.: The Testing Practitioner. Uitgeverij Tutein Nolthenius (2002)
7. Copeland, L.: A Practitioner's Guide to Software Test Design. STQE Publishing (2003)
8. Crnkovic, I., Larsson, M.: Building Reliable Component-Based Software Systems. Artech House Publishers (2002)
9. Lau, K.K., Wang, Z.: A Survey of Software Component Models (May 2006) Second edition, Pre-print CSPP-38, School of Computer Science, The University of Manchester.
10. Shaw, M., Garland, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall (1996)
11. Sundmark, D., Pettersson, A., Sandberg, C., Ermedahl, A., Thane, H.: Finding DU-Paths for Testing of Multi-Tasking Real-Time Systems using WCET Analysis. In: Proceedings of the 7thInternational Workshop on Worst-Case Execution Time Analysis, (WCET'2007). (July 2007)

12. Pretschner, A.: Compositional Generation of MC/DC Integration Test Suites. Electronic Notes in Theoretical Computer Science **82**(6) (2003)
13. Goldberg, A., Wang, T.C., Zimmerman, D.: Applications of Feasible Path Analysis to Program Testing. In: ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international Symposium on Software Testing and Analysis, New York, NY, USA, ACM Press (1994) 80–94
14. Gustafsson, J., Ermedahl, A., Lisper, B.: Algorithms for Infeasible Path Calculation. In: Sixth International Workshop on Worst-Case Execution Time Analysis, (WCET'2006), Dresden, Germany (July 2006)
15. Hayes, I., Fidge, C., Lermer, K.: Semantic Characterisation of Dead Control-Flow Paths. IEE Proceedings - Software **148**(6) (December 2001) 175 – 186
16. Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M.: The SAVE approach to component-based development of vehicular systems. Journal of Systems and Software **80**(5) (May 2007) 655–667
17. Nierstrasz, O., Arévalo, G., Ducasse, S., Wuyts, R., Black, A.P., Müller, P.O., Zeidler, C., Genssler, T., van den Born, R.: A component model for field devices. In: Proc. of the 1st Int. IFIP/ACM Working Conference on Component Deployment, Springer (2002) 200–209
18. Sun Microsystems: JavaBeans Specification 1.01 (August 1997) Available from http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html.
19. Box, D.: Essential COM. Addison-Wesley (1997)
20. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala Component Model for Consumer Electronics Software. IEEE Computer **33**(3) (2000) 78–85
21. Lundbäck, K.L., Lundbäck, J., Lindberg, M.: Component Based Development of Dependable Real-Time Applications. Technical report, Arcticus Systems, http://www.arcticus.se
22. Sandström, K., Fredriksson, J., Åkerholm, M.: Introducing a component technology for safety critical embedded realtime systems. In: International Symposium on Component-based Software Engineering (CBSE7), Edinburgh, Scotland, Springer Verlag (May 2004)
23. Ke, X., Sierszecki, K., Angelov, C.: COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems. In: Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE (2007) 199–208
24. Bureš, T., Carlson, J., Crnković, I., Sentilles, S., Vulgarakis, A.: ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University (June 2008)
25. RTCA: Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B. RTCA (December 1992)
26. Yang, C.S.D., Pollock, L.L.: All-uses Testing of Shared Memory Parallel Programs. Software Testing, Verification and Reliability **13**(1) (2003) 3 – 24
27. Chilenski, J.J., Miller, S.P.: Applicability of Modified Condition/Decision Coverage to Software Testing. Software Engineering Journal (September 1994) 193 – 200
28. Groß, H.G., Mayer, N.: Evolutionary testing in component-based real-time system construction. In: GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2002) 1393
29. Mao, C., Lu, Y.: Regression testing for component-based software systems by enhancing change information. In: APSEC '05: Proceedings of the 12th Asia-

Pacific Software Engineering Conference, Washington, DC, USA, IEEE Computer Society (2005) 611–618
30. Pelliccione, P., Muccini, H., Bucchiarone, A., Facchini, F.: TeStor: Deriving Test Sequences from Model-based Specifications. In: Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005), Lecture Notes in Computer Science, LNCS 3489. (2005) 267 – 282
31. Weyuker, E.J.: Testing Component-Based Software: A Cautionary Tale. IEEE Softw. **15**(5) (1998) 54–59
32. Katayama, T., Itoh, E., Ushijima, K., Furukawa, Z.: Test-Case Generation for Concurrent Programs with the Testing Criteria Using Interaction Sequences. Proceedings of sixth Asia-Pacific Software Engineering Conference (APSEC'99) **00** (1999) 590
33. Taylor, R.N., Levine, D.L., Kelly, C.D.: Structural Testing of Concurrent Programs. IEEE Transactions on Software Engineering **18**(3) (1992) 206–215
34. Wong, W.E., Lei, Y., Ma, X.: Effective Generation of Test Sequences for Structural Testing of Concurrent Programs. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), Washington, DC, USA, IEEE Computer Society (2005) 539–548
35. Yang, R.D., Chung, C.G.: Path Analysis Testing of Concurrent Program. In: Information and Software Technology. Volume 34(1). (Jan 1992) 43–56