Component Integration Testing by Graph Transformations

Reiko Heckel and *Leonardo Mariani*
Department of Mathematics and Computer Science
University of Paderborn
33095 Paderborn, Germany
{reiko, mariani}@upb.de

Abstract Component-based technology can increase reuse and productivity, but high-quality component-based systems are often difficult to implement. Component developers do not know the systems where the components will be used, while software engineers must develop new systems with limited knowledge on available components.

We propose a new testing technique that generates, at the time of component development, integration test cases from the specification of the behavior expected from other components of the system. The specification is provided by executable graph transformation rules that are visualized by a UML-based notation. Test cases are executed at an early stage to validate the integration of the component with the expected behavior of the system, and then are re-executed with concrete components at deployment time.

The technique presented in this paper supports both the component developer, who can early test the integration of the components with the system, and the software engineers, who can test concrete components at deployment-time, simply re-using existing test cases.

Keywords: Testing of Component-Based Systems, Integration Testing, Testing by Graph Transformations, Data-Flow Analysis, Model-based Testing.

1 Introduction

Component-based technology supports development of complex systems by composition of components [8]. Integration of components can facilitate development, but provides new challenges for testing. Interactions between components can hide faults that are difficult to identify and repair, for instance components that behave correctly in isolation can be incompatible when integrated.

Software engineers validate the quality of a set of components by *integration testing*. The possible lack of knowledge about the behavior of single components and their requirements towards other components introduces sever difficulties. In fact, components' expectations are often only partially provided in the documentation, source code is seldom available, and the specification of the components' behavior is often incomplete. Therefore, engineers cannot take into account all possible behaviors when designing test cases.

In this paper, we propose a new testing technique that uses graph transformations (GT) [16] for deriving early and effective integration test cases during component development. The component developer specifies the expected high-level behavior of the components used by the component under development (CUD) with executable GT rules from which test cases are derived. Components can be then deployed with test cases that can be executed by software engineers without requiring specific knowledge about components.

Executable GTs can be derived at design time from UML diagrams with little effort [2]. The existence of the GTs can be exploited in the following ways:

1. the component developer can use the executable GTs to simulate external components during unit testing
2. the component developer can test if the CUD correctly uses third-part services. In this case, testing is performed against the *specified* expected behavior

3. test cases that have been derived in 2 can be delivered with the concrete component, so that engineers can execute them at deployment time

4. if the external components are provided with a model of their behavior, the expected behavior and the provided behavior can be matched to detect incompatibilities

Points 1 and 4 have been already addressed in [17] and [12], respectively. In this paper, we focus on 2 and 3 by discussing techniques for deriving integration test cases.

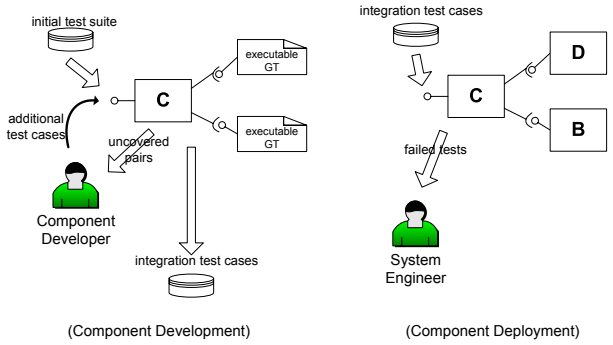

Figure 1: The general scenario.

The application of the technique proposed in this paper on a component `C` is shown in Figure 1. An initial test suite is generated and executed. The behavior of the components that are not available is simulated by executing the corresponding GTs. If an execution generates a previously unobserved pair of dependent requests from `C` to other components, i.e., the effect of the second request depends from the effect of the first request, the execution is selected as integration test case. The whole set of pairs of dependent calls that can be automatically generated by `C` can be computed by accessing both the GTs and the source code of `C`. Therefore, once the test cases in the initial test suite have been executed, the component developer inspects the set of uncovered pairs and generates new test cases for increasing coverage. Test cases are generated until either the component developer is satisfied of the coverage or all pairs have been covered. Results of computations can be stored with test cases.

Note that integration test cases are executed early by the component developer against the executable GTs. If the target component passes all test cases, the component developer can be confident about the correct integration of the CUD with the behavior expected from the used components.

The integration test cases are then deployed together with the component, so that the system engineer can both execute them and use observed results as oracle, i.e., a test case is passed if the produced result matches the observed one. The system engineer can examine failed tests to discover if they either correspond to a valid behavior or a fault.

The paper is organized as follow. Section 2 introduces both the use of GT for the specification of the expected behavior and the definition of possible relations among rules. Section 3 presents the coverage criterion that is then used in Section 4 for generating the concrete test cases. Section 5 provides early evaluation of the technique and discusses related work. Finally, Section 6 summarizes our contribution.

2 Specification of the Expected Behavior by GT

A component provides a coherent set of services by relying on the existence of other components that can be accessed at run-time. For instance, a cart component that provides a purchase service can depend on the existence of a catalog component to retrieve details of each single product. When the cart component is developed, it is likely that the catalog component is not available. However, the cart component can be implemented only by assuming that the catalog component will satisfy some kind of contract. In our approach, the expected behavior is specified at the conceptual level by executable GT rules.

In particular, the GT rules describe how the state of the accessed component is modified when a service is executed. The state of a component at a given time is represented by an attributed graph [16] that can be visualized with a UML object diagram. Objects in the diagram represent conceptual entities that can be very different from concrete objects instantiated in the concrete components.

A GT rule has a precondition and an effect, both of

them specified by a graph. If a copy of the precondition graph exists as part of the current configuration, the rule can be applied. The application of the rule consists of replacing the subgraph that matches the precondition with the graph specified in the effect. More than one rule could be enabled by the same configuration. The assumption is that only one rule (non-deterministically) takes place when a service is executed.

For instance, the expected behavior for a `removeProduct` service when the parameter is a valid reference to an object in a category can be represented by the production rule in Figure 2 (a). Figure 2 (b) shows the application of the rule to an example conceptual state of the Catalog component.

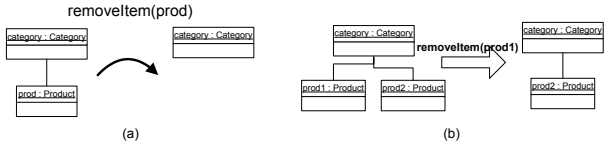

Figure 2: (a) the example of a production rule for the `removeItem` service, and (b) an example of application of the rule on a possible conceptual state.

The effect of one GT rule can interact with another GT rule leading to both conflicts and dependencies. We expect that interactions including both conflicts and dependencies are more likely to reveal faults than simple calls. Therefore, our integration test cases consist of stimuli which generate such interactions. We now informally introduce both dependency and conflicts among GT rules. Precise definitions can be found in [11].

Let G and H be attributed graphs and p a GT rule, a GT from G to H obtained by the application of p to G is denoted with $G \stackrel{p}{\Rightarrow} H$. We say that a GT $G \stackrel{p_1}{\Rightarrow} H_1$ is *parallel independent* of $G \stackrel{p_2}{\Rightarrow} H_2$, if the second transformation does not invalidate the applicability of the first, i.e., the second rule does not delete, create or modify anything that is necessary for the application of the first rule.

A GT $G \stackrel{p_1}{\Rightarrow} H_1$ is *sequential independent* of $H_1 \stackrel{p_2}{\Rightarrow} X$, if the second transformation can be carried out independently from the occurrence of the first transformation, i.e., the second rule does not require the existence or forbid anything that is created or deleted by the first rule.

Finally, we say that

- p_1 *may be disabled* by p_2 if there exist transformation steps $G \stackrel{p_1}{\Rightarrow} H_1$ and $G \stackrel{p_2}{\Rightarrow} H_2$ such that $G \stackrel{p_1}{\Rightarrow} H_1$ is not independent of $G \stackrel{p_2}{\Rightarrow} H_2$
- p_1 *may cause* p_2 if there exist transformation steps $G \stackrel{p_1}{\Rightarrow} H_1 \stackrel{p_2}{\Rightarrow} X$ such that $H_1 \stackrel{p_2}{\Rightarrow} X$ is not independent of $G \stackrel{p_1}{\Rightarrow} H_1$

3 Derivation of Possible Test Cases

An integration test case for a component `C` consists of a sequence of invocations on `C` that generates calls to other components of the system. Since it is not feasible to cover all possible sequences of calls, we select relevant sequences by exploiting the criteria of dependency between rules that have been defined in the previous section. The underlying assumption is that sequences of calls that modify and use the same part of the component's state are more likely to expose an integration fault, i.e., an assumption performed by the CUD that is not verified by the accessed component.

A similar idea has been already successfully exploited in data-flow testing where combination of definitions and uses of the same state variables are tested [15, 9]. We propose an original application of this idea for integration testing of component-based system. In our approach, the role of the state variables is played by the conceptual states of the used components and the role of instructions, which define and use variables, is played by service invocations, which modify and use the state graph.

In classical data-flow testing, a well-known and successfully applied coverage criterion is "all def-use pairs" [10, 14]. We adapt this criterion to integration testing of component-based systems by requesting the coverage of both all dependent pairs of service calls and all potential pairs of conflicting calls. A pair of services is dependent if the first service may cause the second service, while a pair of service is conflicting if the first service may disable the second service. Test cases are generated in three steps:

1. we identify all dependent and conflicting pairs of rules in the set of services accessed by the CUD

2. we then generate the Protocol-Based Inter-Procedural Control-Flow Graph (PBICFG) of the CUD and we map the set of pairs identified in the previous step to concrete pairs of calls in the PBICFG

3. finally, during unit testing, we monitor executions of the CUD and we record all stimuli that correspond to executions that increase coverage. The whole set of recorded stimuli is the set of integration test cases.

In the following, we present how we address the first two points, while next Section addresses the generation of the concrete test cases.

Computation of Dependent Pairs Dependent calls can be automatically identified by considering all possible pairs and checking if any relation holds between calls [17].

The computation of the dependent pairs includes a pre-processing operation which output is used in the next step. The pre-processing consists in associating to each rule the set n_m of nodes, n_e of edges and n_a of attributes that are modified by the rule. Moreover, each pair is associated with the set of entities n_r that is modified in the first rule and then used in the second.

Computation of Pairs of Nodes Once the dependent pairs of calls have been computed, we must identify the concrete instructions that can generate them. This step requires the availability of the source code. We generate an Interprocedural Control-Flow Graph (ICFG) for each CUD's service. Only instructions that either generate calls to services provided by other components or modify the flow of control, are represented in the ICFG.

We then generate a PBICFG, which is obtained by combining the derived ICFGs with the specification of the CUD's protocol. The component developer provides the specification of the protocol by a Finite State Automaton (FSA) that generates the possible invocation sequences. If the FSA is not available, we assume that there are no constraints on the possible order of invocation (that is equivalent to providing a FSA that generates any unbound sequence of invocations). The FSA specifies service invocations by node labels. The PBICFG is obtained by replacing the nodes of the FSA that represent calls to CUDs' services with the corresponding ICFG. The ICFG contains the name of the executed service as entry and exit nodes. Invocations to services not implemented in the CUD are not unfolded, but are represented by singles nodes labeled with the name of the executed service.

The PBICFG contains two kinds of edges: dotted edges and solid edges. Dotted edges correspond to edges in the initial FSA, while solid edges represent edges of the ICFG. This distinction is useful for interpreting paths in the PBICFG. An entry node reached by a solid edge corresponds to a service invoked from another service, while an entry node reached by a dotted line corresponds to the user that requires the service. A test case can be derived from a path in the PBICFG by considering the sequence of entry nodes that are reached through dotted lines. An example of a PBICFG is provided in Figure 3.

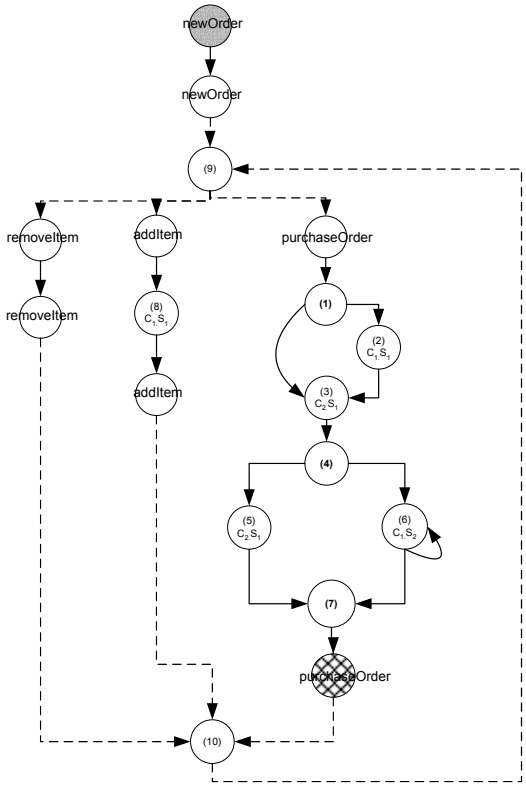

Figure 3: An example of a PBICFG. The gray node is the initial node, while the node with the matrix is the final node.

The mapping between a pair of dependent invocations (s_1, s_2) and the concrete nodes that generate them is performed by identifying all pairs of nodes $p = (n_i, n_f)$ of the PBICFG so that, n_i generates s_1, n_f generates s_2, and there exists a path from the initial node to the final node that traverses n_i and then n_f. Moreover, if $n_1, \ldots n_k$ are nodes traversed between n_i and n_f, the effect of these nodes must not completely invalidate the relation between n_i and n_f, i.e., it is not true that all these relations hold: $\cup_{i\in\{1,\ldots n\}} n_m(n_i) \supseteq nodes(n_r(p))$, $\cup_{i\in\{1,\ldots n\}} n_e(n_i) \supseteq edges(n_r(p))$, $\cup_{i\in\{1,\ldots n\}} n_a(n_i) \supseteq attributes(n_r(p))$. In practice, nodes traversed between a pair of dependent calls must not modify all entities in n_r, because otherwise the effect of the first rule would be completely overwritten by the intermediate calls.

The identified concrete pairs in the PBICFG can still be infeasible to generate. In fact, given a path, there is no guarantee that there exists an input covering it. However, the component developer can use the PBICFG (1) to exclude the existence of further pairs that must be considered, (2) to decide which pairs can be skipped because less important, and (3) to inspect the code for discarding infeasible pairs.

Let us consider the example in Figure 3. If $C1.S1$ *may cause* both $C1.S1$ and $C2.S1$, $C2.S1$ *may cause* $C1.S2$, $C1.S2$ *may cause* $C1.S2$, and $C1.S1$ *may disable* $C1.S2$; and $C2.S1$ modifies state nodes so that the *may disable* relation between $C1.S1$ and $C1.S2$ is overwritten, we have that the following set of node pairs that should be covered: $(2,3)$ $(2,5)$ $(3,6)$ $(6,6)$ $(8,3)$ $(8,5)$.

We remark that the PBICFG and the set of concrete nodes that should be covered can be automatically derived from the protocol specification, the GT specification and the source code.

There are several GT rules that can be executed when a call to an external service is issued. Requiring coverage of pairs of call nodes in the PBICFG is not enough for covering pairs of dependent calls. Therefore, we consider also the identifier of the GT rule that is expected to be executed as part of the coverage criterion. For instance, if the may cause relation between nodes $(2,3)$ depends on the execution of GT rules p_2 and p_3 respectively, the requested coverage is $(2\ p_2, 3\ p_3)$. In particular, all possible dependent pairs of production rules are considered for each pair of nodes that must be covered. A weaker relation can be obtained by considering only one pair of dependent calls when multiple dependent calls exist.

4 Derivation of Concrete Test Cases

In the previous Section, we explained how to identify pairs of nodes which generate call sequences that must be tested. The specification of the pairs that must be covered is augmented with the rules that should be activated on the server component. We now present how to generate concrete test cases.

The component developer initially generates functional and/or structural test suites that are used for unit testing of the CUD. Unit testing is facilitated by the existence of the GT rules that can be executed to simulate the behavior of external components once their initial states have been defined. The CUD is then instrumented to monitor nodes of the PBICFG that are traversed during executions, while stubs trace the executed GT rules. Integration test cases are generated by recording inputs that trigger executions increasing pair coverage. The test case is given by both the initial states of components involved in the execution and a sequence of invocations on the CUD.

If a service has a non-deterministic behavior, the execution of a test case can generate interactions that differ from those observed when the test case has been recorded. In our application scenario, we assume that the service behavior is described enough precisely to avoid non-determinism. Eventually, the system engineer can decide to re-execute multiple times the same test case to generate the different non-deterministic behaviors.

At the end of unit testing, all uncovered pairs are signaled to the component developer. If both functional and structural coverage have been achieved, we expect that little effort is required for both discarding unfeasible pairs and generating further test cases that cover the missing pairs. In fact, it is likely that most of missing pairs are related to insidious combination of GT rules, which are activated for particular states when combined with particular execution paths in the CUD.

The derived test cases are deployed together with the corresponding component. Each test case has a precondition on the states of the used components, which consists of the specification of the expected conceptual initial state. The software engineers initialize the components

of the system to enable consistent execution of test cases. The initial state of the components can be set by preliminarily executing components' services until a given state is reached or by exploiting components' built-in facilities to directly set the state [4, 19].

The outputs generated from components during testing can be stored together with test cases, so that the software engineers can easily generate oracles matching the observed results with the stored ones. Only outputs that differ from values recorded during unit testing need to be inspected by software engineers.

5 Related Work and Preliminary Evaluation

Testing of interactions among software components has been already addressed in testing of software protocols, see [5] for a survey on this topic. Existing approaches are generally based on the existence of a specification, e.g., a FSA, and a coverage criterion for generating test cases, e.g., all edges. Several models with different degree of expressivity exist, however they present some important differences with respect to our approach. The specification of components' behavior is often required for all components of the system involved in an interaction. Instead, the integration of a component with the system can be tested by our technique without requiring any specification for the other components. GT rules are very effective in both describing the behavior of object-oriented components that exchange complex parameters and taking into account the expected evolution of the internal states of components in the system. Models from protocol testing are less suitable to address both object-oriented software and the expected evolution of components' internal states than GT behavioral models. However, protocol testing can be used to complement the set of test cases generated by our technique with coverage of possible sequences of calls.

The use of GT for test case generation is also an original contribution of this paper. To our knowledge, only the paper by Baldan, König and Stürmer [1] addressed the same issue for "code generators".

Currently, we have partial tool support. AGG [17] provides analysis for identification of both dependent and conflicting pairs, but is not suitable for dealing with object attributes. Engines that execute GTs for simulating real components are available at [17, 18].

Early experiences with our technique have showed that the generated test cases are suitable for revealing mismatches between the expected and the provided behavior. Incompatibilities due to interactions taking place when external components are in specific states are likely to be revealed. These kinds of incompatibilities are often difficult to be revealed with other testing techniques.

We remark that our technique can be applied as far as the component developer can inspect the CUD for generating further integration test cases to reach a satisfying degree of coverage. Symbolic execution [7] and automated deduction [6] can aid the component developer in refinement of test cases.

Integration test cases can be automatically derived with the technique by Mariani, Pezzè and Willmor [13]. They automatically infer a model of component interactions and they automatically generate the corresponding integration test cases. Test cases cover aspects related to both the protocol and data values used for interactions, but neglect coverage of the assumptions that single components perform on state evolution of the other components of the system.

Other approaches already proposed to deploy test cases together with components [4, 3, 19]. However, they are mainly concerned with the problem of providing a framework for automatically executing integration test cases, rather than being concerned with the problem of deriving integration test cases. These frameworks can be exploited by our technique.

6 Conclusions and Future Work

Integration testing of component-based systems is difficult because engineers must often design test cases despite limited knowledge on reused components.

In this paper, we propose a technique that enables early design of test cases at the developer's site. Thus, component developers can both produce integration test cases and preliminarily test the component under development against the specification of the behavior expected from other components of the system. Test cases are packed

with components and software engineers can re-execute them to effectively test the integration of components without necessarily knowing details about components' behavior. Results obtained during unit testing can be reused for generating oracles.

Future work concerns with the extension of the technique with the generation of test cases that cover variables defined in the CUD and then used on other components, and vice versa. To this end, classical data-flow analysis must be integrated with GT rules to compute pairs of definitions and uses that span the code and the GT specifications. We think that this integration can simplify test case generation and can improve the effectiveness of test cases.

References

[1] P. Baldan, B. König, and I. Stürmer. Generating test cases for code generators by unfolding graph transformation systems. In *proceedings of 2nd Internation Conference on Graph Transformation*, Rome, Italy, 2004.

[2] L. Baresi and R. Heckel. Tutorial introduction to graph transformation: a software engineering perspective. In *proceedings of the Internation Conference on Graph Transformation*, volume 1 of *LNCS*. Springer, 2002.

[3] A. Bertolino and A. Polini. A framework for component deployment testing. In *proceedings of the 25th International Conference on Software Engineering*, pages 221–231, Portland, Oregon, 2003. IEEE Computer Society.

[4] R. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, 1994.

[5] G. V. Bochmann and A. Petrenko. Protocol testing: review of methods and relevance for software testing. In *proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 109–124, Seattle, Washington, United States, 1994. ACM Press.

[6] A. Bundy. A survey of automated deduction. In M. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today*, volume 1600 of *LNAI*, pages 153–174. Springer-Verlag, 1999.

[7] L. Clarke. A system to generate test data and symbolic execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[8] I. Crnkovic and M. Larsson. *Building Relyable Component-Based Software Systems*. Artech House, 2002.

[9] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[10] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, 1991.

[11] J. Hausmann, R. Heckel, and G. Taentzer. Detecting conflicting functional requirements in a use case driven approach: A static analysis technique based on graph transformation. In *proceedings of the International Conference on Software Engineering*, pages 105–155. ACM/IEEE Computer Society, 2002.

[12] J. H. Hausmann, R. Heckel, and M. Lohmann. Model-based discovery of web services. In *proceedings of the International Conference on Web Services*, 2004.

[13] L. Mariani, M. Pezzè, and D. Willmor. Generation of integration tests for self-testing components. In *proceedings of the 1st Internationl Workshop on Integration of Testing Methodologies*, LNCS. Springer, 2004.

[14] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[15] S. Rapps and E. Wejuker. Data flow analysis techniques for program test data selection. In *proceedings of the 6th International Conference on Software Engineering*, pages 272–278, 1982.

[16] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1 - Foundations. World Scientific, 1997.

[17] Technical University Berlin. The attributed graph grammar system (AGG). http://tfs.cs.tu-berlin.de/agg/.

[18] University of Paderborn. Fujaba. http://wwwcs.upb.de/cs/fujaba/.

[19] Y. Wang, G. King, and H. Wickburg. A method for built-in tests in component-based software maintenance. In *proceedings of the Third European Conference on Software Maintenance and Reengineering*, pages 186–189. IEEE Computer Society, 1999.