Towards a Semantic Framework for Software Product Line Test Coverage

Ruben Haasjes
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.e.y.haasjes@student.utwente.nl

ABSTRACT

Software product line engineering has helped the industry to efficiently create families of similar products. However, the process of testing these products is not as advanced as the development techniques. Testing methods to efficiently test software product lines have been developed, but the majority of these techniques define test coverage in terms of syntactic characteristics. The problem with syntactic coverage criteria is that they give no insight in the actual behavior that is tested. This could lead to inefficient or inadequate testing. This study presents a more semantic definition of software product line test coverage. This semantic notion expresses test coverage in terms of weights, related to the behavior and variability in a software product line. These weights could be used to select and prioritize test cases based on their importance, resulting in more meaningful testing.

Keywords

Integration Testing, Software Product Line, Test Coverage

1. INTRODUCTION

Software product line (SPL) engineering is a methodology for developing collections of similar software. The key concept is the reuse of common and variable features. Features are either included or excluded resulting in a set of similar, yet unique software systems. SPL engineering has proven to enable developers to create collections of similar software systems more efficiently [4]. Figure 1 gives a simple example of a software product line in the form of a feature diagram (FD). A FD is a commonly used method for modeling the variety and commonality in a SPL. The FD in the example models a SPL of coffee vending machines. It visualizes the different options for drinks and payment.

SPL engineering is used in a variety of products. Examples are vending machines, cars, e-shops and numerous others. SPL engineering is also applied in safety-critical systems, thus making testing of vital importance [3]. An example that illustrates the tremendous implications of insufficient testing is the Ariane 5 crash. It took the European space agency approximately 10 years and 7 billion dollars to produce this rocket. A lot of the software architecture from

25^{th} *Twente Student Conference on IT* July 1^{st}, 2016, Enschede, The Netherlands.

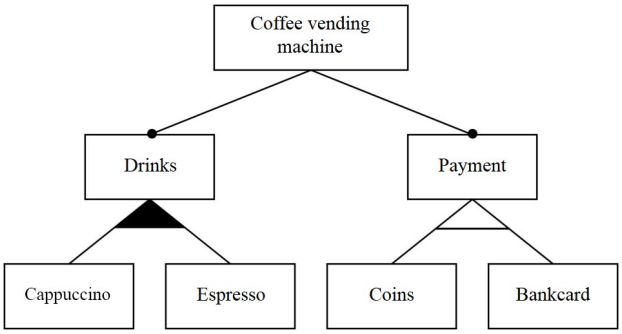

Figure 1. Coffee vending machine example. Inspired by a vending SPL example. [7]

similar rockets like the Ariane 4 was used in the Ariane 5. The horizontal velocity of the rocket was stored in 16-bits because of the re-use of software. However, the Ariane 5 was significantly faster than predecessors, resulting in a velocity that could not be stored in 16-bits and ultimately, the crash of the Ariane 5 [8].

The need for adequate testing is clear, but adequate testing is difficult to define. When discussing SPL's in particular it is argued that the most critical point of testing is the interaction between components. The question remains at what point a SPL has been tested thoroughly. In theory testing is never done. New test cases can always be derived, for example with larger input values or by repeating test cases for unexpected behavior. This is especially difficult in a SPL, testing one product thoroughly is difficult but testing each possible product in a SPL is not feasible for larger product lines [3, 11].

In recent years methods for more efficient SPL testing have been developed. One approach uses the FD of a SPL. By testing interactions between features in representative products, the problem of testing each product individually can be reduced. Another approach uses a behavioral model of the SPL. The model contains all the possible behavioral actions of all possible products. Then test suites can be generated at SPL level. One common aspect among these approaches is that their coverage measure is often defined in terms of syntactic characteristics. Testing all the possible interactions between features, or all the actions that can be done in a behavioral model are examples. An issue with these syntactic definitions of coverage is that they do not take the actual behavior of a SPL into account, which can result in inefficient or inadequate testing.

2. RESEARCH GOAL

2.1 Problem statement

The problem with coverage measures in terms of syntactic

characteristics is that the actual coverage of a test becomes difficult to define. Syntactic characteristics give little insight in the actual behavior that is tested. For example, the importance and impact of an error are not taken into account. This implies that coverage might be measured differently for syntactically different but behaviorally similar products. Another problem is that a far more important component does not account heavier towards the coverage than a less important component. Therefore the aim of this study is to define software product line test coverage in a more semantic manner.

2.2 Research questions

The scope of this research is integration testing. The main question during this research is: **How can test coverage be defined for software product line testing?**

In order to answer the research question the following questions will be answered:

Q1: To what extent are test coverage definitions for current testing methods for software product line testing semantic?

Q2: What is a suitable (behavioral) model for specifying test coverage for software product line testing?

Q3: How applicable is this model to a real-life example?

3. BACKGROUND

3.1 SPL integration testing

The key concept in software product lines is variability. Products in software product lines are created by including or excluding commonly used functional units. Every individual feature can be thoroughly tested. This however does not guarantee that interaction between units in a specific product will work. The faults that only occur under specific combinations of units are called *interaction faults*. Therefore testing all reasonable combinations of units seems necessary. However testing each possible product combination seems not feasible, since the amount of tests needed will grow exponentially with the amount of features [11].

3.2 Feature diagrams

A way to reduce the amount of tests needed to test a complete SPL is by using feature diagrams. In this paper the definitions as specified by B. Batory [1] are used. FDs are build using the following symbols:

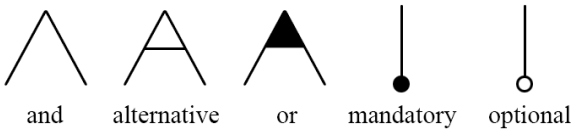

Figure 2. The symbols used in a feature diagram.

The symbols have the following meanings:

- And: Both features must be selected.
- Alternative: Only one feature can be selected.
- Or: One or more features can be selected.
- Mandatory: The feature must be selected.
- Optional: The feature can be selected.

A test case derived from a FD is an interaction between a set of features. For example a test case for the FD presented in figure 1 could be testing if an espresso can be bought using coins.

3.3 Featured transition systems

Although feature diagrams are capable of modeling the complete variability in a SPL, they give little information about the actual behavior in a SPL. Feature transition systems (FTS) effectively model the behavior of a complete SPL. This paper uses the FTS as defined by X. Devroey et al [7]. An FTS is defined as a tuple: (S, Act, trans, i, d, γ) where

- S is a set of states;
- Act is a set of actions;
- trans $\subseteq S \times A \times S$ is the transition relation;
- i $\in$ S is the initial state;
- d is a feature diagram;
- γ labels each transition with a boolean expression over the features specified in d, that are able to execute the transition. The expression is specified using classical boolean operators.

Below a simple FTS of the behavior of the coffee vending machine.

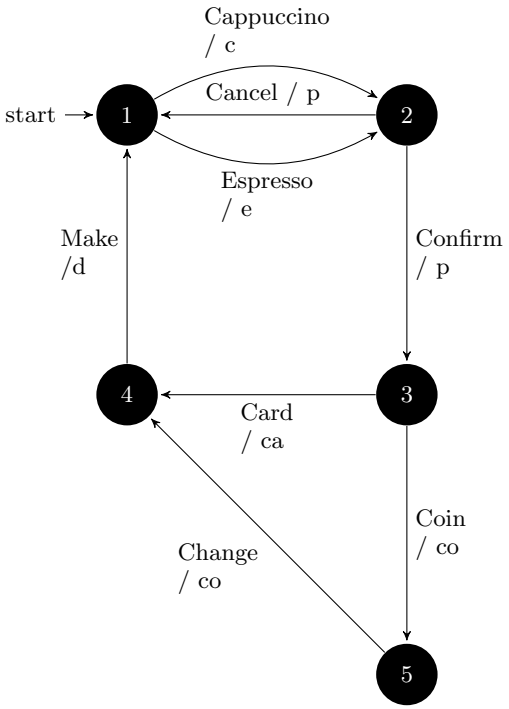

Figure 3. Example weighted featured transition system for the coffee vending machine.

DEFINITION 1. *A* test case *is a finite sequence T of actions.*

For the FTS of the simple coffee machine SPL a possible test case could be: *(Espresso, Confirm, Pay by card, Take product)*. These test cases are then mapped to input values and actions in the actual system.

4. RELATED WORK

4.1 Test coverage

A study relevant to the one in here, is by L. Brandan Briones et al. [2] on a semantic framework for test coverage. The study deals with the problem of test coverage in general being specified syntactically. They describe how weighted fault models can be used to express to what extent a test suite covers a specification and furthermore provide algorithms to calculate the minimal test suit with maximal coverage.

Using their algorithms the user can test more efficiently, by testing important safety-critical aspects more thoroughly than less critical system parts.

4.2 Test case prioritization

R. Lachmann et al. [9] have addressed one of the issues discussed in the problem statement, testing more import aspects more thoroughly then less critical aspects. They have developed a method for prioritizing test cases. For their approach they assume that test cases for every product variant have already been generated, using an incremental product-by-product approach. The first step in their approach is to calculate the differences between products, the so called regression delta. Afterwards they create delta graphs that are used to calculate the degree of change between the product currently being tested and the previously tested product. Using the delta graphs, weights are calculated for each component in a product under test. These weights are then used to order and prioritize test cases.

5. CONTEMPORARY TEST COVERAGE DEFINITIONS

5.1 FDs and test coverage

Recent SPL testing techniques [6] use a FD as input in order to generate samples. By doing this they a small relevant subset can be tested instead of all the possible product variations.

The most common coverage criteria for sample selection is that every *valid combinatorial pair* of features should be covered in at least one product [10].

5.2 FTSs and test coverage

The coverage of tests with regard to a FTS are usually specified using one of the following definitions.

- *(State / all states coverage)* The states coverage, is the ratio between visited states in all test cases off a test suite and the total amount of states in a FTS.
- *(Transition / all transition coverage)* The transition coverage, is the ratio between all visited transitions in all test cases off a test suite and the total amount of transitions in a FTS.
- *(Transition pairs / all-transition pairs)* The transition pairs coverage, is the ratio between adjacent transitions successfully entering and leaving a state visited in all test cases off a test suite and the total amount of possible adjacent transitions, entering and leaving a state.
- *(Path / all paths coverage)* The path coverage, is the ratio between covered executable paths in all test cases of a test suite and all the executable paths. Here an executable path is a sequence of actions such that the γ associated with the sequence of actions is valid with the FD.

With one of the these definitions of coverage, test cases are derived from the FTS.

6. A SEMANTIC DEFINITION OF TEST COVERAGE

In practice, a lot of undesired behavior and errors are related to the interaction between features, thus a coverage defined as all the valid combinatorial pairs of features seems reasonable [6]. However, this definition of coverage has little relevance when thinking about the actual semantic or behavioral coverage. The FTS test coverage definitions presented by X. Devroey at al. [6] have more relation to the actual behavior of the SPL. However, these definitions do not differentiate in importance of certain behavior.

In this section we define coverage for SPL's in a semantic manner. The main challenge when defining coverage for a SPL, is that all the possible actions are weighted equally. But this makes little sense, since some actions are obviously more important than others. For example when testing a product line of cars, testing the brakes or the airbag is far more important than the radio.

6.1 Risk

In order to differentiate between importance of certain actions we use a classical definition of *risk*.

DEFINITION 2. *The* risk *R of an action is $R \in \mathbb{R}_+$ obtained from the* probability *that a fault occurs multiplied with the* impact *of the fault.*

In other words, the risk expresses the likelihood that a fault will occur when performing a certain action, and its consequences. The risk can be expressed as a matrix with arbitrary values for probability and impact, see Table 1.

Table 1. Example risk matrix, with 1-5 for probability and impact. Columns for probability, rows for impact.

	Very low	Low	Medium	High	Very high
Very high	5	10	15	20	25
High	4	8	12	16	20
Medium	3	6	9	12	15
Low	2	4	6	8	10
Very low	1	2	3	4	5

Any numeric scale for probability and impact can be used. Depending on how much difference in risk there is. We propose to add this notion of *risk* to a FTS. Thus we extend the FTS tuple with r resulting in Weighted Featured Transition System: WFTS = (S, Act, trans i, d, γ, r) where r is a function that labels every transition with a *risk* factor. Risk analysis should be done in cooperation with an expert.

We created an example WFTS for the FTS of the coffee vending machine. Faults that involve paying, or receiving the product are deemed to have more *risk* than selecting, confirming and canceling your choice.

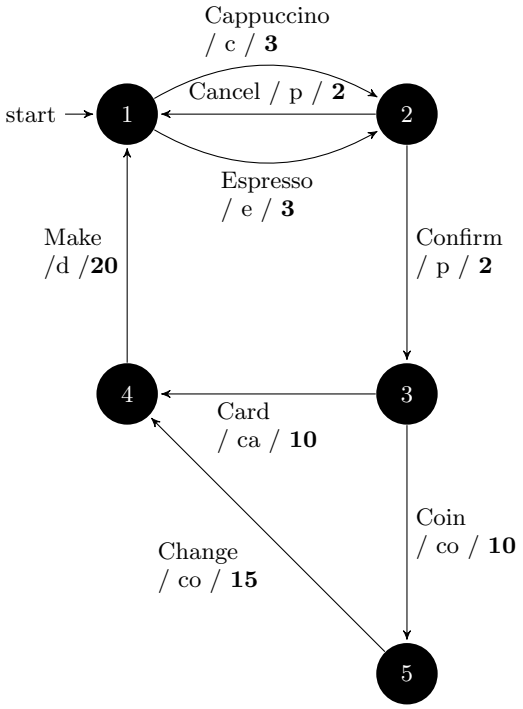

Figure 4. Example weighted featured transition system for the coffee vending machine.

6.2 Test validity

The WFTS is an efficient way of modeling all behavior of a SPL. However, tests could be derived for which no product can ever exist. Therefore only tests that are *valid* should be considered.

Definition 3. *A test case is* valid *if there exists at least one product that is able to execute the test.*

Checking whether a product exists that is able to execute a test case can be done using propositional logic. The relations in a FD can be translated in the following way [12]:

Table 2. Feature diagram to propositional logic mapping

Feature Diagram	Propositional formula
f_1 is an optional sub-feature of f	$f_1 \rightarrow f$
f_1 is a mandatory sub-feature of f	$f_1 \leftrightarrow f$
$f_1, .., f_n$ or sub-features of f	$f_1 \vee .. \vee f_n \leftrightarrow f$
$f_1, .., f_n$ alternative sub-features of f	$(f_1 \vee .. \vee f_n \leftrightarrow f) \wedge \bigwedge_{i<j} \neg(f_i \wedge f_j)$

For example the coffee vending machine FD could be translated into the following rules:

- Coffee machine
- Drinks $\leftrightarrow$ Coffee Machine
- Payment $\leftrightarrow$ Coffee Machine
- Cappuccino $\vee$ Espresso $\leftrightarrow$ Drinks
- (Coins $\vee$ Bankcard $\leftrightarrow$ Payment) $\wedge\neg$(Coins $\wedge$ Bankcard)

Each action in a test case is labeled with a feature expression. If the total feature expression combined with the feature expression from the FD is *satisfiable*, a product exists that is able to execute the test. By *satisfiable* we mean, that there is at least one interpretation or model of the expression that makes the expression true.

Many tools have been developed for solving the above mentioned question, e.g. SAT-solvers. Using SAT-solvers all the possible products with respect to the FD can be derived. Furthermore using the feature expression of a test case, all the products able to execute the test can be derived. A problem is that the satibility problem is NP-complete. Thus if many SAT checks are done on a large model, it might lead to performance issues. However the study by X. Devreoy et al. [7] shows that, when used efficiently, performance is no issue, even in a model with over 60 states and over a 1000 transitions.

6.3 Action sequence relevance percentage

For our definition of test coverage not only the risk of an action is important. A product line consists of a variety of products. Therefore the *percentage of relevance* of each action in a test case should be considered. Or in other words: *which percentage of the products is able to execute the particular sequence of actions?*

Definition 4. *The* action sequence relevance percentage*, is the percentage of products that is able to execute the particular sequence of actions.*

In order to derive this percentage, expert knowledge about the production amounts is needed. Let us take the coffee vending machine as an example. From the translated FD to propositional logic we can derive all possible products. We have added sample production values to do some simple calculations.

Table 3. Product configurations of the coffee vending machine example, with sample production amounts

Configuration	Amount
{d, p, c, co}	25
{d, p, c, ca}	50
{d, p, e, ca}	50
{d, p, e, co}	25
{d, p, c, e, co}	100
{d, p, c, e, ca}	200
Total production amount	450

If we have action sequence *(Cappuccino, Confirm)* the feature expression becomes: c $\wedge$ p. The following products satisfy both the feature expression of the action sequence and the feature expression of the FD: {d, p, c, co}, {d, p, c, ca}, {d, p, c, e, co}, {d, p, c, e, ca}. Thus 4 out of 6 products are able to execute the action sequence *(Cappuccino, Confirm)*. If we map the products with actual production amounts the action sequence relevance percentage of action sequence *(Cappuccino, Confirm)* becomes: $\frac{25+50+100+200}{450} = \frac{375}{450} \approx 83\%$. Or in other words, 83% of the products is able to execute this action sequence.

6.4 Test coverage in a SPL

Thus when defining test coverage for SPLs, we consider the following aspects:

- The *risks* that can be found in a test case.
- The *relevance percentages* of the action sequences in a test case.

The test coverage in a single test case can be found in the following manner. We propose to rewrite the test as multiple action sequences. Each action in a test case is replaced with the complete sequence to get to the action. Let us illustrate this according to a single test case in the coffee vending machine example: *(Cappuccino, Confirm, Card, Make)* will be rewritten this as *(Cappuccino), (Cappuccino, Confirm), (Cappuccino, Confirm, Card), (Cappuccino, Confirm, Card, Make)*.

The next step is to calculate the *weight* of each action sequence. Since we rewrote the test case as multiple action sequences, the weight is acquired by taking the risk of the last action in the action sequence and multiplying this with the action sequence relevance percentage.

Thus the coverage of this test case can be calculated in the following manner. Test coverage of *(Cappuccino, Confirm, Card, Make) = weight(Cappuccino) + weight(Cappuccino, Confirm) + weight(Cappuccino, Confirm, Card) + weight(Cappuccino, Confirm, Card, Make)*.

The risk of Cappuccino is 3 and the action sequence *(Cappuccino)* corresponds using SAT-solving to a relevance of 4/6 product variations, or mapped with the sample production amounts to a percentage of approximately 83%. The risk of Confirm is 2 and the action sequence relevance percentage of *(Cappuccino, Confirm)* also corresponds to approximately 83%. The risk of Card is 10 and the action sequence *(Cappuccino, Confirm, Card)* using SAT-solving is relevant for 2 / 6 product variations, or mapped with the sample production amounts to $\frac{250}{450} \approx 56\%$. The risk of Make is 20 and the action sequence relevance percentage of *(Cappuccino, Confirm, Card, Make)* is also approximately 56%.

Thus we can calculate: test coverage*(Cappuccino, Confirm, Card, Make)* = (3 * 0.83 + 2 * 0.83 + 10 * 0.56 + 20 * 0.56) ≈ 20.95.

6.4.1 Total coverage in test suite

We are interested in the test coverage of a complete test suite. The test coverage of a complete test suite is the sum of the coverage of each particular test case. However, identical test cases should not be added multiple times.

For example if we have test suite {*(Cappuccino, Confirm), (Cappuccino, Cancel)* }, the weight of Cappuccino should not be counted twice, since it is essentially the same test. But this can be avoided by writing the test cases as action sequences. Then the test suite becomes: {*(Cappuccino), (Cappuccino, Confirm)*}, {*(Cappuccino), (Cappuccino, Cancel)*}. Now the duplicate weights can be filtered by taking the union of all the action sequences in the test suite. We call this the *supertest*. The supertest for this example will result in: {*(Cappuccino), (Cappuccino, Confirm), (Cappuccino, Cancel)*}. The test coverage of a test suite can then be found by summing all the weights of each action sequence. Figures [5-7] provide a visual representation of this process.

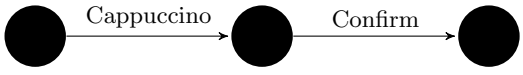

Figure 5. Test case *(Cappuccino, Confirm)*

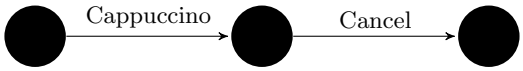

Figure 6. Test case *(Cappuccino, Cancel)*

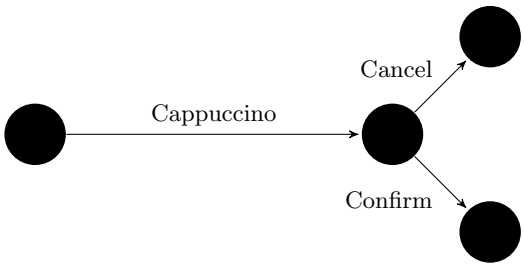

Figure 7. Supertest {*(Cappuccino, Confirm), (Cappuccino, Cancel)*}

Definition 5. *We define:*

- *The* absolute coverage *is the sum of all the action sequence weights in a supertest;*
- *The* total coverage *is the absolute coverage of the supertest of all valid test cases;*
- *The* relative coverage *is the absolute coverage divided by the total coverage.*

6.5 Algorithms

The total coverage can be found by taking the absolute coverage of the supertest of all valid test cases. This can cause problems. Because it could occur that the amount of possible valid test cases is infinite. If we consider our vending machine WFTS we could keep repeating actions indefinitely long. To solve this problem we could consider test cases of finite length only. In the finite test length approach only test cases with length $k \in \mathbb{N}$ and $k >= 1$ are deemed valid. Algorithm 1 is used for calculating the absolute coverage in a test suite.

Algorithm 1: Calculating absolute coverage in a test suite

```
Data: WFTS f, test suite T, finite length k
Result: absolute coverage in a test suite
1  supertest ← ∅;
2  for each test in T do
3  |  for each actionsequence of test do
4  |  |  if length of actionsequence <= k then
5  |  |  |  supertest ← supertest ∪ actionsequence;
6  |  |  end
7  |  end
8  end
9  abscov ← 0;
10 for each sequence in supertest do
11 |  abscov ← abscov + weight of sequence;
12 end
13 return abscov;
```

Line 1-8 describe the process of creating the supertest. The weight of a sequence, mentioned on line 11 in Algorithm 1 is calculated using Algorithm 2.

The total coverage in a WFTS can then be found by giving all valid test of finite test length equal or smaller than k as input to Algorithm 1.

Algorithm 2: Calculating the weight of an action sequence

Data: WFTS f, action sequence S
Result: weight of an action sequence
1 risk ← risk of last action in S
2 arp ← action relevance percentage of S
3 return risk * arp;

6.6 Example

We use the coffee machine WFTS f for our example. In this example, for a finite length k, the total, absolute and relative coverage is calculated. Because the set of valid tests grows in size rapidly we take $k = 2$.

The total coverage is calculated using Algorithm 1. The set of all valid tests with length $k <= 2$ is given as input: *(Cappuccino), (Espresso), (Cappuccino, Cancel), (Cappuccino, Confirm), (Espresso, Cancel), (Espresso, Confirm).*

The output of the supertest is equal to the set of all valid test, since the set of all valid test implicitly contains all the action sequences. For this example the production amount values presented in Table 3 are used. Then Algorithm 2 is performed on every action sequence in the supertest. First we give the associated risk with the action sequence and after that the action sequence relevance percentage. The associated risks are (3, 3, 2, 2, 2, 2). The action sequence relevance percentages (calculated according to Table 3) are all approximately 83%. Thus the total coverage in the coffee machine WFTS with finite length $2 = (3 * 0.83 + 3 * 0.83 + 2 * 0.83 + 2, 0.83 + 2 * 0.83 + 2 * 0.83) = 11.62$.

This total coverage value can be used to express the relative coverage of a test suite. For example test suite T = {*(Cappuccino, Cancel)*}. The supertest becomes {*(Cappuccino), (Cappuccino, Cancel)*}. Using Algorithm 1 the absolute coverage can be calculated as $(3*0.83+2*0.83) = 4.15$. However, the absolute coverage is only useful in relation to the total coverage. The relative coverage for test suite T = $\frac{4.15}{11.62} \approx 36\%$. Thus can be concluded that test suite T is able to detect approximately 36% of the total weight of faults that could occur in the WFTS.

6.6.1 Relative coverage for syntactic definitions of coverage

We have calculated the total coverage of the coffee vending machine for different finite lengths k. Table 4 gives an overview of the coverage percentages of different syntactic coverage definitions.

According to the all-states algorithm in [7], the test suite becomes: {*(Cappuccino, Confirm, Coin, Change)*}. This results in absolute coverage ≈ 11.11.

A possible test suite for the all-transition criteria is: {*(Cappuccino, Confirm, Coin, Change, Make), (Cappuccino, Confirm, Card), (Espresso, Cancel)*}. This results in absolute coverage ≈ 26.38.

A possible test suite for the transition-pairs criteria is: {*(Cappuccino, Confirm, Card, Make, Cappuccino), (Cappuccino, Confirm, Coin, Change, Make), (Cappuccino, Cancel), (Espresso, Confirm, Card, Make, Espresso), (Espresso Confirm, Coin, Change, Make), (Espresso, Cancel)*}. This results in absolute coverage ≈ 66.66.

Table 4 shows that the relative coverage for test suites statisfying syntactic coverage criteria is surprisingly low. This implies that test case selection can be improved.

Table 4. Relative coverage percentages, for different syntactic coverage definitions, and different finite test lengths.

	$k = 5$ $TC = 128$	$k = 6$ $TC = 237$
All states	9%	5%
All transitions	21%	11%
All transition pairs	52%	28%

7. DISCUSSION

Although our proposed method is intuitive, some aspects are worth discussing. We currently calculate the action sequence relevance percentages according to concrete production information. However, this information might not always be available. Another way of calculating the percentages is using probabilistic feature diagrams [5]. Using probabilistic feature diagrams and the feature expression resulting from the action sequence, the percentage of products able to execute the sequence can be calculated without knowledge about exact production amounts. Another problem is the calibration of the finite test length. It is difficult to decide what test length is adequate. Another possible issue can be seen when examining Table 4. Even though the calculations are done for relatively low test length values, the relative coverage of syntactic coverage definitions is extremely low. This is due to the *cancel* action in the coffee vending machine WFTS. The set of all possible actions contains a lot of repetition, with a slightly different action sequence. As a consequence, these test cases weight heavily in the total coverage, even though there is a lot of repetition. Therefore future work might require a discount factor. By introducing a discount factor, repetitive test cases will weight less heavily in the total coverage.

8. CONCLUSION

In this paper we defined test coverage for software product lines in a semantic manner. A literature study on contemporary test coverage criteria was conducted. Most of the examined coverage criteria are syntactic, consequently these criteria give insufficient information about the actual tested behavior. We define test coverage according to two criteria: the risk of a fault and the percentage of products in which this fault may occur. We propose to use weighted featured transition systems (WFTS), since they model both the behavior, with associated risks, and the products that are able to execute the behavior. We introduced methods for calculating the total, absolute and relative coverage in a WFTS. The methods and algorithms introduced in this paper have been tested on a small example and the relative coverage percentages of test suites generated by contemporary coverage criteria have been calculated. The percentages show, that there is room for improvement in the generation and prioritization of test suites. Future work should aim at verifying and using this semantic definition of coverage. We plan to implement the algorithms into commonly used model checkers. By doing this, examples can be tested more efficiently, further verifying the usefulness of this research. By using our semantic definition, test coverage can be expressed in terms of weights that relate to the behavior and variability in a software product line. We believe that in future work, these weights can be used to generate and prioritize tests according to the behavior of a software product line, resulting in *more meaningful tests.*

9. REFERENCES

[1] D. Batory. Feature models, grammars, and propositional formulas. pages 7–20, 2005.

[2] L. Briones, E. Brinksma, and M. Stoelinga. A semantic framework for test coverage. *Lecture Notes in Computer Science*, 4218:399–414.

[3] H. Cichos, S. Oster, M. Lochau, and A. Schürr. Model-based coverage-driven test suite generation for software product lines. pages 425–439, 2011.

[4] P. C. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[5] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 22–31. IEEE, 2008.

[6] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-Y. Schobbens, and P. Heymans. *Coverage Criteria for Behavioural Testing of Software Product Lines*, pages 336–350. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[7] X. Devroey, G. Perrouin, and P.-Y. Schobbens. Abstract test case generation for behavioural testing of software product lines. pages 86–93, 2014.

[8] J. Gleick. Little bug, big bang, 1996.

[9] R. Lachmann, S. Lity, S. Lischke, S. Beddig, S. Schulze, and I. Schaefer. Delta-oriented test case prioritization for integration testing of software product lines. pages 81–90, 2015.

[10] D. Reuling, J. Bürdek, S. Rotärmel, M. Lochau, and U. Kelter. Fault-based product-line testing: effective sample generation based on feature-diagram mutation. pages 131–140, 2015.

[11] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration testing of software product lines using compositional symbolic execution. *Lecture Notes in Computer Science*, 7212:270–284, 2011.

[12] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 191–200, Washington, DC, USA, 2011. IEEE Computer Society.