Systematic Operational Profile Development for Software Components

Rakesh Shukla, David Carrington and Paul Strooper
School of Information Technology and Electrical Engineering,
The University of Queensland, St. Lucia 4072, Australia.
{shukla, davec, pstroop} @itee.uq.edu.au

Abstract

An operational profile is a quantification of the expected use of a system. Determining an operational profile for software is a crucial and difficult part of software reliability assessment in general and it can be even more difficult for software components. This paper presents a systematic method for deriving an operational profile for software components. The method uses both actual usage data and intended usage assumptions to derive a usage structure, usage distribution and characteristics of parameters (including relationships between parameters). A usage structure represents the flow and interaction of operation calls. Statecharts are used to model the usage structures. A usage distribution represents probabilities of the operations. The method is illustrated on two Java classes but can be applied to any software component that is accessed through an Application Program Interface (API).

1. Introduction

Software-based systems are some of the most complex artefacts ever produced. There is a growing trend to build complex software by integrating software components. As a result, the assurance of quality and especially reliability of software components has become an issue of critical concern. The *reliability* of a software component is a probability prediction for failure-free execution of the component.

The reliability of a software component depends on how the component is used. If different users use the component in different ways, they are likely to encounter different errors in the component, if there are any. Statistical testing of a component generates an estimate of the reliability of the component based on some model of its usage, which we call an *operational profile*. Specifically, an *operational profile* is a quantification of the expected use of a software component consisting of a set of input operations and their associated probabilities of occurrence for each possible state of the component. The test cases that are executed during a reliability test are a sample from the operational profile. The measure of reliability obtained in this way depends upon the operational profile. Determining an accurate operational profile for software is difficult in general [1] and it is very difficult for many software components because it requires anticipating the future use of the component [2] .

Poore et al. [3] define a two-phase construction process for an operational profile using Markov chain models. The structural phase establishes the states and transitions of the model, and the statistical phase assigns the transition probabilities. They define three approaches to the statistical phase: (1) uninformed- used when no information is available about future usage of the software; (2) informed- used when some actual usage sequences (usage data) are available from a prototype or a prior version of the software; and (3) intended usage assumptions- used when the sequences are obtained by hypothesizing runs of the software by a careful and reasonable user. Whittaker and Poore [4] state that “the informed approach with known sequences for one or more classes of users is best”. We build on this work and develop a method for developing operational profiles of software components based on both usage data and intended usage assumptions.

Our contributions are to derive a usage structure from both usage data and intended usage assumptions and use of statecharts [5, 6] to model this structure. The *usage structure* represents the flow and interaction of operation calls. We use statecharts because of their ability to define highly structured models with hierarchy and parallelism, and their popularity means that tool support is widely available.

Another contribution is that we describe a method for deriving constraints on, and relationships between, operation parameters, so that we can generate suitable values for input parameters during testing. Most of the research on operational profiles has focused on operations and little is said about the operation parameters. For software components, we have found that assigning appropriate values for input parameters can be quite complicated, because they depend not only on the type of the parameter, but there are often also constraints on individual parameters and intricate relationships between different input parameters (to the same or even different calls) and between output parameters of calls and input parameters of subsequent calls. In this paper, we describe a method for deriving those constraints and relationships as part of the operational profile.

This paper presents a systematic method that provides a step-by-step procedure for developing operational profiles for software components. The method uses both intended usage assumptions and usage data to discover a usage structure, usage distribution and characteristics of parameters. The usage distribution defines the probabilities of the operations. The method contains four steps: (1) information gathering; (2) structure modelling; (3) usage quantification; and (4) parameter analysis. The method develops a usage structure using guidelines from the information gathered in the first step, and then models the usage structure using statecharts [5, 6]. The usage quantification process produces a usage distribution. The parameter analysis process defines parameter characteristics. To demonstrate the method, we apply it to the SymbolTable and Tree components of the PGMGEN testing tool [7].

The method is intended to be part of a more general framework for the reliability assessment of software components [8]. The framework includes support for test case execution and output evaluation, which are not discussed further.

Section 2 describes our use of statecharts. Section 3 defines our method for operational profile development. Sections 4 and 5 describe the results of applying the method to the symbol table and tree case studies. Section 6 describes related work and Section 7 presents conclusions and future work.

2. Probabilistic Statecharts

Component usage is event-driven and reactive. State-based formalisms are especially useful in modelling event-driven and reactive systems. A problem with finite state machines and other flat, directed graphs is the description of the inherent complexity of the real world. As the size of the problem increases, state transition diagrams become unmanageable, resulting in unstructured and chaotic diagrams. To overcome this state explosion problem, we use statecharts, which are convenient and sufficiently powerful for describing component usage. Statecharts constitute a visual formalism for describing states and transitions in a modular fashion, enabling clustering and 'zoom' capabilities for moving easily back and forth between levels of abstraction. Following [5], we extend statecharts by adding probabilities.

Figure 1 shows an example of a probabilistic statechart. The statechart relates events and states. A change of state caused by an event is called a transition. A state is drawn as a rounded box containing an optional name. An initial state is shown by a solid circle and a final state is shown by a bull's-eye. A transition is drawn as an arrow from the source state to the target state. The transition can be labelled with Event Condition Action (ECA) rules, to which we have added probabilities. The syntax of a rule is E[C, P]/A, where E is an event, C denotes a guard condition, P represents probability of occurrence, and A represents action(s). These are all optional. An event represents an operation call. A guard condition is a boolean function that must be satisfied to enable an associated transition. A probability represents the probability that a transition whose guard is true takes place. An action is an executable atomic computation that results in a change in the state of the model or the return of a value.

To simplify modelling and reasoning with these probabilistic statecharts, we place two restrictions on them. First, we do not allow overlapping guard conditions between different transitions from a single state. For example, the conditions on the two transitions from State 1 in Figure 1 are c=n and 0<=c<n. Overlapping guards can always be eliminated by a simple transformation. Second, the probabilities of all the transitions with the same guard from a single state must add up to 1.0.

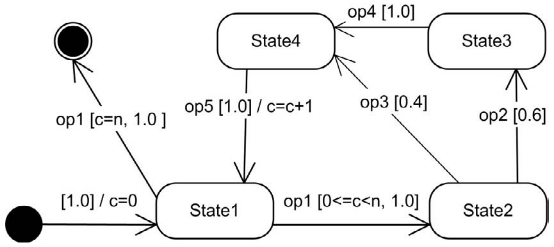

Figure 1: A probabilistic statechart

The context of the statechart in Figure 1 consists of the counter variable c and the constant n. The transition from the initial state to State1 also results in the action that assigns 0 to c. While in State1, the transition event op1 occurs conditionally ($0 \le c < n$) and the state changes to State2. While in State2 the probability of occurrence of op2 is 0.6 and op3 is 0.4. The action c=c+1 on the transition from State4 to State1 updates the context. When the counter c is equal to n, an op1 event occurs and the state changes to the final state.

3. Method Overview

Figure 2 shows an overview of the method to develop an operational profile for a software component from the intended usage assumptions and/or usage data. The rectangles represent processes and ovals indicate inputs/outputs.

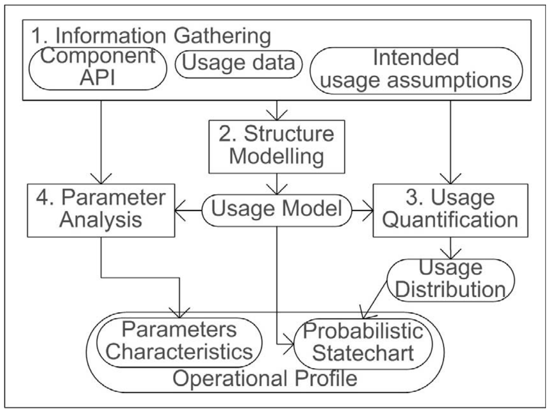

Figure 2: Method overview

The method contains four steps: (1) information gathering; (2) structure modelling; (3) usage quantification; and (4) parameter analysis. The steps are described further below.

3.1 Information Gathering

The information gathering process gathers and generates inputs for our method by collecting information about the component and its use. The component under reliability test can be either an in-house or a third-party product. However, we consider the component as a black box for operational profile development, and assume the component will be accessed through an API. The specification of a component is therefore the specification of its interface. Expected use of the component is described in the form of intended usage assumptions and actual use of the component is described by usage data.

A *component API* defines how a component is accessed by other components. A component API has a (class/component) name and a collection of operation signatures. An operation signature consists of the name of the operation and a number of inputs and outputs. Exceptions are treated as outputs. Java and CORBA interfaces are examples of APIs.

Intended usage assumptions are quantitative and qualitative assumptions about the expected use of a component. The intended usage can be described in natural language and/or using diagrams.

The actual usage data consists of traces from execution of a previous version of the component, a prototype, an actual implementation, or a simulation. A *trace* is an ordered sequence of operation invocations, including both inputs and outputs. A *complete trace* is produced by a complete run of a system/application in which the component is used. When a system/application does not terminate, there are no complete, finite traces, and we must consider finite initial traces. We assume that any usage data that we collect represents a correct behaviour of the component.

3.2 Structure Modelling

The structure modelling process shown in Figure 3 consists of two steps: structure analysis and modelling.

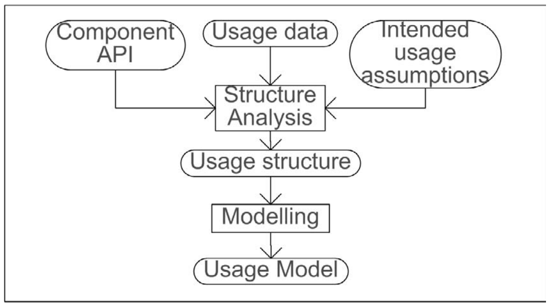

Figure 3: Structure modelling

The component's API, usage data and intended usage assumptions are used for structure modelling. The component's API defines the syntax and semantics of the component operations. Structure analysis extracts a usage structure that represents the flow and interaction of operation calls, ignoring the operation parameters. Modelling provides a visual representation of the usage structure using statecharts.

We propose two approaches to derive a usage structure: top-down and bottom-up. The top-down approach can be applied when patterns such as repeated sequences of operations can be identified in the usage data. When this is not the case, the bottom-up provides a more suitable approach. In complex cases, it may be possible to apply a top-down approach first, followed by a bottom-up approach (see Tree example in Section 5).

The top-down approach consists of the following steps.

1. Examine the operations in the traces and divide them into groups of repeating patterns of operations.
2. Identify relationships between the identified patterns, both within sequences of repetition and across such sequences.
3. Identify context variables and predicates on these variables that capture these relationships.

Once these steps have been performed, the translation into a statechart is a straightforward process. The initial state is typically the class constructor or a similar initialisation. The states and transitions are determined by the patterns derived. Finally, the context variables and predicates associated with the transitions are specified on guard conditions and on actions as context update functions to build a usage model.

With the bottom-up approach, the statechart is constructed more directly. The bottom-up approach consists of the following steps.

1. Add a state for each operation that appears in the usage data. This state represents the last call that was made.
2. Add a separate initial state. For each initial operation call c_1 appearing in a trace in the usage data, add a transition from the initial state to the state representing that operation.
3. For each subsequent operation call c_i appearing in a trace in the usage data, add a transition from the state representing the operation call immediately preceding c_i to the state representing operation c_i.

3.3 Usage Quantification

Usage quantification adds probabilities into our statecharts. The usage quantification process shown in Figure 4 involves: (1) transition analysis; (2) frequency calculation; and (3) probability calculation or estimation. Usage quantification is not always necessary. A usage model is *deterministic*, if for every source state s, there is exactly one output transition for each guard condition c (recall from Section 2 that we do not allow overlapping guard conditions on output transitions). If the usage model is deterministic, there is no need for the usage quantification process; each transition in the probabilistic statechart will have a probability of 1.

When the usage model is nondeterministic, that is, if there is a source state s and a guard condition c, for which there is more than one output transition, then usage quantification is necessary. For this, either the usage data can be used to calculate probabilities following the process outlined in [3, 9, 10], or the intended usage assumptions must be used to estimate these probabilities. If available, we prefer to use usage data for accurate usage distribution.

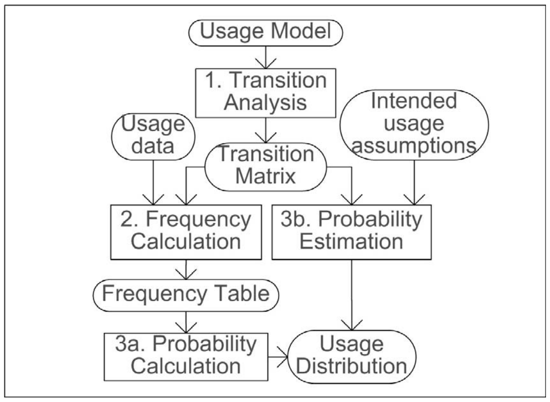

Figure 4: Usage quantification

Following [3, 9, 10], we first construct a transition matrix that records all possible transitions with guard conditions between states. Deterministic transitions have a probability of 1, impossible transitions have a probability of 0, and the nondeterministic transitions are the ones for which we have to calculate or estimate the probabilities. To calculate the probabilities from the usage data, we first count frequencies of all the transitions in the transition matrix and then convert these to probabilities using a straightforward calculation. For example, if there are two possible transitions from state s1 to state s2 with the same guard condition c, and the first one is taken 3 times and the second one 7 times, then the probabilities for the two transitions are 0.3 and 0.7 respectively.

A significance test [11] can be applied to the frequency table when usage data is used. The significance test produces a significance level, the probability that a given result could not have occurred by chance. If the significance level is not "satisfactory" more usage data must be collected or the probabilities must be refined with intended usage assumptions. When intended usage assumptions are used instead of usage data, an entropy computation (following [3]) on the usage distribution can be used to measure the degree of disorder or randomness in the usage model.

3.4 Parameter Analysis

Appropriate values for input parameters depend not only on their types, but often also on additional constraints and relationships between them. During parameter analysis, the usage data and intended usage assumptions are used to model the parameter types, constraints and relationships so that suitable values for input parameters can be determined during test case generation.

Parameter analysis consists of three steps.

1. Identify the type of the parameters from the API.
2. Analyse the usage data to look for constraints on individual parameters and relationships between parameters. Here the usage model derived during the structure modelling is often useful to distinguish between different occurrences of the same operation. Intended usage assumptions can also be used to suggest constraints and relationships, or to provide a justification for the existence of these in the data. Of course, if no usage data is available, only intended usage assumptions can be used to define any constraints or relationships.
3. Model the constraints and relationships so that they can be used for test case generation. We have used textual descriptions for this.

Note that in some cases, some of the identified relationships may force us to reconsider the structure we have come up with in the structure analysis, although we have found in practice that the separation of structure and parameter analysis is a useful separation of concerns.

4. Case Study 1: Symbol Table

We apply our method to PGMGEN's SymbolTable component. PGMGEN [7] is a testing tool that generates batch test drivers from test scripts. A test script contains a set of test cases and each test case contains a description of the expected behaviour, including any expected exceptions.

The purpose of this and the next case study is to derive operational profiles that reflect how the SymbolTable and Tree components are used in PGMGEN. Because the use of usage data represents the more interesting and challenging aspects of the method, we try to use this usage data as much as possible in the case studies. In more realistic case studies, such usage data may not be available, especially when devising an operational profile for new components or existing components in new applications, and one would have to rely more on intended usage assumptions.

4.1 Information Gathering

The API of SymbolTable is shown in Figure 5. The table stores pairs of symbols (strings) and identifiers (integers). Symbols and identifiers must be unique. The constant `MAX_SYMBOLS` indicates that a maximum of 50 symbols are allowed in the table and `MAX_SYM_LENGTH` indicates that the maximum length of a symbol is 20. The operation `insert(sym)` adds a new symbol `sym` and assigns an identifier to it. `size` returns the number of pairs in the table. The function `existId(id)` returns whether identifier `id` is in the table. Similarly `existSym(sym)` returns whether symbol `sym` occurs in the table. The operation `del(id)` deletes identifier `id` and its corresponding symbol from the table. The functions `getSym` and `getId` return the symbol and identifier for a given identifier and symbol respectively. The Java exception handling mechanism is used to signal exceptions in the implementation. The operation `insert(sym)` throws `MaxLengthExc` if `sym` has more than `MAX_SYM_LEN` characters, `FullExc` if the table has `MAX_SYMBOLS` symbols in it, and `ExistSymExc` if `sym` already exists in the table. The operations `del(id)` and `getSym(id)` throw `NotExistIdExc` if there is no identifier `id` in the table. The operation `getId(sym)` throws `NotExistSymExc` if `sym` is not in the table.

```
public class SymbolTable {
    static final int MAX_SYMBOLS = 50;
    static final int MAX_SYM_LENGTH = 20;
    public SymbolTable();
    public void insert(String sym) throws
      MaxLengthExc, FullExc, ExistSymExc;
    public int size();
    public boolean existId(int id);
    public boolean existSym(String sym);
    public void del(int id) throws
      NotExistIdExc;
    public String getSym(int id) throws
      NotExistIdExc;
    public int getId(String sym) throws
      NotExistSymExc;
}
```

Figure 5: API for SymbolTable

The intended usage assumptions are based on knowledge about how SymbolTable is used in the PGMGEN system. SymbolTable first stores exception names as symbols. The supplied exception names must be unique. Then PGMGEN uses the list of exception names to generate exception handler code in the test driver.

To gather actual usage data about how SymbolTable is used, we instrumented the SymbolTable implementation and ran PGMGEN on five test scripts. Figure 6 shows one complete trace obtained from SymbolTable in this way. The operations are separated by commas. Each operation contains the name of the operation, input parameter values in brackets and output values after a colon (:). For the trace in Figure 6, two symbols are added to the table (“empty” and “full”), and these symbols are assigned the identifiers 0 and 1 respectively, as can be seen from the calls to `getId` in the trace. We use the special symbol “termination” to indicate termination of the system, so that we can easily distinguish complete traces from incomplete ones.

```
SymbolTable(), size():0, existSym(“empty”):false, insert(“empty”),
size():1, existSym(“full”):false, insert(“full”), size():2, size():2,
size():2, size():2, getSym(0):”empty”, size():2, getSym(1):”full”,
size():2, size():2, size():2, getSym(0):”empty”, size():2,
getSym(1):”full”, size():2, size():2, existSym(“empty”):true,
getId(“empty”):0, existId(0):true, existId(0):true,
existSym(“empty”):true, getId(“empty”):0, existId(0):true,
existId(0):true, existSym(“full”):true, getId(“full”):1, existId(1):true,
existId(1):true, termination
```

Figure 6: A complete trace

While the five traces we collected are sufficient to come up with an operational profile, these traces were all obtained by analysing the use of PGMGEN on existing, textbook scripts. As such, they may not be representative of the actual usage of PGMGEN and hence the usage of SymbolTable within PGMGEN. However, collecting such actual usage data is beyond the scope of this paper. Instead, we assume that the usage data we have in the five traces is representative and use the method to derive a suitable operational profile from that usage data.

4.2 Structure modelling

We are able to identify patterns in the usage data and therefore apply the top-down approach. The first step of the top-down approach is to examine the operations (ignoring operation parameters) in the traces and divide them into groups. The process of subdividing is typically an iterative process, where by investigating one or more traces, particular patterns are identified, which are then confirmed (or rejected) by investigating other traces in the usage data.

As expected, the first operation in each trace is the class constructor and it occurs only once in each trace, so the first sequence identified is S1 = <SymbolTable>. We then recognise a pattern of repeated calls to size, existSym, and insert, which is confirmed by investigating the other traces. We record the pattern as a repeated occurrence of the sequence S2 = <size, existSym, insert>. Note that we could also have recorded it as the occurrence of a single call to size (S2' = <size>) and then a repeated occurrence of the sequence S3' = <existSym, insert, size>. The other traces do not provide further information about which of the two is more appropriate, so we assume the first, slightly simpler version is sufficient (until we find data to the contrary). Continuing with this, we identify the following other sequences of interest: S3 = <size>; S4 = <getSym, size>; S5 = <existSym, getId, existId, existId> and S6 = <termination>.

The second step is to identify relationships within and across the sequences. Sequence S1 is always first in a trace followed by the sequence S2. S2 contains size, existSym, and insert. Each existSym in S2 returns false in the usage data. From the intended usage assumptions, we deduce that this sequence of calls is probably used to add exception names that appear in the test script to SymbolTable, and that the check to existSym is there to ensure that the same exception name is not added multiple times. The reason that the call always returns false in our usage data is that we have only run PGMGEN on correct test scripts, which do not contain duplicate exception names. As indicated above, this may not reflect how PGMGEN would actually be used in practice, but in line with the assumption that the usage data we have is representative, we mimic this behaviour in our operational profile. Thus, S2 appears as many times as there are exception names defined in the PGMGEN script. After that, S3 appears four times. The next sequence S4 appears the same number of times as S2 appears and then S3 appears twice. S4 appears again the same number of times and then S3 once. From the intended usage assumptions, we deduce that the sequence S5 appears once for each test case in the PGMGEN script with an exception in it. Finally, S6 appears once. Thus the sequences appear in the order S1, S2, S3, S4, S3, S4, S3, S5, S6, where S2 and S4 are repeated n1 times, once for each exception name defined in the PGMGEN script, and S5 is repeated n2 times, once for each test case in the PGMGEN script with an exception in it.

The third step is to capture the above relationships using context variables and predicates. We use two constants to control the iterations of S2, S4, and S5:

- n1 - represents the number of exceptions in the script and is used to control the number of iterations of S2 and S4; and
- n2 - represents the number of test cases with exceptions in the script and is used to control the number of iterations of S5.

We also need a number of context variables that count the number of iterations as we traverse the model.

Figure 7 shows part of the resulting statechart. It contains seven composite states, where the composite state `create table` represents iterations of S2, `get symbols` represent iterations of S4, `access table` represents iterations of S5, and the states `Size1`, `Size2`, and `Size3` represent different repetitions of S3. The bottom part of Figure 7 shows the details for the composite state `create table`, which models n1 iterations of the sequence S2=<size, existSym, insert>.

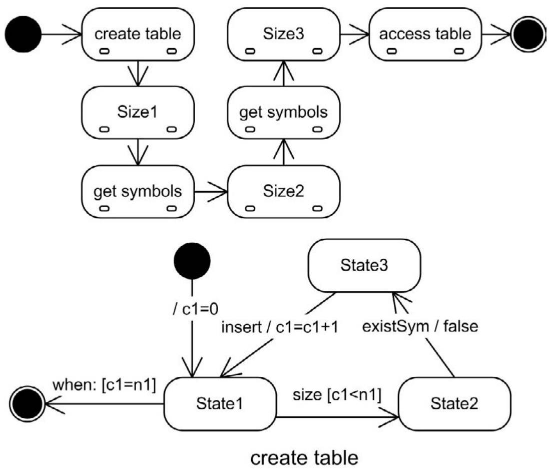

Figure 7: Usage model

4.3 Usage Quantification

The statechart derived for SymbolTable is deterministic because all decisions have been incorporated into context variables and constants based on intended usage assumptions. As a result, usage quantification is not necessary.

To generate test cases from the statechart we still need to select suitable values for the constants n1 and n2. However, since we know what these constants represent, we can use this knowledge to select these values from a suitable range.

4.4 Parameter Analysis

The first step is to identify the types of the parameters from the API (Figure 5). The operations `insert`, `existSym`, and `getId` have a `String` input parameter representing a symbol. The operations `existId`, `del`, and `getSym` have an integer input parameter representing an identifier. We also note the output parameters: booleans for `existId` and `existSym`, a symbol (`String`) for `getSym`, and an identfier (`int`) for `getId`. Although we do not need to assign values to output parameters when generating test inputs from our operational profile, there are sometimes relationships between output parameters of calls and input paramaters of subsequent calls that we should capture in our operational profile.

We then analyse each sequence identified during the structure modelling and consider the ones for which input parameter values must be determined. In this case, these sequences are S2, S4, and S5.

For the sequence S2, the two instances in the trace (Figure 6) are:

size():0, existSym(“empty”):false, insert(“empty”)
size():1, existSym(“full”):false, insert(“full”)

From the intended usage assumptions we have deduced this is where SymbolTable is loaded. From the usage data, it is clear that the input parameter `sym` to `existSym` and `insert` in each occurrence of S2 is the same in both calls. As discussed in Section 4.2, we have also deduced that each of the symbols is unique. We must therefore ensure that the same is true for the test sequences that we generate.

For S4, both instances of repetitions of S4 are identical and contain two occurrences of S4:

getSym(0):”empty”, size():2
getSym(1):”full”, size():2

In this case, the input parameter `id` to `getSym` seems to increase from 0 to n1-1, where n1 is the number of symbols in SymbolTable, which is confirmed by the other traces.

For S5, the three instances are:

existSym(“empty”):true, getId(“empty”):0,
existId(0):true, existId(0):true,
existSym(“empty”):true, getId(“empty”):0,
existId(0):true, existId(0):true,
existSym(“full”):true, getId(“full”):1,
existId(1):true, existId(1):true

In this sequence, SymbolTable is accessed and the constraints on the input parameter `sym` to existSym and getId are that the same parameter is used in both cases and that it is an input parameter that has previously occurred in S2 (i.e., it represents a symbol that is actually stored in the table). Moreover, the input parameter `id` of both calls to `existId` is the same as the output parameter of the preceding call to `getId`.

4.5 Test Case Generation

We have implemented a Java program that generates sequences of calls to SymbolTable according to the operational profile derived above. The program is a straightforward implementation of the probabilistic statechart, with the constraints and relationships that were derived in Section 4.4 implemented to generate the input parameters for the operation calls in the sequences.

5. Case Study 2: Tree

We have also applied the method to the Tree component of PGMGEN, which is used to build an abstract syntax tree of the input script file. Tree supports a forest of trees and has operations to add new nodes to the forest, to add a sub-tree as a child of another node, and to traverse a tree. Each node has a value (the token in the input file), a type (the type of the token), and the line number on which the token occurs.

The operations `makeNode`, `setType`, `setValue`, `setLnum`, and `addChild` are used to construct a tree. `makeNode` adds a new empty node to the forest. The operations `setType`, `setValue` and `setLnum` assign a type, a value, and a line number of a given node. `addChild` adds a child node as the rightmost child of a parent node. The other operations are used for tree traversal. `childCount` returns the number of children of a given node. `getNthChild`, `getType`, `getValue` and `getLnum` return the n-th child node, the type, the value, and the line number of a given node.

The same five PGMGEN scripts as for SymbolTable were used to generate actual usage data for Tree. In this case, the traces are much longer and more complex. For example, the shortest trace consists of 2024 calls, while the longest trace consists of 13015 calls.

Although we can identify some patterns in the traces for Tree, there are some sequences of calls in the traces for which we cannot detect or explain the order in which they occur. We therefore first apply a top-down approach to deal with the patterns that we can identify, and then the bottom-up approach for the remaining calls. One pattern that we can identify is that every use of Tree consists of an initial phase in which the tree is constructed, followed by a phase in which the tree is traversed. Moreover, the tree creation phase consists of n calls to `makeNode`, followed by calls to `setType`, `setValue`, `setLnum`, and `addChild` (not necessarily in that order and in some cases repeated more than once). This is why we apply the bottom-up approach to fill in the remaining detail.

The resulting statechart is not deterministic, so we then apply usage quantification on the usage data. The resulting probabilistic statechart is shown in Figure 8, where the two composite states correspond to tree creation and tree traversal. The labels in the statechart represent the following operations: O1 - `makeNode`, O2 - `setType`, O3 - `setValue`, O4 - `setLnum`, O5 - `addChild`, O6 - `getType`; O7 - `getLnum`; O8 - `getNthChild`, O9 - `childCount`, and O10 - `getValue`.

We applied a statistical significance test [11] to the frequency table used to calculate the probabilities and the result was satisfactory.

The parameter analysis for Tree was much more complex than for SymbolTable. For example, all methods of Tree except `makeNode` have at least one parameter that must represent a pointer to a valid node that was previously added using a call to `makeNode`. If such a parameter does not represent a valid node, the method should throw the `InvalidNodeExc` exception. This means that if we randomly generate the parameters for such calls, it is very likely that all these calls will throw this exception (even if we restrict the range to valid pointer values). However, in the use of Tree in PGMGEN, no such exceptions should be thrown, indicating that there are relationships between the parameters of calls to Tree that should be incorporated into the operational profile.

Note that if we attempt to completely capture all such relationships our operational profile may become arbitrarily complex (e.g. more complex than the implementation of Tree itself). We therefore have to make a trade-off between the accuracy of the operational profile (how will it captures the use of Tree in PGMGEN) and its complexity.

The parameter analysis revealed a significant number of non-trivial relationships between the parameters. For example, during tree creation, the node parameter to all calls to `setType`, `setValue`, and `setLnum` are the output parameter of the last call to `makeNode` preceding it. Similarly, the two node parameters to `addChild` were return values to preceding calls to `makeNode`, but not always the most recent one. We incorporated many of these relationships in the operational profile. Even so, a

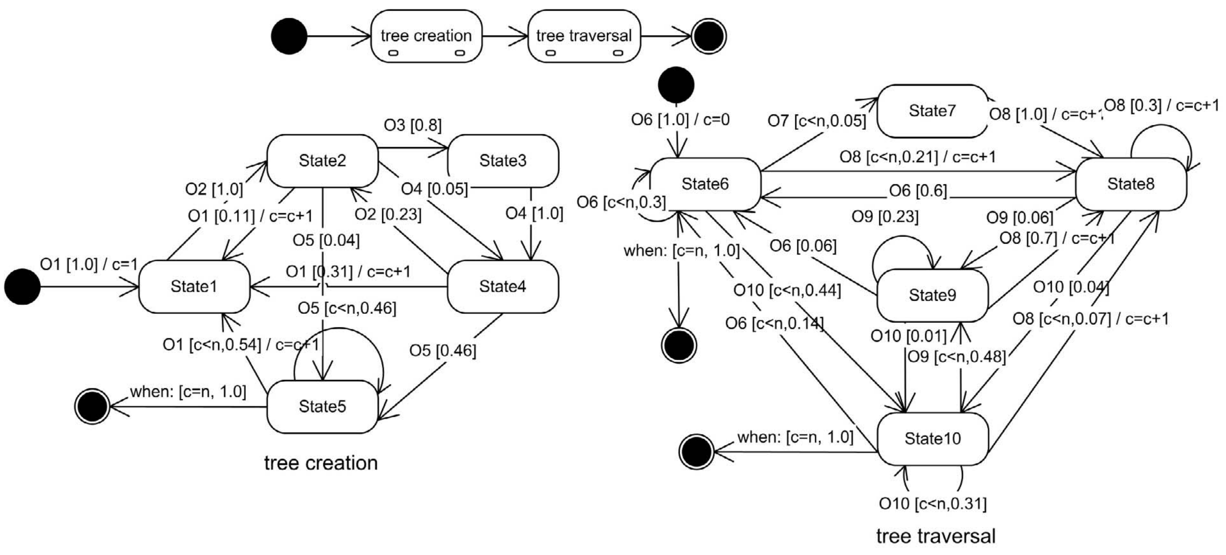

Figure 8: Probabilistic statechart

number of calls generated by the operational profile can still throw exceptions. Since the operational profile for this case study represents a simplified approximation to the usage data, our generated test cases produce some exceptions not observed in the actual usage data. Further refinement is required to minimise these exceptions.

Similar to SymbolTable, the operational profile for Tree was implemented in a Java program that generates test cases for Tree according to this profile.

6. Related Work

Our work focuses on operational profiles and the use of state machines to model these. In the literature, there are different approaches to developing operational profiles using different usage environments and models. In [12, 13], the actual software usage is described by assigning unconditional probabilities to software inputs. The usage is described using a tree, and different possible sequences of stimuli are described as paths in a tree.

In recent years, Markov models have been proposed to describe operational profiles [3, 9, 14, 15]. Whittaker and Poore [14] use Markov chain properties to define a usage model (distribution) and to generate test cases based on the model. In the Markov chain, the usage is described in terms of states and transitions. The states represent user states and user stimuli are connected to the transitions. Whittaker and Thomason [15] extend the Markov chain model by including analytical results and failure data in the testing chain.

Doerner and Gutjahr [9] extend the Markov chain model by defining the syntax and semantics of a language that allows non-Markovian transitions between states. The addition of this feature adds limited non-standard state in Markov models maintaining the Markovian property, but it requires a special-purpose language for describing non-Markovian transitions. Woit [10] describes a technique for the specification of operational profiles, using a small example of intended usage, without mentioning how to develop operational profiles. We expand on Woit's approach and provide a systematic method for deriving a usage structure and parameter characteristics.

A major problem in operational profile development for software components is in the difficulty of describing usage behaviours. The Finite State Machine (FSM) [16] provides concepts of states and state transitions for specifying behaviour. However, a complex behaviour cannot be effectively described using a FSM because of the state explosion problem related to the inherent complexity of the real world. Harel [5] introduces statecharts to extend finite state machines for complex behaviour. The UML features state machines based on Harel's statechart notation [6]. We use statecharts to model our usage structure to avoid the state explosion problem and the problem of having to introduce a new language/notation. To incorporate probabilities, we adopt the probabilistic statechart approach suggested by Harel [5].

The method described in this paper is an important part of our framework for the reliability assessment of software components [8], which also incorporates test case execution and output evaluation.

7. Conclusions

We have presented a systematic method for developing operational profiles for software components using both usage data and intended usage assumptions. After gathering information about the component and its usage, the method attempts to derive a deterministic usage model by identifying relationships within and across the identified patterns using context variables and predicates. However when a usage model contains nondeterministic choices, the choices are replaced by probabilities from the usage distribution. Parameter analysis considers constraints on, and relationships between, parameters so that suitable values for input parameters can be derived during test case generation from the operational profile.

The application of the method to two components and their usage data establishes the practical viability of the method. For structure analysis, the top-down approach has been applied to one example and a mix of the top-down and bottom-up approaches to the second. Although both components are Java classes, the method is applicable to any component that can be accessed through an API (such as CORBA, EJB, and COM+ components).

The main areas for future work are to provide tool support for the method and the more general framework [8], and to apply it to an industrial case study.

References

[1] D. Hamlet, "Are We Testing for True Reliability?," *IEEE Software*, vol. 9, pp. 21-27, 1992.

[2] J. A. Stafford and J. D. McGregor, "Issues in predicting the reliability of composed components," In Proceedings of 5th ICSE Workshop on Component-based Software Engineering (CBSE5), Orlando, Florida, USA, 2002.

[3] J. H. Poore, G. H. Walton, and J. A. Whittaker, "A Constraint-based approach to the representation of software usage models," *Information and Software Technology*, vol. 42, pp. 825-833, 2000.

[4] J. A. Whittaker and J. H. Poore, "Statistical testing for cleanroom software engineering," In Proceedings of Twenty-Fifth Hawaii International Conference on System Sciences, pp. 428 -436, 1992.

[5] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231-274, 1987.

[6] J. Rumbaugh, I. Jacobson, and G. Booch, *The unified modeling language reference manual*: Reading, Mass. : Addison-Wesley, 1999.

[7] D. M. Hoffman and P. A. Strooper, *Software Design, Automated Testing, and Maintenance A Practical Approach*: International Thomson Computer Press, 1995.

[8] R. Y. Shukla, P. A. Strooper, and D. A. Carrington, "A framework for reliability assessment of software components," In Proceedings of 7th International Symposium on Component-based Software Engineering (CBSE7), Edinburgh, UK, pp. 272-279, 2004.

[9] K. Doerner and W. J. Gutjahr, "Representation and optimization of software usage models with Non-Markovian state transitions," *Information and Software Technology*, vol. 42, pp. 873-887, 2000.

[10] D. Woit, Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules, PhD Thesis, in *Computing and Information Science*. Kingston, Ontario, Canada: Queen's University, 1994.

[11] A. J. Hayter, *Probability and statistics for engineers and scientists*. Boston: PWS Pub. Co., 1996.

[12] B. D. Juhlin, "Implementing operational profiles to measure system reliability," In Proceedings of Third International Symposium on Software Reliability Engineering, Research Triangle Park, NC, USA, pp. 286 -295, 1992.

[13] J. D. Musa, "Operational profiles in software-reliability engineering," *IEEE Software*, vol. 10, pp. 14 -32, 1993.

[14] J. A. Whittaker and J. H. Poore, "Markov analysis of software specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 2, pp. 93-106, 1993.

[15] J. A. Whittaker and M. G. Thomason, "A Markov chain model for statistical software testing," *IEEE Transactions on Software Engineering*, vol. 20, pp. 812-824, 1994.

[16] A. Gill, *Introduction to the theory of finite-state machines*. New York: McGraw-Hill, 1962.