Integration Testing in Software Product Line Engineering: A Model-Based Technique*

Sacha Reis[1], Andreas Metzger[1], and Klaus Pohl[1,2]

[1] Software Systems Engineering, University of Duisburg-Essen,
Schützenbahn 70, 45117 Essen, Germany
{reis, metzger, pohl}@sse.uni-due.de
[2] Lero (The Irish Software Engineering Research Centre),
University of Limerick, Ireland
pohl@lero.ie

Abstract. The development process in software product line engineering is divided into domain engineering and application engineering. As a consequence of this division, tests should be performed in both processes. However, existing testing techniques for single systems cannot be applied during domain engineering, because of the variability in the domain artifacts. Existing software product line test techniques only cover unit and system tests. Our contribution is a model-based, automated integration test technique that can be applied during domain engineering. For generating integration test case scenarios, the technique abstracts from variability and assumes that placeholders are created for variability. The generated scenarios cover all interactions between the integrated components, which are specified in a test model. Additionally, the technique reduces the effort for creating placeholders by minimizing the number of placeholders needed to execute the integration test case scenarios. We have experimentally measured the performance of the technique and the potential reduction of placeholders.

1 Motivation

Software product line engineering (SPLE) is a proven approach for deriving a set of similar applications at low costs and at short time to market [8][21]. SPLE is based on the planned, systematic, and pro-active reuse of development artifacts (including requirements, components, and test cases). There are two key differences between SPLE and the development of single systems [21]:

1. The development process of a software product line (SPL) is divided into two sub processes: domain engineering and application engineering. In domain engineering, the commonalities and the variability of the SPL are defined and reusable artifacts, which comprise the SPL platform, are created. In application engineering, customer-specific applications are realized by binding the variability and reusing the domain artifacts.

---

2. The variability of an SPL is modeled explicitly by variation points and variants. Variation points describe what varies between the applications of an SPL, e.g. the payment method in an online store. Variants describe concrete instances of this variation, e.g. payment by credit card, debit card, or invoice.

Because of the division of the development process into domain and application engineering, there are two major kinds of development artifacts that have to be tested. In domain engineering, the SPL platform has to be tested, in application engineering the derived applications have to be tested.

The SPL platform contains variability. This variability prevents the use of existing testing techniques for single systems, because the domain artifacts do not define a single application but a set of applications. With existing techniques from single system testing, each of these applications would have to be individually tested in domain engineering, resulting in an enormous test effort.

In the literature, several approaches for SPL testing have been proposed (e.g., [4][10][15][17][20][24]). However, these approaches cover unit and system testing only.

This paper presents a model-based, automated technique for integration testing in domain engineering. Our technique generates integration test case scenarios (ITCSs), which support the test of the interactions between the components of an integrated sub-system. An ITCS describes the order and the type of interactions that have to be performed during test execution. By augmenting an ITCS with test data, integration test cases can be derived.

The basic idea of our approach is to create placeholders for necessary variable parts and all components that are not part of the integrated sub-system. These placeholders are considered during the model-based generation of ITCSs. The benefits of our technique are as follows:

Other failures than the ones uncovered in unit testing can be found, because the goal of integration testing is to uncover intercomponent failures (see e.g. [6]).

Testing can be performed earlier compared to system testing. For integration testing the complete system is not needed. Missing components or variants can be simulated. Such an early test can significantly reduce costs (see e.g. [7]).

Generally, not all possible interactions between components can be tested in integration testing. Our technique systematically selects a subset of all possible interactions by using a test model.

The ITCSs are derived in such a way that the development effort for necessary placeholders is minimized. Thus, the testing effort is reduced.

2   Related Work

In the literature, several approaches for scenario-based integration test case derivation are presented (see [2][12][16][26][29]). All these approaches only support integration testing for the development of single systems, and therefore none of these approaches considers variability. As the consideration of variability is essential when testing in domain engineering, these approaches are not suitable for the use in the context of

SPLs without substantial extensions. Still, they substantiate that a scenario-based approach is one suitable approach towards integration testing.

Many approaches for test case derivation in SPLE can be found in the literature [25]. Representatives are approaches by Bertolino and Gnesi [4][15], Geppert et al. [10], McGregor et al. [17][18], and Nebut et al. [20]. These approaches focus on unit and system testing only. In our previous work, we have developed the ScenTED technique for system testing [24][15] and performance testing [23]. McGregor focused on unit [18] and system testing [17], but furthermore he pointed out that all common artifacts can already be tested in domain engineering by integration tests. However, he did not present a concrete technique for that.

In an approach by Muccini and van der Hoek [19] challenges in testing product line architectures are presented. For integration testing, they suggest to integrate all common components first and then to integrate the variant components. However, they do not present solutions for integration test case derivation. Cohen et al. [9] state that applying well-understood coverage criteria when testing the applications of an SPL could lead to a low coverage of the SPL platform. They have developed specific coverage criteria to improve on this situation. The approach by Cohen et al. [9] and our previous work [24][15] both support an early test in domain engineering by creating sample applications. However, creating a sample application assumes that all relevant components and variants have been developed. Consequently, this approach will be typically applied very late in the domain engineering process.

General approaches for deriving test case scenarios from control flow graphs also exist (e.g., [22][5][27]). In the approach of Wang et al. [27], all possible scenarios through a test model are derived by employing existing algorithms. Then, a mathematical optimization is performed to achieve the minimal set of these test case scenarios for branch coverage. Other criteria than branch coverage are also discussed. Because of the possible use of other coverage and also optimization criteria, this is a quite general approach. Still, variability and specifics of integration testing are not explicitly discussed.

Summarizing, none of the existing approaches for SPL testing support integration testing in domain engineering.

3   Overview of the Technique

In the following subsections, the test models that are used as input to our technique are described and an overview of the activities of the technique is given.

3.1   Test Models

Test models, which are used as input to our technique, are specified by UML 2.0 activity diagrams. We use UML 2.0 activity diagrams to define the control flow of the platform (i.e. the actions and the allowed transitions between these actions) and to define which components perform a given action. The components that perform an action are specified by activity partitions (as an example, see C1, C2, and C3 in Fig. 1). A set of these components has to be integrated before our technique is performed according to a defined integration strategy. For each new integrated subsystem, our technique can be applied again.

To model variability, we stereotype certain elements in the activity diagrams to denote variation points (see VP1 in Fig. 1) and merge points (see MP1 in Fig. 1). We define a merge point as a specific node in the diagram where all variant control flows of a variation point are merged together and continue in a single control flow.
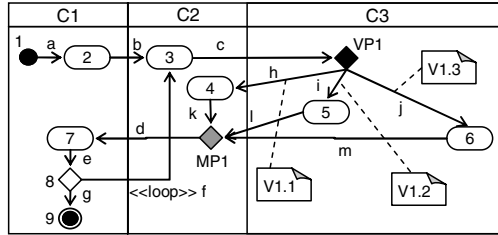

Fig. 1. Example of a Test Model

We assume that within the test model, variability always starts with a variation point. There are no variable parts that can be reached without passing through a variation point. Moreover, the control flows of all variants of a variation point are always merged together in exactly one associated merge point. The only exceptions are loops, which are explicitly marked by a stereotype (see <<loop>> in Fig. 1).

By using activity diagrams as test models, we follow other approaches where control flow graphs are used for this purpose (e.g. see [14]). In general, a test model can specify the behavior of the complete system or of the sub-system that should be integration tested. The abstraction level of the model strongly depends on the way the model has been developed. For example, the model can be developed on the basis of use cases from requirements engineering. If the scenarios of the use cases are documented in activity diagrams, these can be refined considering the architecture of the system. The abstraction level influences the quality of test results. The more abstract (i.e., less detailed) the model is, the less the coverage of the test object will be. Thus, the test model defines the quality of the test results.

3.2 Activities

Our technique for the generation of ITCSs for SPL consists of three main activities D1-D3. Fig. 2 shows these three activities together with their inputs and outputs.
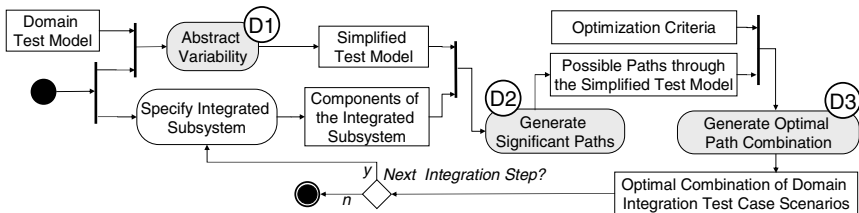

Fig. 2. Overview of our Technique

In the first activity D1, we abstract the variability of the given test model. The variability is the main problem that prevents the application of test techniques from single system development. Because of variability, no executable system exists in domain engineering. We abstract the variability and handle it as a black box for the ITCSs generation, because we want to test only the common parts of the platform (see Sec. 4.1). Thereby, the complexity of the test model is reduced. The relevant result of this activity is a simplified test model where the variability is abstracted.

In the second activity D2, all significant paths through the simplified test model are derived. For the derivation, we use Beizer's Node Reduction Algorithm [3]. Because the ITCSs that are generated with our technique only need to cover the interactions between the integrated components, we typically can reduce the number of paths (see Section 4.2). We refer to the reduced set of paths as significant paths.

In the third activity D3, we generate an optimal path combination, i.e. an optimal set of ITCSs, from the set of significant paths. We calculate the path combination with the minimal number of included abstracted variability. The variable parts within an ITCS have to be simulated by placeholders. In contrast to placeholders that are required for structural reasons, e.g. for enabling compilation, these placeholders are more complex because they have to simulate functionality. Therefore, the placeholders of the variable parts within the ITCSs represent a significant additional test effort in domain engineering that should be minimized to keep the overall testing effort reasonable.

4   Generation of Integration Test Case Scenarios

In this section, the activities of our technique are described in detail.

4.1   Abstraction of Variability (Activity D1)

In our technique, we consider variability in functionality (e.g. alternative control flows) as well as variability in the architecture (e.g. alternative components). If a component is a variant (i.e. it can be included in a customer-specific application or not), we will not integrate it in a sub-system for an integration test in domain engineering. The interactions between these components have to be tested in application engineering when the component is used as part of a specific application. Because variant components are not part of the integrated sub-system, they are simulated by placeholders in the same way that common components, which are not part of the integrated sub-system, are simulated.

In the test model, variability in functionality is represented by variation points, merge points and variants in the control flow. This variability is abstracted in the first activity D1. During this abstraction, the control flow between the variation point and the associated merge point as well as the variation point and the merge point itself are replaced by a new action. We define these new actions as abstracted variability.

Depending on the actual structure of the test model the abstraction is performed differently. We can differentiate between a normal arrangement of a variation point (see "A" in Fig. 3.) and a nested one (see "B" in Fig. 3.):
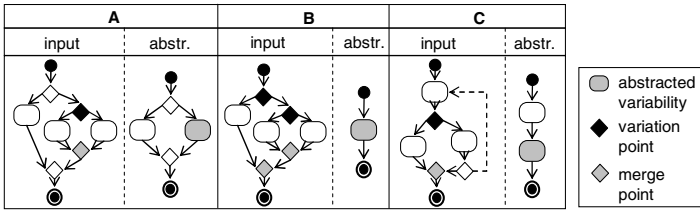
Fig. 3. Examples for Transformations for Abstracting Variability

Loops in a test model can be classified into three different types:

A backflow within a variable part exists: This type poses no problem for abstraction and is already covered by the abstraction of a normal variation point ("A").

A backflow out of a common part into a variable part exists: This type is not conforming to our assumptions from above. Because each variable part has to begin with a variation point, this composition is not allowed.

A backflow out of a variable part into a common part exists: This type represents a valid violation of our above assumption that all variants have to be merged in one merge point. Because we stereotyped all backflows, they can be identified and thus abstracted accordingly (see "C" in Fig. 3.).

The implementation of our abstraction algorithm uses a relation matrix as a data structure to represent the test model. We have decided to use relation matrices, because in the subsequent activities of our technique for the generation of integration test cases, we will use existing algorithms that are based on matrices.

The dimension of the relation matrix corresponds to the number of nodes in the test model (i.e. actions, decision points, variation points etc.). The entries of the matrix represent the edges between the different nodes.
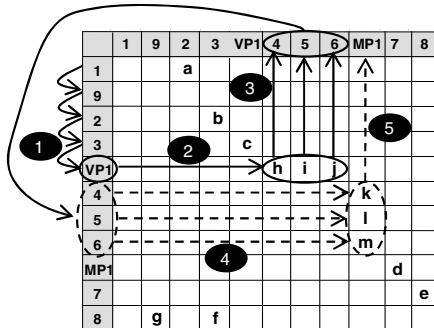

Fig. 4. Steps of the Abstraction Algorithm applied to the Example from Fig. 1

In Fig. 4. the steps of the abstraction algorithm are illustrated. The example test model consists of 11 nodes, including one variation point. The algorithm iterates through the rows of the matrix until a variation point is identified (see 1). With the

entries of the identified row (see 2), the reachable nodes are identified (see 3). For these nodes, again the reachable nodes are identified in the same way (see 4, 5). This procedure is repeated until the merge point of the identified variation point is reached. If a loop is detected, the procedure is finished for the respective loop node before reaching the merge node. The rows and columns that are identified with the algorithm are eliminated from the matrix and are replaced by a new action that represents abstracted variability.

4.2   Generation of Significant Paths (Activity D2)

The second activity D2 of our technique is divided into five steps (D2a – D2e). These steps are illustrated by the example in Fig. 5., in which components C1 and C2 are integrated into the sub-system to be tested.
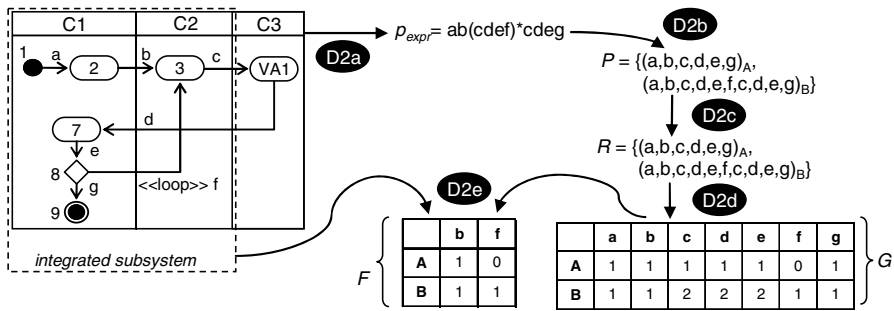

Fig. 5. Creation of an EP Matrix

Step D2a: Derivation of path expression. We start activity D2 with computing a path expression that represents all paths of the test model in a compact string. This path expression is generated by Beizer's node reduction algorithm of [3]. In the example, this leads to the expression "ab(cdef)*cdeg" (the asterisk '*' denotes an arbitrary repetition of the expression in parentheses).

Step D2b: Derivation of paths through the simplified test model. The path expression from step D2a can be used to derive all paths through the test model. However, if the test model contains loops, this can lead to an infinite number of paths. As we want to cover all interactions between the integrated components, we can limit the number of paths by restricting the loop iterations to at most one. This leads to a set $P := \{p_1, p_2, …, p_n\}$ of paths p_i. In the example, P contains the two paths A = (a, b, c, d, e, g) and B = (a, b, c, d, e, f, c, d, e, g).

Step D2c: Selection of significant paths for the sub-system. For the integration test of a given sub-system, only paths that affect the integrated sub-system need to be considered. Therefore, all paths that do not contain any edges that are associated to a component of the integrated sub-system are deleted. Further, all infeasible paths should be eliminated. The identification of infeasible paths is nontrivial. Still, several approaches have been suggested to identify infeasible paths (e.g., [13]). We suggest using one of these existing approaches to eliminate the infeasible paths. Also, a do-

main expert could perform this step. The result of this step is a set $R \subseteq P$ that contains all significant paths. In the example, paths A and B are significant, thus $R = P$.

Step D2d: Creation of an EP Matrix. To prepare for the following activities of our technique, an edge-path frequency matrix (EP matrix) is created from the paths in the set R. The rows of an EP matrix G represent the paths. The columns of the matrix represent the edges, $e \in E$, of the test model. An element $G(p, e)$ of the matrix G represents how often an edge e is contained in a path p. In the example, the element $G(B, c)$ of the EP matrix contains a value of 2, because edge c appears twice in path B.

Step D2e: Reduction of the EP Matrix. As a final step, the EP matrix is reduced. In integration testing, the interactions between the components of the integrated sub-system are tested. Therefore, only edges of the test model that cross the component boundaries within the integrated sub-system have to be considered. As a consequence, all columns can be eliminated that represent other transitions, leading to a reduced matrix F. In the example, transition e is eliminated, because it is an internal transition of component C1. Transition c is eliminated, because it is not part of the component interactions within the integrated sub-system.

4.3   Generation of the Optimal Path Combination (Activity D3)

The third activity D3 generates a set of ITCSs based on the EP matrix F. The set of ITCSs has to cover all necessary edges of the test model, i.e. all interactions between the integrated components. Moreover, the set should lead to the minimal number of abstracted variability and thus required placeholders.

An ITCS is represented by a path through the simplified test model. The optimal set of ITCSs therefore is represented by a subset c of the paths R that are specified in the matrix F. We want to determine a path combination that guarantees a complete coverage of the interactions between the integrated components. The coverage for any given path combination c can be computed from the EP matrix F as follows.

Let $s(c, e)$ be

$$s(c, e) = \sum_{p \in c} F(p, e) \tag{1}$$

The coverage of the interactions between the integrated components is only achieved if $s(c, e)$ is larger than zero for all edges e. Otherwise, at least one edge has not been considered in the path combination.

The generation of an optimal path combination through the simplified test model obviously represents an optimization problem. We use the generalized optimal path selection model by Wang et al. [27] to generate the optimal path combination. This model can easily be used for different optimization criteria. Wang et al. define the objective function Z as follows:

$$Z = b^{\mathrm{T}} W^{\mathrm{T}} x \tag{2}$$

The vector of decision variables x contains one decision variable for each significant path through the simplified test model. The decision variable indicates, whether

the path is selected for the path combination or not. Therefore, the decision variables are binary:

$$x = [\, (\, X_i \, | \, X_i = 0, \, 1) \,] \tag{3}$$

The vector b and the matrix W enable the weighting of the decision variables to upgrade or downgrade specific possible solutions.

The minimization of the abstracted variability that is contained in the significant paths is realized by the weighting matrix W. Thereto, we specify a matrix V. The rows of V represent the significant paths R. The columns of V represent the nodes v within the simplified test model that represent the abstracted variability. An element $V(p, v)$ of the matrix V is 1 iff the path p contains the node v, otherwise the element has the value 0. The matrix V is now used as weighting matrix W.

The vector b can be used to prioritize the complexity of the single abstracted variability. In contrast to this, the matrix V is used to optimize the number of abstracted variability in the path combination. Because the complexity of abstracted variability depends on many different factors (e.g., on the way of implementation), we currently do not use the vector b in our optimization. The resulting objective function thus becomes:

$$Z = 1^\mathrm{T} \, V^\mathrm{T} \, x \tag{4}$$

It should be noted that using the matrix V leads to an approximation with respect to minimizing the number of abstracted variability within the selected path combination. Because of an easier realization, the overall number of abstracted variability is minimized, not the number of different abstracted variability.

The desired coverage of the test model is achieved by defining auxiliary conditions:

$$A^\mathrm{T} \, x \geq r \tag{5}$$

The matrix A defines the elements that have to be covered (e.g., paths, branches, or nodes). The variable r specifies the desired degree of coverage. We specify the coverage of the interactions between the integrated components with the matrix F. Therefore, we can replace the matrix A with matrix F. To guarantee the coverage, it is sufficient to run through each needed edge once. Therefore, we can set the degree of coverage r to 1. The adapted auxiliary conditions are the following:

$$F^\mathrm{T} \, x \geq 1 \tag{6}$$

The optimization problem that is described in this manner can be solved with the branch-and-bound approach (see e.g. [1]). Branch-and-bound is a general algorithmic method for finding optimal solutions of integer optimization problems.

5   Evaluation of the Technique

In this section, we present the results of the experimental evaluation of our technique concerning the performance and the benefits of the technique.

5.1   Design of the Experiment

We have implemented a prototype for the complete technique in Java (JDK 1.4.2). The multi-constraint selection of an optimal set of paths through a graph is an NP complete problem. Therefore, the activity D3 of our technique is the most critical one and we focus on this activity in our evaluation. Our prototype uses for this activity D3 the Gnu Linear Programming Kit (GLPK, [11]) for solving the optimization problem with the branch-and-bound approach.

We have defined simplified test models by the random generation of EP matrices and W matrices. Then, we have generated the optimal set of ITCSs by applying our prototype. We have measured the computation time as well as the number of selected scenarios and the number of the contained abstracted variability. Altogether, we have generated and calculated over 2000 examples. All measurements have been performed on a standard PC with a 2.8 GHz Pentium IV processor and 1 GB RAM, running Windows XP (SP2).

The test models that have been used as input to activity D3 are represented by EP matrices and W matrices and can be characterized by six parameters:

1. the number of paths through the simplified test model (rows of the EP matrix)
2. the number of reduced interactions between the integrated components (columns of the EP matrix)
3. the number of entries of the EP matrix, i.e. how many cells of the EP matrix have a value greater than 0
4. the allocation of the EP matrix, i.e. the assignment of the values greater than 0 to the cells of the EP matrix
5. the number of abstracted variability in the simplified test model (columns of the W matrix)
6. the allocation of the W matrix, i.e. the assignment of the values greater than 0 to the cells of the W matrix

For the execution of the test runs, we have varied the number of paths and interactions of the EP matrix. We have started with 10 paths and 10 interactions and increased the values incrementally by 10. For each paths/interactions combination, we generated 5 examples with different kinds of allocations and different W matrices. Because of our experience with previous example test models, we have fixed the number of entries in the EP matrices to 50% and 25%. The allocation has been generated randomly. We prevent columns that contain a value greater than 0 in every row, because these columns would be covered by every path and therefore they would be needless and could be deleted. We also prevent columns that contain 0 in every row, because this would lead to a wrong test model. These columns could not be covered by any path through the simplified test model. The bounds of the number of abstracted variability, *VA*, in the simplified test models (i.e., the number of columns in the W matrices) are the following: $1 < VA < (10*(\text{columns of EP}))$. We assume that only 10% of the interactions of the original test model represent interactions between the integrated components. If more than 10% of the interactions represent interactions between the integrated components, the upper bound would be lower and less variability could be included in the model (e.g., 5*columns of EP for 20%). The number of entries and also the allocation of the matrix W have been generated randomly.

5.2   Validity Threats

We have analyzed different types of threats to the validity of the results of our evalua-
tion (c.f. [28]). One threat leads to the necessity of using objective and repeatable
measures. We have measured the computation time, the number of test case scenarios,
and the number of reduced abstracted variability. These measures are objective, be-
cause it involves simple counting. Because the technique is automated, the measures
are also repeatable.

  The environment of our experimentation consists of a PC that has no network con-
nection and there are no other applications running on the PC. We have performed all
measurements on the same PC. Thus, the results are comparable. The implementation
of the prototype has been intensively tested before we have performed the experiment.

  Because of the high number of more than 2000 computations of different test mod-
els, in our opinion the results can be generalized and with a high probability they are
also relevant for industrial practice.

5.3   Performance of the Technique

In our experiment, the performance is measured by computation time. We have ap-
plied our prototype to more than 2000 generated examples with different characteris-
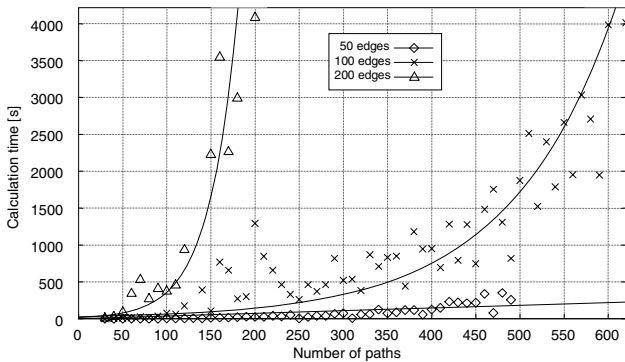tics of EP matrices and W matrices.

Fig. 6. Computation Times [s]

  Fig. 6 shows the results concerning the computation time exemplary for EP matri-
ces with 50, 100, and 200 edges and an allocation of 50%. The results in the diagram
respectively represent average values of five computations. Moreover, the figure
shows the regression curves for each data set. As expected, simplified test models
with more included interactions need more computation time. The longest computa-
tion time of all executed computations was 9087 seconds for an example with 200
paths, 200 interactions (edges), and 1880 included abstracted variability. The variabil-
ity in this example could only be reduced to 1842. Examples with 100 interactions
have been smoothly calculated in an acceptable time, even if they have 600 paths. The
computation time of examples with 50 interactions was minimal. Analyzing the re-

sults, we observed that the more variability could be reduced, the more computation time decreased. The most complex computations were those, where no variability could be reduced.

It should be noted that the measured computation times correspond to the dimensions of the simplified test model. Thus, directly determining computation time from the size of the (non-simplified) test model is not possible. However, the size of test models typically corresponds to a multiple of the size of the simplified test models. Because all computations of our experiment have been performed in an acceptable time (less than three hours), these results indicate that (non-simplified) test models of sufficient size can be calculated.

5.4 Benefit of the Technique

Our technique supports an early test in domain engineering because of the following aspects:

The complete system is not needed for applying our technique, because missing parts (e.g., variability or not implemented components) can be simulated.

The parts of the system that have to simulated are easily to identify. The variable parts can be identified by the nodes in the ITCSs that represent abstracted variability. The placeholders that are needed because of structural reasons can be identified by the architectural information within the ITCSs.

The expected coverage is guaranteed. All common interactions between the components of the system are covered.

There are two additional benefits that could be measured in our experiment:

Low number of integration test case scenarios. Applying our technique leads to a small set of ITCSs that guarantees the coverage of all necessary interactions of the test model. A small test set can reduce the test effort during test execution.
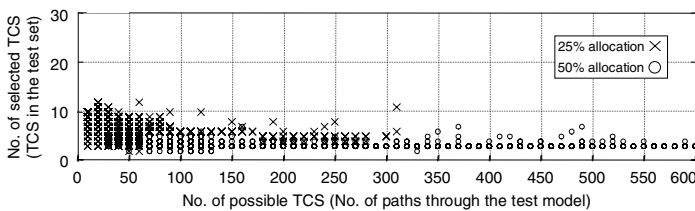

Fig. 7. Number of selected TCS

Fig. 7. shows the number of ITCSs within the generated test set in contrast to the number of possible ITCSs for 25% and 50% allocations of the EP matrix. The number of scenarios within the generated test set is very small, i.e. only a few ITCSs are sufficient to cover all necessary interactions of the test model. Although the number is not minimized during the computation and the generated test models are partially very

complex, the average number of scenarios for 25% allocations of the EP matrix is 7.2 scenarios. On an average only 3.6 scenarios are sufficient for the coverage, if the EP allocation is 50%.

Minimal number of abstracted variability. Applying our technique leads to a test set that contains a minimal number of abstracted variability. The minimal number of abstracted variability reduces the test effort during the preparation of the test, because less placeholders have to be developed.
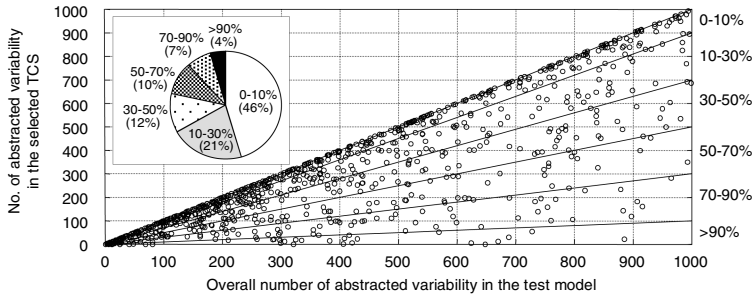

Fig. 8. Measured reduction of Variability

Fig. 8 shows the number of different abstracted variability in the selected ITCSs proportional to the overall number of different abstracted variability in the test model (50% allocation of EP). The difference between these numbers represents the reduction of variability and therefore a reduction of needed placeholders. In the diagram, the numbers of all performed computations are illustrated. The mean reduction is 25%. However, the standard deviation is very high and therefore the mean reduction is not really significant. Because of the high distribution of the values, we have divided the values in six categories. In 46% of the computations, only a reduction of less than 10% could be reached. But in more than 20% of the computations, a reduction of more than 50% could be reached. 4% of the computations lead to reduction of more than 90%. Summarizing, a high reduction is possible, but because of the high distribution of the results estimating the reduction for a given test model is not possible. The high distribution is due to the fact that the results are influenced by a set of parameters (e.g., allocation of the variability in the test model, size of the test model).

6   Conclusion and Outlook

In this paper, we have presented a model-based, automated technique for integration testing in domain engineering. The technique generates integration test case scenarios, based on which integration test cases can be developed. Our technique provides four significant benefits for software product line testing:

> The technique facilitates integration testing by considering a test model that describes the control flow as well as its assignment to the components of the software product line platform. Components and variable parts, which have to be simulated by placeholders, are explicitly modeled.

The technique supports an early test in domain engineering. Variability is abstracted and placeholders can be used to simulate the abstracted variable parts of the software product line platform.

The technique selects integration test case scenarios systematically. Based on the test model, the integration test case scenarios are derived by our technique such that the coverage of all interactions between the components of an integrated subsystem is guaranteed.

The technique reduces the testing effort, because the variable parts within the selected integration test case scenarios, and thus the development effort for placeholders, is minimized. Our experimental evaluation has shown that on an average the number of variable parts in the scenarios can be reduced by about 25%.

Although the computations for minimizing the variable parts in the integration test case scenarios are quite complex, our experiments have shown that the technique can deal with large test models. However, the technique depends on the arrangement of variability in the test models. The abstraction of an unfavorable arrangement can – under rare circumstances – lead to an over-simplified test model, which prevents a reasonable application of the technique.

Currently, we are planning to apply our testing technique in an industrial setting. In addition, the reuse of the generated integration test case scenarios in application engineering is one topic of our future research. We are convinced that the testing effort can significantly be reduced, if the test case scenarios are systematically reused in application engineering.

References

1. Balas, E.; Toth, P.: Branch and Bound Methods. In: Lawler, E.L.; Lenstra, J.K.; Rinnooy Kan, A.H.G.; Shmoys, D.B. (eds.): The Traveling Salesman Problem, Wiley, New York (1985) 361-401

2. Basanieri, F.; Bertolino, A.: A Practical approach to UML-based derivation of integration tests, In: Proc. of the Quality Week Europe, paper 3T (2000)

3. Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold, New York (1990)

4. Bertolino, A., and Gnesi, S. PLUTO: A Test Methodology for Product Families. In: Software Product-Family Engineering – 5th Intl. Workshop, LNCS 3014, Springer (2004)

5. Bertolino, A.; Marré, M.: Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. IEEE Transactions on Software Engineering, Vol. 20, No. 12 (1994) 885-899

6. Binder, R.V.: Testing Object-Oriented Systems. Addison-Wesley (2000)

7. Boehm, B.; Basili, V.R.: Software Defect Reduction Top 10 List. IEEE Computer, 34(1) (2001) 135-137

8. Clements, P., and Northrop, L. Software Product Lines: Practices and Patterns. Addison-Wesley (2002)

9. Cohen, M.B.; Dwyer, M.B.; Shi, J.: Coverage and Adequacy in Software Product Line Testing. In: Proc. of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, ACM, New York (2006) 53-63

10. Geppert, B.; Li, J.; Rößler, F.; Weiss, D.M.: Towards Generating Acceptance Tests for Product Lines. In: Proc. of the 8th Intl. Conf. on Software Reuse, LNCS 3107, Springer, Heidelberg (2004) 35-48

11. GLPK (Gnu Linear Progrmming Kit), Gnu Project, http://www.gnu.org/software/glpk/.
12. Hartmann, J.; Imoberdorf, C.; Meisinger, M.: UML-Based Integration Testing, In: Harrold, M.J. (ed.): Proc. of the Intl. Symposium on Software Testing and Analysis, ACM, New York (2000) 60-70
13. Hedley, D.; Hennell, M.A.: The Causes and Effects of Infeasible Path in Computer Programs, In: Proc. of the 8th Intl. Conf. on Software Engineering, IEEE (1985) 259-267
14. Jorgensen, P.C.; Erickson, C.: Object-Oriented Integration Testing. Communications of the ACM, Vol. 37, No. 9 (1994) 30-38
15. Käkölä, T.; Duenas, J.C. (Eds.): Software Product Lines – Research Issues in Engineering and Management. Springer (2006)
16. Kim, Y.; Carlson, C.R.: Scenario Based Integration Testing for Object-Oriented Software Development, In: Proc. of the 8th Asian Test Symposium, IEEE (1999) 383-288
17. McGregor, J.D. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon University, SEI (2001)
18. McGregor, J.D., Sodhani, P., and Madhavapeddi, S. Testing Variability in a Software Product Line. In Proc. of the Intl. Workshop on Software Product Line Testing, Avaya Labs, ALR-2004-031 (2004) 45-50
19. Muccini, H.; van der Hoek, A.: Towards Testing Product Line Architectures. In: Proc. of the Intl. Workshop on Test and Analysis of Component-Based Systems, Electronic Notes in Theoretical Computer Science, Vol. 82, No. 6 (2003)
20. Nebut, C.; Fleurey, F.; Le Traon, Y.; Jézéquel, J.-M.: A Requirement-based Approach to Test Product Families. In: Software Product-Family Engineering – 5th Intl. Workshop, LNCS 3014, Springer, Heidelberg (2004) 198-210
21. Pohl, K.; Böckle, G.; van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
22. Prather, R.E.; Myers, J.P.: The Path Prefix Testing Strategy. IEEE Transactions on Software Engineering, Vol. 13, No. 7 (1987) 761-766
23. Reis, S.; Metzger, A.; Pohl, K.: A Reuse technique for Performance Testing of Software Product Lines. In: Proc. of the Intl. Workshop on Software Product Line Testing, Mannheim University of Applied Sciences, Report No. 003.06, (2006) 5-10
24. Reuys, A.; Kamsties, E.; Pohl, K, Reis, S.: Model-based System Testing of Software Product Families. In: Advanced Information Systems Engineering - CAiSE 2005, LNCS 3520, Springer, Heidelberg (2005) 519-534
25. Tevanlinna, A., Taina, J., and Kauppinen, R., Product Family Testing – a Survey. ACM SIGSOFT Software Engineering Notes, 29(2) (2004)
26. Tsai, W.T.; Bai, X.; Paul, R.; Shao, W.; Agarwal, V.: End-To-End Integration Testing Design. In: Proc. of the 25th Annual Intl. Computer Software and Applications Conf., IEEE, Los Alamitos (2001) 166-171
27. Wang, H.S.; Hsu, S.R.; Lin, J.C.: A Generalized Optimal Path-Selection Model for Structural Program Testing. The Journal of Systems and Software, Vol. 10 (1989) 55-63
28. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslen, A.: Experimentation in Software Engineering – An Introduction. Kluwer Academic Publishers (2000)
29. Wu, Y.; Chen, M.-H.; Offutt, J.: UML-Based Integration Testing for Component-Based Software. In: Proc. of the 2nd Intl. Conf. on COTS-Based Software Systems, LNCS 2580, Springer (2003) 251-260