Software Testing with an Operational Profile: OP Definition

CAROL SMIDTS, CHETAN MUTHA, MANUEL RODRÍGUEZ,
and MATTHEW J. GERBER, The Ohio State University

This article is devoted to the survey, analysis, and classification of operational profiles (OP) that characterize the type and frequency of software inputs and are used in software testing techniques. The survey follows a mixed method based on systematic maps and qualitative analysis. This article is articulated around a main dimension, that is, OP classes, which are a characterization of the OP model and the basis for generating test cases. The classes are organized as a taxonomy composed of common OP features (e.g., profiles, structure, and scenarios), software boundaries (which define the scope of the OP), OP dependencies (such as those of the code or in the field of interest), and OP development (which specifies when and how an OP is developed). To facilitate understanding of the relationships between OP classes and their elements, a meta-model was developed that can be used to support OP standardization. Many open research questions related to OP definition and development are identified based on the survey and classification.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Reliability, Statistical methods, Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; G.3 [**Probability and Statistics**]: Reliability and Life Testing

General Terms: Reliability, Verification

Additional Key Words and Phrases: Software testing, operational profile, taxonomy, usage models, software reliability

ACM Reference Format:
Carol Smidts, Chetan Mutha, Manuel Rodríguez, and Matthew J. Gerber. 2014. Software testing with an operational profile: OP definition. ACM Comput. Surv. 46, 3, Article 39 (February 2014), 39 pages.
DOI: http://dx.doi.org/10.1145/2518106

1. INTRODUCTION

OP Definition

As defined by Musa [1993], "an operational profile [(OP)] is a quantitative characterization of how a [software] system will be used." It "consists of the set of operations that a system is designed to perform and their probabilities of occurrence."

OP Importance

An OP helps derive reliability estimates by testing a software system as if it was in the field. An OP increases productivity, increases reliability, and shortens development time. As stated by Musa [1993], "Using an operational profile to guide testing ensures that if testing is terminated and the software is shipped because of imperative schedule constraints, the most-used operations will have received the most testing and the reliability level will be the maximum that is practically achievable for the given test time."

Authors' address: C. Smidts, C. Mutha, M. Rodríguez, and M. J. Gerber, 201 W. 19th Ave, Columbus, OH-43210.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 0360-0300/2014/02-ART39 $15.00
DOI: http://dx.doi.org/10.1145/2518106

Testing with an OP

Testing with an OP generally involves a number of steps. These include (1) development of the OP model (including model analysis and validation), (2) derivation of OP probabilities, (3) generation of test cases, (4) test planning (including the design of pass, fail, and stop criteria), (5) test execution, and (6) analysis of results. An OP model consists of a representation of how the software system will be used. Representations based on tree-like models, Markov chains, and statecharts are commonly used. The core of an OP model typically consists of a number of events that represent the use of the software system (such as mouse clicks or data input, as well as customer/user actors, configuration alternatives, and software functions) together with their associated occurrence probabilities. These probabilities can be determined a number of different ways, generally by employing techniques built on the analysis of historical data, experimental data, or expert judgment.[1] OP models are commonly evaluated and validated with respect to field use assumptions and test constraints. Test cases can be generated with a higher or lower degree of automation from the OP models by employing different techniques (random walk models, analytical formulae, etc.) and strategies (with respect to probability levels, test cost, model coverage, etc.).

Need for this Survey

As stated by Juhlin [1992], "the foundation of Software Reliability Engineering (SRE) is Operational Profiles. No matter how sophisticated the metric and modeling techniques used, they're only as good as the data that serves as input to them. That data, in turn, is only as good as the methods used to generate it. For SRE, those are the methods that define and implement the Operational Profile."

This survey is necessary because no related survey on OP testing currently exists in the literature. Furthermore, authors and practitioners define and apply OP in ways that differ to various extents. How they are defined and applied depends on multiple aspects such as the application domain (safety-critical systems, Internet systems), the modeling language (tree-like models, Markov chains, finite state machines), the scope (consideration of the OS, the hardware, or other software running on the same computer), the number of profiles (customer, functional), the determination of probabilities (use of historical or experimental data), and the employed testing approach (model-based, partition). Because of these differences, the usefulness of testing with an OP is best assessed by considering the multiple facets and perspectives. The survey, analysis, and classification provided in this paper is oriented toward this end.

Organization of this Article

The topic of OP testing is vast but involves two distinct phases: (1) OP development and (2) OP-based testing. In this article, we address the characteristics of OP development and their classification and relationships. Related papers that focus on these topics are discussed in detail.

The organization and topics covered in this article are as follows: Section 2 describes the methods used to identify research papers related to the definition of the OP model, generate the OP classes, and classify the papers found. Section 3—the bulk of the article—describes and discusses the OP classes. A meta-model outlining the relationships between the classes of the OP model is also provided in Section 4 and fully

[1]The incorporation of expert judgment in software reliability is crucially important to software developers and researchers. It involves the elicitation and aggregation of the opinions of experts on relevant topics; e.g., how an attribute may affect the reliability of a piece of software or the impact on reliability of a module or the number of lines of code. Campodónico and Singpurwalla [1995] present a variation of the Musa-Okumoto process using expected number of software failures as expert knowledge.

Table I. Results Obtained through the Inclusion and Exclusion Process

Search Term	**IEEE Explore**	**Citeseer**	**Science Direct**	**Google Scholar**	**Unique**
Abstract or Title	Title or Abstract	Title or Abstract Include citation	Title or Abstract	Title	
"operational profile"	171	141	34	156	388
"Usage model" software	38	53	15	27	86
"usage profile" software	16	18	8	7	38
"usage distribution" software	3	3	0	1	7
"input distribution" software	7	5	3	0	13
"input profile" software	2	3	2	2	6
"user profile" software	24	54	19	8	89
"usage testing" software	7	8	4	7	19
"operational distribution" software	4	25	0	0	27
"profile of use" software	0	43	1	0	40
"usage chain" software	0	0	1	6	6
Totals		**915**			**542**
		Elimination Round 1			**350**
		Elimination Round 2			**84**
		Elimination Round 3			**55**
		Elimination Round 4			**19**

described in Appendix B. Section 5 provides a discussion of open research questions as they relate to the OP classes and the OP model definition. Finally, Section 6 concludes the article.

2. REVIEW AND CLASSIFICATION METHOD

This section describes the mixed method used to identify research papers related to OP definition, generate classes, and classify the papers found. The method builds on the systematic mapping methods described or used in Petersen et al. [2008], Jørgensen and Shepperd [2007], and Kitchenham [2004] and on qualitative analysis (more specifically grounded theory) [Charmaz 2006; Corbin and Strauss 2008; Glaser and Strauss 2009]. Each such method has been modified to fit our purpose.

Inclusion and Exclusion Criteria

Inclusion and exclusion criteria and process are discussed in this section.

Inclusion

A paper was included in our survey if it described OP research. "Operational profile" and semantically equivalent search terms were used to select relevant papers. The results are given in Table I. Note that "software" was added to further narrow down the search. The search focused on the title and/or the abstract of the paper as the search engines permitted. A total of 915 papers that satisfied the criteria were found. The search was performed on July 24, 2013 in the following online databases: IEEE Explore, Google Scholar, Citeseer, and ScienceDirect. The search includes all papers published by that date. The set of 915 papers was reduced by eliminating duplicates; 542 unique papers resulted.

Exclusion

Papers were excluded through four rounds of elimination. The criteria used in each round of elimination are discussed in the following text, and the results are given in Table I.

Elimination Round 1: The downselection of papers considered the following criteria. Papers, books, and thesis were excluded if:

1. They were not published in English.
2. There was no paragraph on OP.
3. The paper, book, or thesis was strictly less than four pages.
4. A paper existed on a similar topic.
5. The paper, book, or thesis could not be retrieved using IEEE Explore, CiteSeer, Google, Electronic Journal Center, or ACM Digital Library.

After the first round of elimination, 350 papers remained (see Table I). These papers were then downloaded for further elimination based on a detailed analysis.

Elimination Round 2: A second round of exclusion to identify OP papers with treatment of OP definition issues was undertaken. In this round, we used Atlas.ti [Muhr and Friese 2004], a qualitative research analysis tool, to perform a thorough search. The search formula is given in the following text, and the search was performed only on the title and abstract.

$$Search := express*|refine*|generate*|exten*|defin*|develop*|creat*|special*|determin*|deriv*|procedure|design*|construct*|approach|specific*|formulate*$$

The search terms are semantically equivalent forms of "definition," where "∗" indicates a wild character and " | " indicates "OR." Seventy-nine papers resulted from this search. The papers excluded by Atlas.ti were manually evaluated to identify "false rejects," that is, papers that included a section on OP definition but were excluded by the Atlas.ti search tool. Five false rejects were identified. This second round of exclusion resulted in 84 papers. After this round of exclusion, no thesis or books remained.

Elimination Round 3: A third round of elimination was then conducted to identify papers with significant OP definition content. The exclusion criterion was papers with an OP definition section strictly less than two columns. In addition, papers less than or equal to two pages were rejected. This round of exclusion narrowed the number of papers down to 55 "primary" References.

Elimination Round 4: A fourth round of elimination was based on subjective evaluation and expert judgment. Out of the set of 55 papers, 16 papers were selected. During this subjective evaluation, papers whose OP representation was similar to other authors' (such as Musa, Whittaker, Runeson, etc.) were eliminated. The 41 papers eliminated provide insights into usage probability calculation and usage data collection. However, these topics are not the focus of this article. In addition, while reviewing the 55 papers, two papers were discovered (both by Whittaker) that did not appear in our initial search. These papers were recovered from a Google search because they were not located in our databases. The Huang et al. [2007] paper did not appear in any of the searches or reviews. However, it was known to be directly relevant and was added to the pool of papers. These three new papers were added to the "primary" list.

The complete list of paper sources is given in Table VXI in Appendix A. The distribution of 19 papers published per year is shown in Figure 1.

Research Questions

The papers were then analyzed to respond to the following questions:

1. What are the main distinguishing characteristics of the operational profiles?
   1.1. What is the scope of the operational profile?
   1.2. What are the typical structures of the operational profile?

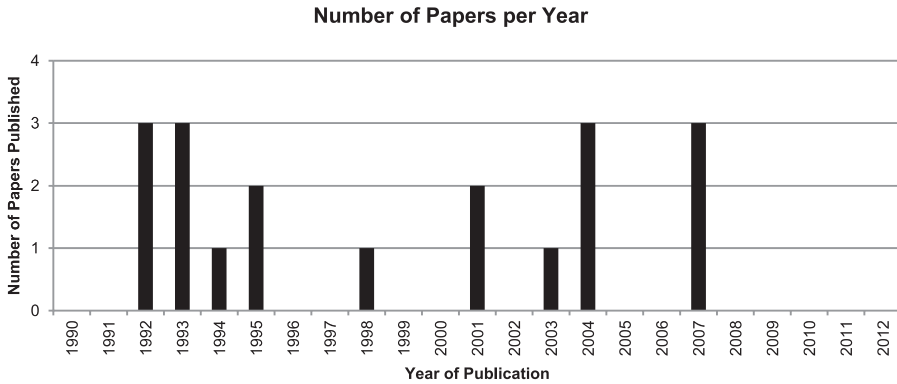

Fig. 1. Number of papers published per year.

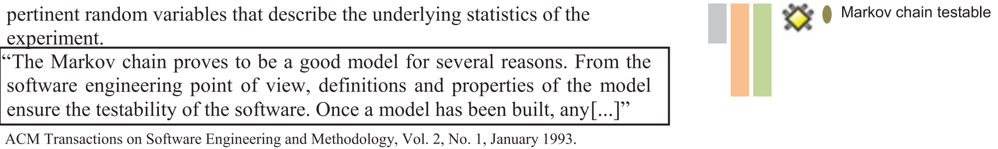

Fig. 2. Coded segment of text extracted from Whittaker and Poore [1993].

1.3. When in the lifecycle phase is the operational profile typically developed?
1.4. Is the operational profile typically developed for the system as a whole?
1.5. What are other operational profile characteristics?

Papers with a major focus on the research questions were identified and further refined into classes.

Class Identification Process

This section describes the process followed for developing OP classes. The process is based on the qualitative analysis [Saldana 2012] of segments of text belonging to the 19 downselected papers (given in Table V). The qualitative analysis reviews papers and attempts to identify salient features and repeating patterns. To do so, specific sections of text are evaluated by an analyst. The sections of text considered include the title, the abstract, the introduction, and OP definition section for the various papers. These sections were selected because they contain the motivation of the paper and set the stage for the research discussed in the paper. In addition, focusing on these portions of text allows us to restrict the analysis and save time and effort. These sections were coded by closely following the text (i.e., using mostly in vivo coding[2]). For example, Figure 2 provides a coded segment of text extracted from Whittaker and Poore [1993]. The coding was performed using Atlas.ti, a qualitative research analysis tool. Having obtained these low-level codes, higher-level categories were formed based on recurrence in the codes (see Table II). Next, the high-level categories were filtered to retain categories of immediate relevance to our research questions (see Table III). In addition, we examined the interrelationships between classes using meta-modeling. This helped further clarify the definitions of each class and make the classes as independent as possible. This step is similar to the notion of axial coding in grounded theory.

[2]In vivo coding is the practice of assigning code to a data segment using the words in that segment. In vivo coding captures the key ideas of the researcher in their own terms.

Table II. Example of Low-Level Codes and Corresponding High-Level Categories from Whittaker and Poore [1993]

Codes (Low-Level)	Categories (High-Level)
"Arcs of Markov chain determines sequences" "Arcs of Markov chain ordering of events" "Ergotic nature of Markov chain- describe underlying statistic" "Finite state discrete parameter Markov chain to model usage" "Finite state, discrete parameter" "Markov chain good model" "Markov chain testable" "Markov chain to conduct statistical testing" "Markov chain tractable stochastic process" "Markov chain useful for defining probability system" "Markov from spec as basis of OP and test generation for statistical testing Rich body of theory/algorithm/" "Sates of Markov Chain entries from input domain" "Standard analytical results of Markov chains have important interpretations" "Statistically typical test cases can be obtained from Markov Chain"	*Markov Chain*

Table III. High-Level Categories Formed from Coding Sections of Whittaker and Poore [1993], Gittens et al. [2004], and Huang et al. [2007]

Paper	Categories (High-Level)	Filtered Categories	OP Classes
Whittaker & Poore [1993]	Cleanroom		
	Markov Chain	*Markov Chain*	*Structure*
	OP		
	Reliability Certification		
	Statistical Testing		
	Sequence of Events	Sequence of Events	Scenario
	Specification	Specification	Early
	Usage Model		
Gittens et al. [2004]	Configuration	Configuration	Configuration
	Current OP		
	Extend OP Definition		
	External Input Data	External Input Data	Input Data
	New Profile	New Profile	Profile
	OP Deficiencies		
	OP Extension Properties		
	OP Limited		
Huang et al. [2007]	Deficiencies		
	Develop New Method		
	Extend Fault Injection		
	Fault Injection		
	Fault Injection Profile	Fault Injection Profile	Profile
	Hardware Failure	Hardware Failure	Executive Scope
	Physics of Failure		
	Representative Fault Models		
	Software Reliability Prediction		
	System Type	System Type	Field-of-Interest

The meta-model is provided in Appendix B. The class identification process is depicted in Figure 3. The process is more complex because it involves constant comparison to already established codes, categories, classes, and meta-models and may require revision of already attributed categories and meta-models. The process is iterative and was repeated for all 19 papers. In addition, if classes or models are still changing, the

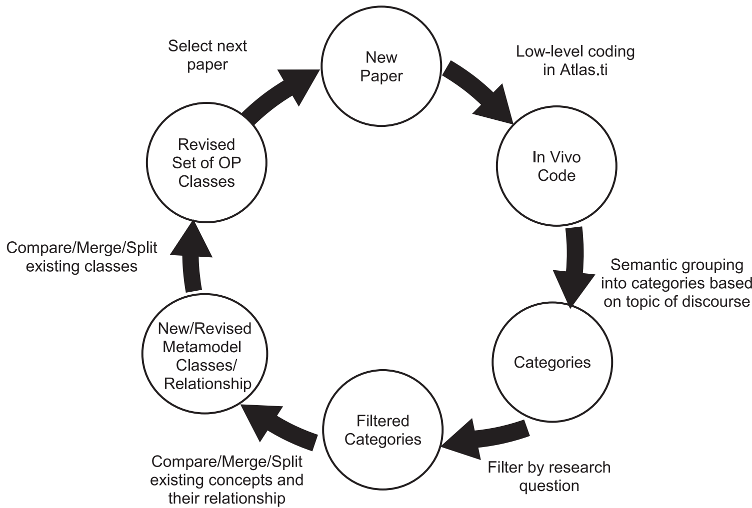

Fig. 3. Extraction of OP classes.

Table IV. Example Keywords Used to Operationalize the Classification Process

Class	Keywords	Comments/Insights
Profile	*customer profile, user profile, configuration profile, system-mode profile, data profile, process profile*	If the OP contains one profile, it was classified as *single-profile*; else, *multiple-profile*. The concept of multiple profiles led us to investigate different types of inputs and their providers. Because of the varied nature of providers, we created the *originator* class.
Structure	*Markov chain, state machine, statechart, set, tree, hierarchy, flow graph, graph, model*	The structures found with these keywords resulted in two important insights. First, *probability of occurrence* is a property of the structure and is not a distinct class. Second, *computability*, which accounts for usage dynamics, is a property of the modelling technique used. Upon realizing these dependencies, we removed these two classes from our original classification. Also, some unconventional and hybrid (state-based plus tree) structures were discovered. After analyzing these papers' context, we formed the class *field-of-interest*. We found that the structure and inputs of an OP are modified based on fields of interest.

process is repeated. The analysis is documented in the tool and can be reviewed by an independent reviewer if and as necessary.

Classification Procedure

In this step, the 19 papers selected are classified with respect to the OP classes developed. Using the in vivo codes, the category names and class names identified in the previous subsection, we define a series of keywords to establish a systematic classification process. These keywords are given in Table IV and Table XII in Appendix A. Classification was facilitated by reading only the section of each paper that described the OP. Because an OP could be hybrid, subclasses were not always considered mutually exclusive.

To determine the extent to which the classification of papers is repeatable, we also evaluated the inter-rater agreement achieved by two consecutive sets of analysts using both Cohen Kappa [Sim and Wright 2005] and AC1 [Gwet 2002] statistics. The analysis shows high inter-rater agreement for each of the two measures (Cohen Kappa coefficient and AC1) for 9 classes out of 11, medium agreement for 1 class and low

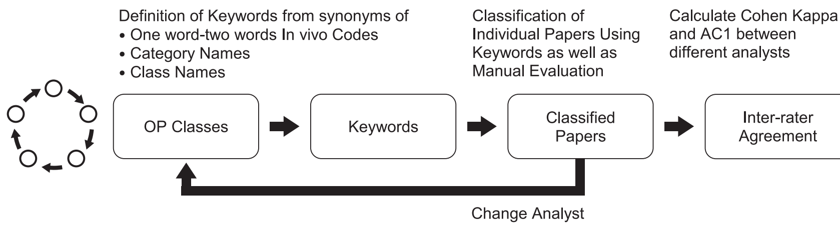

Fig. 4. Classification process for individual papers.

agreement for 1 class. The two latest results stem from ongoing fluctuations in the definitions of the classes at the time of first classification. The inter-rater agreement therefore demonstrates sufficient stability in the classification.

The process of classification is described in Figure 4.

Selection and Classification Process Quality

We should concern ourselves of the quality of the process defined because that will determine the quality of the outcomes of the study. Aspects to consider are validity, reliability, and generalizability [Gibbs and Lewins 2005]. According to Gibbs and Lewins [2005], validity can be ensured using four different approaches (triangulation, auditability/audit trail, use of negative cases, and constant comparison). Out of these, auditability (i.e., providing enough detail for the approach to be auditable) and constant comparison (i.e., checking code and their consistent interpretation throughout the sample of papers) are the techniques used in this article, as illustrated in the earlier discussion. For ensuring reliability, Gibbs and Lewins [2005] suggest the use of notions of interrater agreement and of member validation. Member validation would consist in asking the authors of the 19 papers selected to review the classes and meta-models defined in this article and the classification of their papers with respect to the OP classes created. Member validation is difficult in practice and may lead to biases; therefore, it was not pursued. In this article, we ensure reliability through interrater agreement of classification of papers into OP classes. Generalizability is difficult to address because the sample was not drawn randomly. The sample was selected through a systematic exclusion process, and all remaining papers were included. In addition, papers were included if they were considered valuable to the study. This is in alignment with the theoretical sampling principles of grounded theory where the sample is chosen according to the needs of the study (i.e., the classes that form from the analysis of the data). In our context, generalizability would involve the extent to which the knowledge generated can be extended to other papers that were not part of the search but nonetheless could exist in other databases. To ensure generalizability in qualitative analysis, it is recommended that a description of the population from which the findings were derived be defined. The findings would be generalizable to a similar yet unknown population. In our case, these include the base papers found through keyword searches described in Table XII. Base papers are papers that significantly increase the knowledge related to OP definition.

3. OPERATIONAL PROFILE CLASSES

A software OP is a probabilistic characterization of software usage—probabilistic because the exact usage prediction of software is impossible (infeasible/impractical) (see Figure 5). As such, usage modeling is essential and use of structures evident. A structure enables the modeling of the desired usage and also provides a computational framework. As such, the structure is the backbone of the usage modeling.

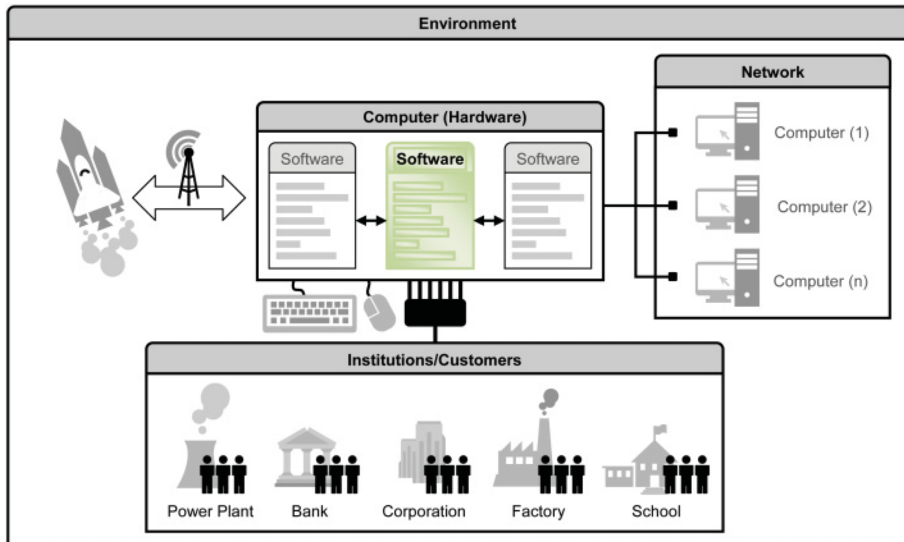

Fig. 5. Software operating in an environment.

A software system is inherently an interactive system that communicates with different types of users such as humans, and other hardware or software systems to fulfill its purpose. The diversity of the environment warrants specific profiles to capture particular interactions. The software boundary is a natural separation between the environment and the software system. Elements outside the boundary are a part of the environment, and elements within the boundary are a part of the software.

Musa [1993] proposes a five-step OP development approach in which he progressively decomposes the entire system (software and its environment) and builds multiple profiles such as “the Customer profile,” “the User profile,” “the System-mode profile,” “the Functional profile,” and “the Operational profile.” The software usage is primarily modeled using a tree-based structure, where each tree node corresponds to elements of the particular profile. Each node is assigned a probability of occurrence. He also characterizes the input data of the operational profile and its originator (e.g., human).

Whittaker and Poore [1993] aim to analyze the software specification before the design and coding phase. One of the fundamental steps in this analysis is the construction of a Markov chain (state-based) model to define the software usage. The model could be built for a system-mode or the entire system. He does not provide any guidelines or rules on how to model configuration changes. The model need not explicitly consider the sequences of input to model the transitions. If the input sequence is not available, transitions with uniform probability distribution accounts for the state change during software usage. Undoubtedly, if the input sequences are available (maybe through direct observation of user behavior), more accurate usage and usage probabilities can be obtained. Whittaker does not explicitly characterize the inputs, nor specifies the type of originator.

The previous discussion leads us to a classification scheme shown in Figure 6. The *Common Features* correspond to distinguishing characteristics of an OP. The classes under *common features* are independent of each other. The *Software Boundary* corresponds to a partitioning of the elements into those pertaining to the software and those that do not. The classes within the *system boundary* are strongly related to some of the classes in *common features*. For example, *Input Data* is related to *Originator*.

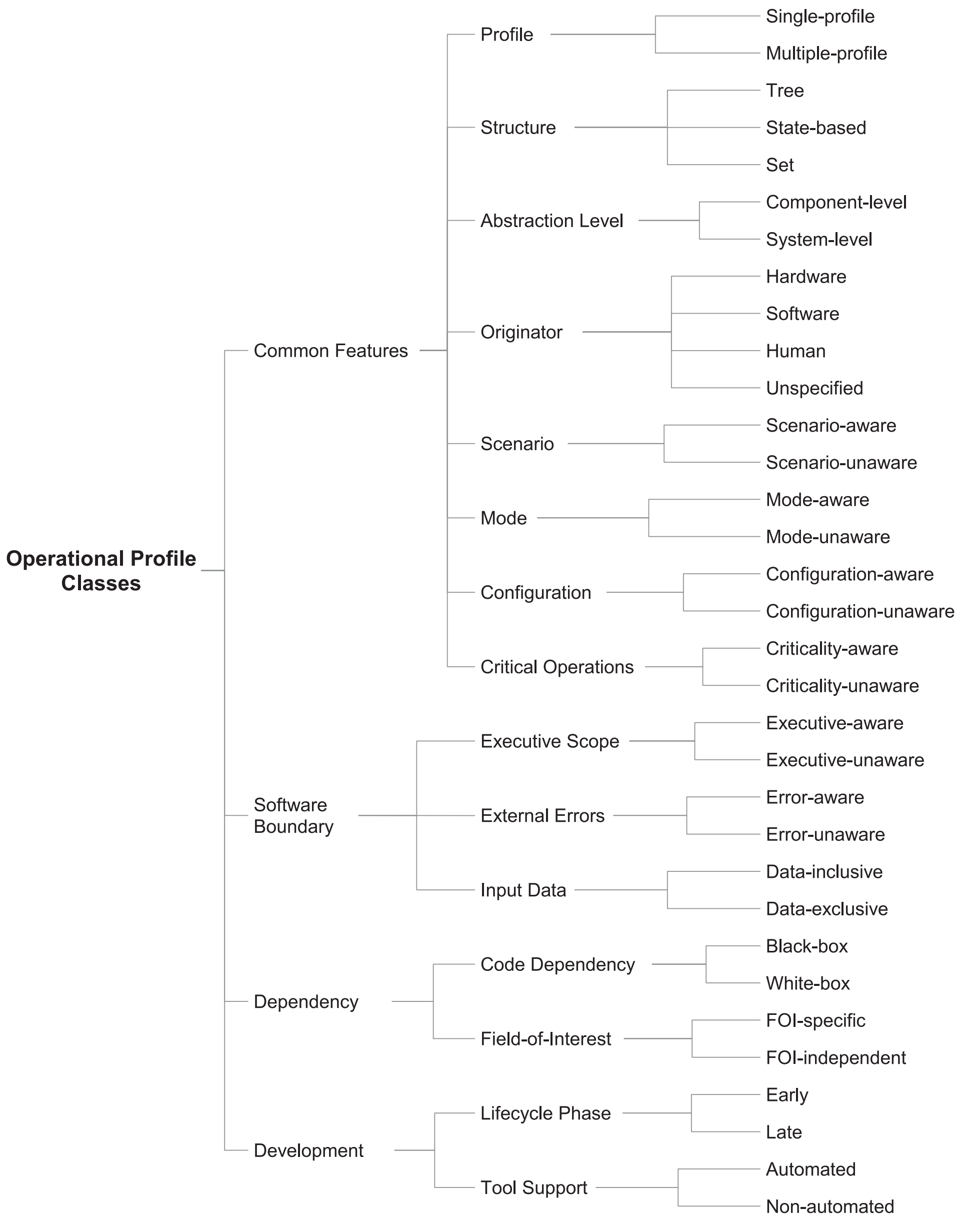

Fig. 6. Operational profile classes.

The *Common Feature* and *Software Boundary* classes were created from the fundamental understanding of the OPs. The *Dependency* class includes classes that account for additional, available information and can help significantly improve the accuracy of the OP profile. Dependency may lead to modification of the common features of the basic OP. This class was created based on information discovered during the survey. The classes under *Development* mostly address the question of when and how an OP is developed.

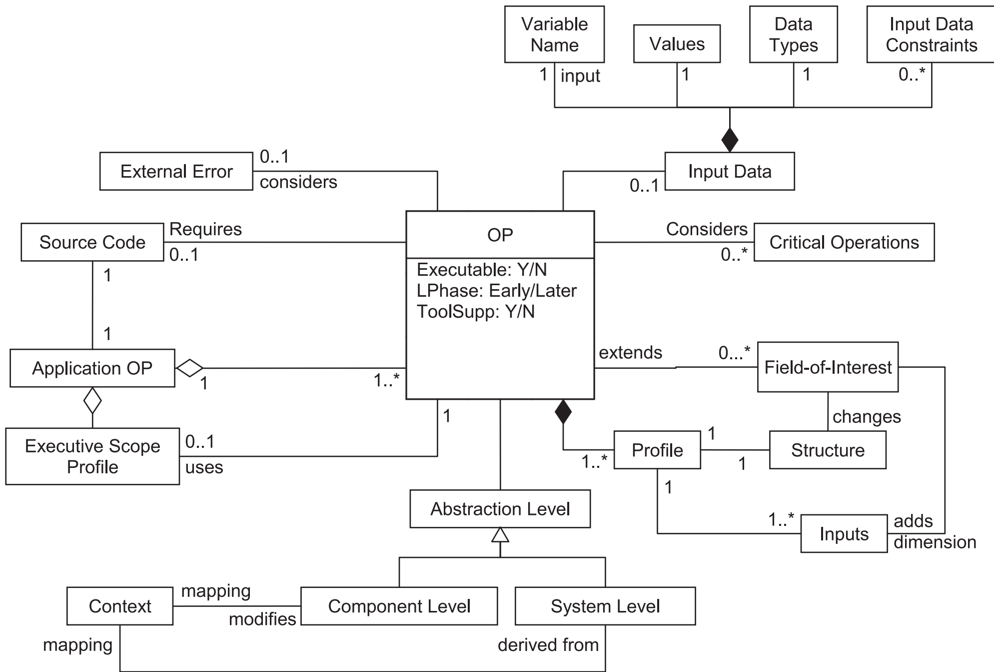

Fig. 7. Operational profile meta-model.

The relationship between different classes and their elements is completely developed in the meta-model shown in Figure 7 and the meta-models shown in Figures 8 through 14 in Appendix B.

Figure 6 shows an OP taxonomy that consists of a hierarchy of classes and subclasses that focus on different perspectives and aspects of OP. Each is discussed in the following sections.

3.1. Common Features Classes

This section describes the *Common Features* class of OP. The *Common Features* class is divided into the classes *Profile*, *Structure*, *Abstraction Level*, *Originator*, *Scenario*, *Mode*, *Configuration*, and *Critical Operations*. Refer to Table V for a classification of the papers reviewed with respect to these classes

3.1.1. Profile Class. The profile is a cross-sectional usage view of the application along a given dimension. It includes the elements of that dimension and the occurrence probabilities of these elements during use. An example set of possible profiles is presented in Table VI.

It is worth noting that the profiles provided in Table VI might be defined and applied in different ways by authors and practitioners because of different interpretations of the notions of function, operation, process, environment, and configuration.

An OP may contain one or several profiles. Accordingly, it can be classified as a *single-profile OP* or a *multiple-profile OP*. The profiles of a multiple-profile OP commonly exhibit dependency relationships. For example, a typical multiple-profile OP can be defined as [Musa 1993]:

{*customer profile, user profile, system-mode profile, functional profile*, *operational profile*}

Table V. Classification of Papers Reviewed

	Common Features																			Software Boundary						Dependency				Development			
	Profile		Structure			Abs. Level		Originator				Scenario		Mode		Config.		Critical Operations		Input Data		Executive Scope		External Errors		Code Depend.		Domain Depend.		Lifecycle Phase		Tool Support	
	Single	Multiple	Tree	State-based	Set	Component-level	System-level	Hardware	Software	Human	Unspecified	Scenario-aware	Scenario-unaware	Mode-aware	Mode-unaware	Configuration-aware	Configuration-unaware	Criticality-aware	Criticality-unaware	Data-inclusive	Data-exclusive	Executive-aware	Executive-unaware	Error-aware	Error-unaware	Black-box	White-box	Domain-specific	Domain-independent	Early	Late	Automated	Non-automated
Bousquet et al. 1998	X		X	X			X				X		X	X			X		X	X			X		X	X		X		X		X	
Brooks and Memon 2007	X			X			X			X		X			X		X		X		X		X		X	X		X		X		X	
Elbaum and Narla 2001	X		X				X			X			X		X		X		X		X		X		X	X			X	X		X	
Gittens et al. 2004		X			X		X	X	X	X			X	X		X			X	X			X		X		X		X		X	X	
Guen and Thelin 2003, Guen et al. 2004	X			X			X			X		X			X		X		X				X		X	X			X	X		X	
Hamlet et al. 2001	X				X	X					X		X		X		X		X	X			X		X	X			X	X			X
Huang et al. 2007	X				X		X	X					X		X	X		X			X	X			X	X			X		X	X	
Juhlin 1992		X					X			X		X			X	X			X		X	X			X	X			X	X		X	
Musa 1992, Musa 1993		X	X				X			X			X	X		X		X		X			X		X	X			X	X			X
Ouabdesselam 1995	X		X				X				X		X		X		X		X		X		X		X	X		X		X			X
Popovic and Kovacevic 2007	X			X			X				X		X		X		X		X		X		X	X		X			X	X		X	
Runeson and Wohlin 1993		X	X	X			X			X			X		X		X		X		X		X		X	X			X				
Shukla et al. 2004a	X			X		X					X	X			X		X		X	X			X		X		X		X	X			X
Walton et al. 1995		X		X			X	X	X	X			X		X	X		X			X	X		X		X			X	X			X
Whittaker and Poore 1993	X			X			X				X		X		X		X		X		X		X		X	X			X	X		X	
Whittaker and Thomason 1994	X			X			X				X		X	X			X		X		X		X		X	X			X	X		X	
Woit 1993	X				X	X					X	X		X			X		X	X			X		X	X			X	X			X
Total	12	5	5	9	4	3	14	3	2	8	8	5	12	5	12	5	12	3	14	6	10	3	14	2	15	15	2	3	14	14	2	10	6
Cohen Kappa	**1**					**1**						**0.204**						**1**		**0.70**		**0.63**		**1**		**0.63**		**0.45**		**0.63**		**0.81**	
AC1	**1**					**1**						**0.12**						**1**		**0.69**		**0.90**		**1**		**0.90**		**0.62**		**0.90**		**0.87**	

Table VI. Example Set of Possible Profiles

Profile Dimension	Definition	Example(s)
Customer	Frequency-of-use probabilities of a software product by different customer types, where a customer type would correspond to a set of entities or organizations *acquiring* the software product and using it in a similar way	Frequency-of-use of an operating system in the aerospace industry vs. the automotive industry
User	Frequency-of-use probabilities of a software system by different user types, where a user type would correspond to a set of entities or organizations *employing* the software system in a similar way	Expert vs. untrained users
System-mode	Occurrence probabilities of the various execution modes of a software system	Initialization, administration, or online modes
Environmental	Occurrence probabilities of different environmental factors affecting the usage of a software system	Probability that a flight software system is used in a radioactive environment such as deep space vs. a low-radiation environment such as Earth
Functional	Occurrence probabilities of the several functions of a software system	Functions of adding or removing a user account in a server system
Operational	Occurrence probabilities of the various types of tasks or operations accomplished by a software system	Adding required information of a recently hired employee into a server system
Usage	Occurrence probabilities of states and transitions in a graph model representing the software system's use	Markov chains, statecharts, etc.
Configuration	Occurrence probabilities of physical and logical configurations of a software system	Use of a RISC or CISC hardware architecture to run a software system
Structural	Occurrence probabilities of quantifiable data structure attributes of a software system	Number of loops, size of arrays, etc.
Data	Occurrence probabilities of the different inputs of a software system	Data values, ranges, types, etc.
Process	Occurrence probabilities of the processes of a software system	Same as operational profile if a process is viewed as an operation

in which hierarchical relationships and mapping relations exist (e.g., a hierarchical relationship exists between customer profile and user profile). For example, let us consider a payroll application. Two distinct customer groups would include private organizations and government organizations. Within the "government organization" customer group, one could distinguish the following user groups: human resources group, employees group, manger group, and a system administration group.

Other relationships between profiles include the mapping relations found between system-mode and functions. For example, a maintenance mode would map to start-up functions, shut-down functions, and reporting functions.

The profile meta-model shown in Figure 9 in Appendix B describes the dependencies between profile and other constructs that are shown in Figure 6 and discussed throughout Section 3.

3.1.2. Structure Class. Several modeling paradigms are used to describe and build an OP. They include trees [Arora et al. 2005; Bousquet et al. 1998; Elbaum and Narla 2001; Musa 1993], state-based representations (including Markov chains [Guen et al. 2004; Popovic and Kovacevic 2007; Prowell 2003; Whittaker and Thomason 1994], probabilistic event flow graphs [Brooks and Memon 2007], Harel statecharts [Shukla et al. 2004a]), and sets [Gittens et al. 2004; Hamlet et al. 2001; Huang et al. 2007,

Table VII. Example Set of Possible Structures

Author	Structure Type	Elements used and Their Interpretations
Bousquet et al. [1998]	Markov and Tree (a tree is a Markov chain with nodes that are trees)	A binary decision diagram (a type of tree) is used to define an OP. The tree node is a tuple of the input variable state and a list of conditional probabilities for that input state. The outgoing tree branches from nodes are the values taken by the variable. The output of the tree is an array of the input space represented as a set of binary equations.
Whittaker and Thomason [1994]	Markov Chain	A state is an "externally visible mode of operation that must be maintained in order to predict the application of all system inputs." The transition represents an input element and its probability of occurrence.
Gittens et al. [2004]	Sets	Gittens et al. [2004] define multiple profiles. Each profile is expressed as a set. For example, a structural profile includes a measurable quantity, its values, and its frequency of occurrence.

2011; Woit 1993]. Examples of the possible structure are given in Table VII. A sample representation of these structures and their constituent elements is shown Figures 10, 13, and 14 in Appendix B.

Probability of Occurrence

The probability of occurrence of the elements (nodes, transitions, inputs, events, etc.) found in the definition of profile varies conditionally with other elements in a manner that is complex to describe and is inherently embedded in the structures we have just described. As an example of a straightforward usage case, the probability of occurrence of a leaf node in a tree may be conditional on N other predecessor leaf nodes in the tree [Musa 1993]. As another straightforward usage example the OP is 1-conditional for Markov chains (where the next state of the OP model only depends on the current state [Guen et al. 2004; Popovic and Kovacevic 2007; Shukla et al. 2004a; Whittaker and Thomason 1994]) or unconditional [Elbaum and Narla 2001; Gittens et al. 2004; Hamlet et al. 2001; Juhlin 1992]. Brooks and Memon [2007] define OPs that are N-conditional, where the element of interest is the input to the GUI application and where dependence lies between N successive inputs. However, N-conditional probabilities between different inputs are tricky and difficult to represent into commonly available structures. To express such N-conditional probabilities, hybrid or new structures are warranted. For example, Bousquet et al. [1998] embed a binary decision diagram, a tree, into the Markov structure to model specific software usage. On the other hand, Woit [1993] used a set-based approach and grouping of prefixes of various lengths with similar impact on the selection of the next input into sets. Runeson and Wohlin [1993] express N-conditionality between leaf nodes (each representing a profile in a profile hierarchy) and 1-conditionality using a Markov chain to represent transitions within a leaf node (modeling microbehaviors within a profile).

Computability

An OP that relies on an executable model is referred to as *executable*; otherwise, it is referred to as *nonexecutable*. Examples of executable models are those based on Markov chains, Harel statecharts, probabilistic event flow graphs, and so on. In general, the ability of an OP model to be executed can potentially help improve the computability of various steps of the OP testing process, bringing important benefits on aspects such as model verification and validation, determination and computation of probabilities, derivation of test cases, computation of quality metrics of test cases, execution of test

cases on the software under test, development of runtime oracles for passed/failed test criteria, and implementation of stopping test criteria.

Note that an OP approach can be executable but nonautomated. For example, Shukla et al. [2004a] and Woit [1993] followed executable approaches for the definition and development of the OP; however, their papers mention that tool support/development will be the object of future work. The reverse is also possible, that is, an OP approach can be nonexecutable but automated (i.e., supported by tools). Indeed, although an executable OP might intuitively be considered as more powerful than a nonexecutable OP, it is also necessary to take into account other aspects such as the maturity and characteristics of the modeling technique chosen (e.g., pros and cons, available features, applicability, descriptive power) or the availability and type of automated tools (see Section 3.4.3). For example, a static OP methodology that relies on the use of standard UML models can potentially access a large number and variety of automated tools and techniques for different purposes that may significantly benefit the entire OP testing process.

3.1.3. Abstraction Level Class. The concept of OP has commonly been related to system-level testing because reliability measurements are traditionally most beneficial when targeted to a software system viewed as a *product*. Today, a large number of software products are primarily developed to be used (or *reused*) as *components* within other products and systems. For example, this is the case of many COTS (Commercial Off-The-Shelf) software such as operating systems. OP testing thus becomes an attractive solution to measure the reliability of these components. An OP aimed to test software components is referred to as a *component-level OP.* A major challenge of applying OP testing to a software component is that, compared to a system, the number of operational environments that the component may encounter in the field is larger, more diverse, more complex, and more difficult to model and foresee a priori. Likewise, initial operational testing of a component is infeasible because of the abundance of possible end users and usage patterns. Because component developers may lack information on the final application environment, a component will be designed and developed for a hypothesized application environment, which may differ from the actual one. Because of this, the component's use becomes context-dependent and must be tested in the environment in which it will be used. The Ariane 5 disaster cited by Weyuker [1998] (in which a CNES launch vehicle exploded because of an insufficiently tested software component reused from a previous vehicle) demonstrates that *not* testing components in their new implementation can lead to substantial and possibly disastrous failures. Weyuker stresses that considerable testing of components in their new environment is necessary to ensure proper behavior and adequate reliability.

That said, few works have addressed OP testing of software components [Hamlet et al. 2001; Shukla et al. 2004a; Woit 1993]. The OP specification technique for software modules proposed by Woit [1993] is built upon the assumption that unlike other software such as *batch* programs, the OP of *modules* cannot be accurately described using unconditional probabilities. It is claimed that an OP model based on a Markov chain would be more suitable but still limited because probabilities are conditioned on the previous event only. Accordingly, in the proposed OP specification technique, the probability of generating a module input can be dependent on any previous input. The technique defines custom syntax and semantics and is built on concepts such as *execution prefix* (subsequence of a module execution[3]) or *events* (change of an object's

[3]"An execution prefix is any subsequence of a module execution ... $_.E_1.E_2. \dots E_{ti}$, where "_" refers to module initialization or reinitialization, E_j is the jth event issued after this module initialization (or reinitialization), and E_{ti}, is the last event issued before the next module re-initialization ($1 \leq j \leq t_i$.)" [Woit 1993].

state) among others. An unbounded integer *stack* is used as an example. The technique also includes a procedure that automatically generates test cases from the OP.

Hamlet et al. [2001] points out the fact that when a "component is embedded in a system," the component profile depends on system input profile, "components position in the system, and the actions of other components." He suggests that development of component-level OPs should be based on two main ideas: (1) *profile mapping,* a component datasheet should specify "mappings from a [system] profile to reliability parameters," and (2) *component subdomains,* partition the component's "input space into functional sub-domains" and assigns relative weights to each sub-domain and within a sub-domain a uniform distribution probability.

Shukla et al. [2004a] propose an approach to construct the OP of software components in the form of *Harel statecharts*. A component is viewed as a piece of code with well-defined operations in an Application Programming Interface (API) such as classes and objects of the Object-Oriented Programming methodology. The statechart of a component is built by analyzing usage data about its execution flow as (a sequence of) executed operations. Usage data includes both available execution traces and intended assumptions ("[operation] sequences obtained by hypothesizing runs of the software by a careful and reasonable user"). The statechart probabilities are calculated by analyzing transitions and frequencies of operations in the usage data. Guidelines are also provided to model constraints and relationships of operation parameters, which are intended to facilitate the generation of suitable values for input parameters during testing.

3.1.4. Originator Class. The OP literature distinguishes different types of input providers called *originators* in this article. For example, in a section of the literature, well represented by Musa's research, the human factor is considered as an important dimension and is explicitly taken into consideration commonly in the form of customer and user profiles included in the OP. Attention to this type of originator emerged in the information technology community when it focused on testing information technology applications in industrial environments (e.g., Arora et al. [2005], Elbaum and Narla [2001], Gittens et al. [2004], Juhlin [1992], and Musa [1993]). Another cross-section of the literature does not distinguish explicitly between user inputs and other types of system inputs (software, hardware, environmental, etc.), but these are commonly implicit in the models used (e.g., Markov chains) (see, e.g., Bousquet et al. [1998], Brooks and Memon [2007], Hamlet et al. [2001], Guen et al. [2004], Huang et al. [2007], Prowell [2003], Shukla et al. [2004a], Popovic and Kovacevic [2007], Whittaker and Thomason [1994], and Woit [1993]). An interesting observation is the fact that in component-based OPs, the originator is unspecified (see Table V). Indeed, one a priori does not know from whom the component will receive its inputs. The inputs to the component depend on its position in the software architecture that may vary from one use of the component to another. A third group of papers explicitly distinguishes originators of type hardware, human, and software [Gittens et al. 2004; Walton et al. 1995]. A fourth group identifies an originator of type hardware [Huang et al. 2007]. Therefore, the originator class is divided into *hardware*, *human*, *software*, or *unspecified* (with the understanding that multiple types of originators may co-exist). The originator meta-model is shown in Figure 11 in Appendix B.

3.1.5. Scenario Class. Scenario is a sequence of user or externally generated inputs (mostly sequences of events) that accomplish a specific application goal. For example, a user wanting to open a Word document in Windows would first select the "File" menu option, then select the "Open" option, and so on. This sequence of steps forms a scenario.

An OP built using explicit scenario information is labeled as *scenario-aware*; otherwise, it is termed as *scenario-unaware.*

Examples of scenario-aware OP approaches follow. Shukla et al. [2004a] develop the structure of the OP using actual usage data, which is in the form of traces, that is, sequences of operations' executions. Woit [1993] uses execution prefixes, that is, subsequences of a module execution that are composed of events and start with module initialization or reinitialization, to specify the OP. Brooks and Memon [2007] use observed sequences of events to build the probabilistic event flow graph. In Guen et al. [2004], UML sequence diagrams can be used to construct the OP. Yan et al. [2004], Zang et al. [2011], and Ai et al. [2012] use UML annotated use case and sequence diagrams to derive a Markov chain usage models.

On the other hand, OPs constructed with Markov chains, state machines, and statecharts encode implicit sequence information (if the models were not directly derived from sequence information). Prowell [2003] is an example of such implicit consideration of scenarios and, as such, is classified as scenario-unaware.

3.1.6. Mode Class. A *mode* corresponds to a particular type of execution of the software system, such as administrator mode, startup mode, safe mode, shutdown mode, and so on. A mode can be "a set of functions or operations that are grouped for convenience in analyzing execution behavior" [Musa 1993]. Examples include user group (administration vs. maintenance), user experience (novice vs. expert), and so on.

A mode can also be a state or a set of states [Bousquet et al. 1998; Whittaker and Thomason 1994].

A *state* of a software system corresponds to a specific set of values that a certain group of (software) variables or conditions of interest can take during operation. Examples of states include overload, nominal, and failure. For example, an operating system usually contains a variable (or a set of variables) specifying the maximum number of threads, tasks, or processes that can be created and executed at any time depending on the capacity and performance of the CPU. Different states can be defined if such a limit is reached (*overload* state) or exceeded (*failure* state), while an average number of processes could be considered as the *nominal* state.

The modes form the system-mode profile [Arora et al. 2005; Musa 1993] or the process-mode profile [Gittens et al. 2004]. An OP could be *mode-aware* if it explicitly considers modes or otherwise *mode-unaware*.

3.1.7. Configuration Class. A *configuration* of a software system corresponds to a setup of the operational (and testing) environment. For example, an Internet browser application can usually operate with different types of execution platforms (Intel, AMD), operating systems (Linux, Windows), or runtime options (JavaScript enabled or disabled, browser history enabled or disabled). Therefore, a choice needs to be made for each available option that will lead to a particular configuration of the system. Most software applications can also be customized during the compilation process (e.g., to set the value of certain internal variables, such as the maximum number of processes in an operating system). The choice of a specific execution mode or state for testing would also be part of the configuration of the testing environment (e.g., testing a program in administrator mode vs. testing the same program in user mode).

In the functional profile [Musa 1992], the so-called *environmental variables* describe the "conditions that affect the way the program runs," (e.g., hardware configuration).

An OP that explicitly considers configuration is called *configuration-aware*; otherwise, it is called *configuration-unaware*.

3.1.8. Critical Operations Class. In general, reliability testing (with or without an OP) focuses on the most frequently used operations and functions of a software system. However, the case of critical-but-infrequent operations (or functions) is also well known. A critical-but-infrequent operation or function is one that rarely occurs during the

system lifetime; however, if it occurs and leads to a system failure, the severity of such a failure will be high (e.g., system loss, large financial loss, or loss of human life). An example is a software routine that triggers a vehicle's airbag in the event of a collision. An OP methodology that provides explicit means to address critical-but-infrequent operations and functions is referred to as *criticality-aware*; otherwise, it is *criticality-unaware.* It should be noted that many of the currently critically-unaware approaches could easily be extended to reflect criticality; however, they are not extended to reflect criticality and are, therefore, classified as criticality-unaware.

There are multiple ways to determine if a software operation is critical-but-infrequent. Whether an operation is infrequent is dictated by the occurrence probabilities provided by the OP. On the other hand, well-known techniques to assess the criticality of operations and functions include criticality analysis techniques such as (software) fault tree analysis; (software) hazard analysis; (software) failure mode, effect, and criticality analysis; and impact analysis techniques such as fault injection.

Because the probability of occurrence of the critical-but-infrequent operations is low, the number of test cases that are nominally allocated to test these operations may not be sufficient to uncover potential problems. For example, faults within the corresponding software code may remain undetected. Besides, when a *threshold occurrence probability* is defined below which no test case is allocated (e.g., $\frac{0.5}{N}$, where N is the total number of test cases, see Arora et al. [2005]), then some critical-but-infrequent operations might not be tested at all. The literature [Musa 1993; Arora et al. 2005] proposes several approaches to circumvent this issue:

(1) *Combine infrequent operations*. This allows increasing the total number of test cases allocated to these operations by implementing these tests as a group; however, it may lead to inconsistencies in the OP because the grouping of semantically diverse operations and the number of test cases per critical operation may still be insufficient.
(2) *Weigh probabilities by criticality*. This allows increasing the number of test cases per critical operation by modifying the probabilities of the OP; however, it may distort the final reliability estimates.
(3) *Categorize operations according to criticality.* This consists of defining categories of criticality (e.g., high, medium, nominal, low) and developing a separate OP for each category. Still, the frequency of some operations may remain too low within a category and the definition of categories may need to be extended to consider other factors such as frequency.
(4) *Assign a "minimum number of test cases for infrequent operations."* This guarantees that these operations will always be assigned a minimum number of test cases, which is equivalent to applying an acceleration factor (e.g., ratio between test occurrence probability and field/OP occurrence probability). However, this technique may lead to distorted reliability estimates.

Authors such as Musa [1993] recommend the use of Technique 3. In general, the purpose of these kinds of tests is to sufficiently test the critical-but-infrequent without compromising the accuracy of the reliability estimates or the significance of the OP as a precise representation of the operational usage of the software system. An alternative to representing critical operations (which have not been given much attention by the software reliability research community) would be the tailoring of simulation-based techniques such as those used in the reliability simulation of complex systems. These techniques (importance sampling-variance reduction techniques [Hammersley and Handscomb 1964]) are designed to handle the accurate representation of

low-probability, high-criticality events with a minimum number of simulation runs (which here can be considered as the equivalent of tests) while preserving the quantity of interest (here reliability).

Approaches and Techniques 1–4 can be further classified under three global categories: (1) *adjustment of the allocation of test cases* (Techniques 1 and 4), (2) *adjustment of the probabilities of the OP* (Technique 2), and (3) *splitting of the OP* (Technique 3). An additional category would consist of (4) *testing separately* the critical-but-infrequent operations using techniques other than OP testing (e.g., robustness testing, fault injection). Conversely, the integration of faults and invalid inputs into the OP itself proposed in Section 3.2 would also be an alternative to enhancing the testing of the critical-but-infrequent operations.

3.2. Software Boundary Classes

This section describes the *Software Boundary* class of OP. The *Software Boundary* class is divided into the classes *Executive Scope*, *External Errors,* and *Input Data*. Refer to Table V for a classification of the papers reviewed with respect to these classes

3.2.1. Executive Scope Class. The executive scope encompasses the executive layers of the software such as the hardware platform (which provides basic software executive resources such as CPU and RAM) and the operating system software (which manages the basic executive resources and provides higher-level ones such as scheduling, synchronization, and file system). In addition, the executive scope includes other system and user tasks that concurrently execute and compete for the same executive resources. An explicit consideration in the OP of the events and errors proceeding from these layers and tasks leads to an *executive-aware* OP, in contrast to an *executive-unaware* OP.

Failures (or errors) propagated from the executive layers or from other tasks that reach the software application are, for example, a type of input events of the highest importance that are commonly ignored in an OP. However, these failure events might not be negligible. For example, Koopman and DeVale [1999] report percentage failures of POSIX operating systems ranging between 3.8% and 29.5%, while Rodríguez et al. [1999] report percentages for synchronization failures of a microkernel ranging between 9% and 87%.

Whittaker and Voas [2000] provided didactic examples of failures from the executive scope that propagate and affect a word processor application even under moderate human use, namely: a document "is marked read-only by a privileged user," "the operating system denies additional memory," "the document auto-backup feature writes to a bad storage sector." Huang et al. [2007, 2011] have proposed to extend the OP with a fault injection profile that accounts for the failure events that originate in the hardware platform and their probability of occurrence. This is done by modeling the physics of failure and propagation mechanisms of the several computer hardware layers from bottom to top (semiconductor layer, gate layer, register layer).

The executive scope meta-model is shown in Figure 8 in Appendix B.

3.2.2. External Errors Class. In addition to failures originating from the executive scope of a system, another important type of failure that can reach the external input interface of a software application during operation is an external error. For example, the inputs provided by a human user may not always be correct: A user may input a name where a phone number was required or a user may push a wrong button on a control panel of a nuclear power plant. These are examples of external errors consisting of invalid software inputs that may frequently occur in practice and must be considered during software testing. Therefore, the OP should not only include nominal (i.e., correct) inputs from human users (or other *nonhuman* users such as other

computers and systems), but it should also include invalid (i.e., incorrect) inputs that can reach the external input interface of the software application. If the OP considers invalid inputs, it is referred to as *error-aware*, versus an *error-unaware* OP, which only considers nominal external inputs.

The analysis of invalid software inputs has traditionally belonged to the realm of software testing techniques such as robustness testing (e.g., Csallner and Smaragdakis [2004], Koopman and DeVale [1999], and Rodríguez et al. [1999]). It is rare to find in the OP literature approaches that explicitly address this aspect. An approach broaching this issue was proposed by Popovic and Kovacevic [2007]. The authors introduce the concept of *stress operational profile* to refer to a FSM-based OP extended with additional states and transitions that occur due to invalid inputs. These additional states and transitions, which are not present in the original OP model, are called *hidden*. The authors refer to this approach as *model-based robustness testing*. The approach is applied to communication protocols implementations, and the invalid inputs consist of syntactically and semantically faulty messages. A stress operational profile "is constructed from a statechart [model of the nominal OP] by adding reactions [(i.e., hidden states/transitions) to the considered faulty messages] and by adorning state transitions" with probabilities. Test cases are generated and executed automatically, and their quality is determined by calculating a mean significance confidence level.

3.2.3. Input Data Class. *Input data* is defined as input variables with their names, values, types, and the constraints they need to satisfy. For example, for an ATM machine, the input data could consist of the PIN of type integer constrained to four digits. The selection of input data is an ultimate and unavoidable task needed to make testing happen in practice. This specifically involves determining what actual values are to be assigned to the input variables of the software under test. An OP approach that provides explicit means to specify input data is referred to as *data-inclusive* OP; otherwise, it is referred to as *data-exclusive* OP.

Some examples of how different authors approach incorporating input data in the OP are provided hereafter. Bousquet et al. [1998] describe a simulator that randomly generates input data that satisfies the environment specification.. Gittens et al. [2004] provide an extended OP that includes profiles for defining and characterizing test data. These are referred to as *structural profile* and *data profile*. The former characterizes the structure of the software and provides occurrence probabilities for quantifiable data structures (e.g., number of loops, size of arrays). The latter provides occurrence probabilities for the values of the different software inputs by taking into account data types, ranges, most frequently occurring data, largest data lengths, and so on. Shukla et al. [2004a] provide guidelines for assigning values to input parameters of object-oriented programs by taking into account the type of the parameters, constraints associated with individual parameters, and relationships across parameters. These are further used to generate appropriate values for input parameters. Woit [1993] provides the semantics of input classes that specify how the input variable arguments are selected. The selection criteria include value selection according to either a uniform distribution or a condition function. Hamlet et al. [2001] consider inputs a semi-inclusive manner. Clearly there is an attention to inputs through the explicit specification of input domains and attention to the constraints between inputs through the explicit consideration of transformations of the input domain by components being executed in the call sequence preceding the component of interest. In Musa [1993], identification of input variable space and partitioning of the input variable space are explicit steps of the construction of the OP. In this, he discusses how to identify the valid and invalid inputs, and their possible states (i.e., values).

3.3. Dependency Classes

This section describes the *Dependency* class of OP. The *Dependency* class is divided into the classes *Code Dependency* and *Field-of-Interest Dependency*. Refer to Table V for a classification of the papers reviewed with respect to these two classes.

3.3.1. Code Dependency Class. If the definition, construction, or use of the OP requires knowledge about or access to the internal structure or to the internal data of the source code of the software application under consideration, then the OP is referred to as *white-box*. Otherwise, if the code of the software application under consideration is not needed to construct or run the OP, it is referred to as *black-box*. A black-box OP views the software application as a *black*, *closed* box where internal code details are not available and only external input and output interfaces are known and can be accessed. On the other hand, a white-box OP views the software application as an *open* box that makes all internal information, structures, and data available.

An example of white-box OP would be the OP provided in Gittens et al. [2004]. The defined structural and data profiles need access to information of the software application that is only available at the internal code level, such as data types and structures (arrays, linked lists, abstract types, database tables, etc.) and measurable quantities of data structures (number of rows/columns of arrays, tables' length, etc.). The software code is also instrumented to obtain and monitor the required information at runtime.

Other research work analyzed follows a black-box OP approach. All information needed about the software application is found in available usage data (e.g., historical data, experimental data, expert judgment), high-level software specifications (e.g., requirements documents, software manuals), or at the level of the external software interfaces (e.g., API—Application Programming Interface, such as the public attributes and methods defined in an object-oriented class). Also, the usage models provided (e.g., Markov chains, statecharts) do not include internal states of the software application. This is consistent with the well-known principle that a usage model is not meant to represent the behavior of the system but the use of the system; therefore, there should be no need for modeling the internal states of the system (e.g., see Prowell [2003]). Note, however, that the modeling methodology defined in Guen et al. [2004] encompasses the notion of the "internal state of the system under test" and differentiates it from the *states of the Markov chain*. However, there is no evidence that internal details of the software are used or required to build or apply in practice the models employed (see Dulz and Zhen [2003] and Guen and Thelin [2003]). Although it is sometimes referred to as a white-box approach due to its treatment of a system as a collection of components, the approach by Hamlet et al. [2001] is categorized here as black-box according to the presented definition.

3.3.2. Field-of-Interest (FOI) Dependency Class. An OP that has been primarily developed for use with systems from a particular application area (e.g., enterprise applications, embedded systems, web applications, GUI-based systems, reactive synchronous systems) is referred to as *FOI-specific* OP; otherwise, it is an *FOI-independent* OP.

Most OP methodologies have been developed to be generic and independent from any particular application area. The OP methodologies proposed by Musa [1992, 1993] and Whittaker and Thomason [1994] have been used by many practitioners and applied to multiple systems and application area (see also Section 3.1.5).

The need for developing a FOI-specific OP arises because a generic OP might not be suitable to certain types of applications and because a FOI-specific OP will commonly lead to the generation of tests of higher efficiency and of more accurate results for

the specific set of applications targeted. Brooks and Memon [2007] apply and extend Whittaker and Thomason's approach and related approaches to GUI systems, whereas Bousquet et al. [1998] have worked on the Reactive Synchronous Systems field. The consideration of specific areas may lead to the development of new structures, for example, probabilistic event flow graphs for GUIs, or to add dimensions to existing structures (e.g., a time index for the tree structure in Bousquet et al. [1998]).

3.4. Development Classes

This section describes the *Development* class of OP. The *Development* class is divided into the classes *Lifecycle Phase* and *Tool Support*. The review results are presented in Table V.

3.4.1. Lifecycle Phase Class. An OP that is readily built early during the lifecycle of the software system is referred to as *early* OP; otherwise, it is referred to as *late* OP. For example, an OP built from the requirements is an early OP, whereas that built from the implemented software is a late OP. The building of an OP involves a number of steps, most of them can potentially occur early in the software lifecycle. These steps include the (1) development of the OP model, (2) derivation of probabilities, (3) analysis of the model, (4) test planning (including pass/fail criteria and stop criteria), and (5) generation of test cases (including assessment of quality metrics about the test cases). The test execution and analysis of test results require that the software system (or at least a prototype) is implemented.

Ideally, the building of an OP should occur as early as possible in the software development lifecycle (before implementation phases, preferably during requirements analysis) and be progressively refined throughout subsequent lifecycle phases. After the software system has been implemented and released, the OP should still change and be improved throughout the life of the system (e.g., operation, maintenance, upgrades, new releases).

The majority of OP analyzed in the literature can be classified as early (see Table V). Note, however, that most of these works focus on the description and demonstration of the major OP features without providing details about how the development of the OP actually occurs and is progressively refined in the lifecycle of the software system. For example, in Brooks and Memon [2007], the focus is on regression testing of GUI applications and the OP is developed from available usage information. The OP methodology for component-based systems proposed by Hamlet et al. [2001] can be considered early from the point of view of the system, but late with respect to an individual component of the system given that the OP of a component needs (among others) failure rate data calculated via random testing. On the other hand, the OP proposed by Gittens et al. [2004] can be classified as late. In this work, the structural and data profiles seem to require information available only during the software implementation phase, such as internal data types and structures.

3.4.2. Tool Support Class. An OP approach that is well supported by automated tools is referred to as *automated* OP; otherwise, it is referred to as *nonautomated*.

There are few integrated tools specifically developed for testing software with an OP. Most of these tools belong to the area of statistical usage testing, namely: GUITAR [Brooks and Memon 2007], JUMBL [Prowell 2003], Lutess [Bousquet et al. 1998], and MaTeLo [Guen et al. 2004]. JUMBL and MaTeLo are meant to be FOI-independent tools, whereas GUITAR and Lutess are specific to GUI systems and synchronous reactive systems, respectively. However, these tools may not provide full automation. For example, as stated in Prowell [2003], "JUMBL does not directly support construction of models or automated execution of test cases."

Table VIII. Examples of Testing and Modeling Tools Able to Be Used to Support OP Testing

Tool	**Modelling**	**Test Generation**	**Test Execution**	**Source***
AETG		✓		aetgweb.argreenhouse.com
AutoFOCUS	✓			autofocus.in.tum.de/index.php/Main_Page
DGL		✓		cs.baylor.edu/~maurer/dgl.html
JUnit-based	✓	✓	✓	www.junit.org
ModelJUnit	✓	✓		cs.waikato.ac.nz/~marku/mbt/modeljunit/
Objecteering	✓			objecteering.com/products_uml_modeler.php
Poseidon	✓			www.gentleware.com
Rational Robot			✓	ibm.com/software/awdtools/tester/robot/
Rational Rose	✓			ibm.com/software/awdtools/developer/rose/
Rational Tau	✓	✓	✓	www-01.ibm.com/software/awdtools/tau/
Reactis	✓	✓	✓	www.reactive-systems.com
Stateflow	✓			www.mathworks.com/products/stateflow/
TGV		✓		irisa.fr/pampa/VALIDATION/TGV/TGV.html
T-VEC	✓	✓	✓	www.t-vec.com

*Last accessed: 1/18/2013.

Some OP approaches take advantage of existing general-purpose testing and modeling tools. For example, Popovic and Kovacevic [2007] create a testing environment that combines *JUnit* to automate the execution of test cases, while Zhen and Peng [2004] suggest the use of *Rational Rose* and *Telelogic Tau* (now *Rational Tau*) for model specification in MaTeLo. These and other related tools are listed in Table VIII.

Most tools in Table VIII that provide modeling capabilities are based on UML, finite state machines, or statecharts. MaTeLo [Guen et al. 2004; Zhen and Peng 2004] and JUMBL [Prowell 2003] also use a number of tools specific to Markov chains such as the Markov Usage Editor (MU), an editor for Markov chains; The Model Language (TML), a notation for specifying Markov chains; the Markov Chain Markup Language (MCML), an XML-based notation for representing Markov chains; and the Model Markup Language (MML), an XML-based markup language also used to represent Markov chains.

Table VIII also includes tools that provide test case generation and execution capabilities. The description of test cases is often done in the Testing and Test Control Notation (TTCN) (e.g., see MaTeLo [Guen et al. 2004]) or in XML (e.g., see the Test Case Markup Language (TCML) tool used in JUMBL [Prowell 2003]). Conversely, Shukla et al. [2004b] use Object-Z to specify oracles for pass/fail test criteria and JUMBL [Prowell 2003] integrates Graphviz/Graphlet tools for the representation of test results. For the reliability analysis of test results, Arora et al. [2005] use the Computer-Aided Software Reliability Estimation (CASRE) tool.

4. OP META-MODEL

The OP classes shown in Figure 6 are represented as a meta-model in Figure 7. The OP meta-model is built to enhance the understanding of the various concepts required to build the OP and the relationship between these concepts. These relationships indicate that the definition of OP is multifaceted. The definition of each concept is provided in Table XIII in Appendix B. Each of these concepts, the important relationships between them, and the ways they are addressed in the literature reviews are provided in Sections 3.1 through 3.4. Some of these concepts are further developed into other meta-models provided in Appendix B.

Figure 6 illustrates that an OP is composed of one or multiple *Profiles*. A *Profile* is related to a *Structure* and has multiple *Inputs*. The concepts of *Profile*, *Structure*, and *Input* are further elaborated in Figures 9, 10, and 12 in Appendix B.

An OP can be defined at different *Abstraction Levels* such as *System Level* or *Component Level*. A *System-Level* OP and a *Component-Level* OP are related to each other through the *Context*. The *Context* is a set of mapping relationships derived from the *System-Level* OP. Based on these mapping relationships, the *Component-Level* OP is modified, thus placing the *Component-Level* OP in the context of the system.

An *Application* OP is an aggregate of one or multiple OPs and may also contain an *Executive Scope Profile*. An *Application* OP is also related to *Source Code*, which may or may not be required when defining the OP.

An OP may consider *Critical Operation*, *External Error*, or *Input Data* information in its definition. The *Input Data* is characterized by its variable name, values, data-type, and input data constraints.

Consideration of one or multiple *Fields-of-Interest* leads to the extension of an already defined OP. The *Field-of-Interest* consideration can also lead to a significant change in the *Structure* of the *Profile*, or may simply add dimensions to the *Inputs* of the *Profile*.

5. DISCUSSION AND OPEN RESEARCH QUESTIONS

The following section discusses open research questions that were identified throughout this review. The discussion first addresses specific research issues for each of the OP characteristics followed by more general considerations. Pairwise relations between OP characteristics are examined last.

Profiles: Rules should be defined to specify which profiles should be used for a given application. Issues of cost and coverage should be considered in addressing this problem.

Structure: Rules should be defined to specify which structure should be used in the context of a given profile and which should be used for a specific application. In addition, a simplified representation of complex models, expansion and compression of models, and rules of abstraction to handle state explosion for complex systems [Whittaker and Poore 1993] should also be defined. Finally, relationships to profile type should be examined to determine whether a particular profile type dictates a particular structure.

Probability of Occurrence: The state of the art does not provide an analysis of advantages and disadvantages of these different approaches for calculating OP probabilities. An N-conditional approach (with $N \gg 1$) should provide more accurate results [Brooks and Memon 2007] at the expense of higher testing effort and cost. However, one would also need to take into account other aspects that contribute to the definition, value, accuracy, and representativeness of the OP probabilities such as the data source used to derive probabilities (e.g., field data, experimental data, heuristics), the models employed to represent the system's use (e.g., Markov chains, statecharts), the number and type of profiles considered (e.g., usage profile, user profile, customer profile), and the depth of dependency permitted (e.g., in modeling languages, test generation algorithms, automated tools).

Another topic for future research would be to consider additional types of stochastic dependencies. Dependencies may also arise due to the type, time, and ordering of past events and these could also have an impact on the occurrence probability of future events and thus constitute additional dimensions of analysis.

Computability: Rules should be developed to define when the use of an executable might be warranted. This also should include performance studies.

Abstraction Level: As demonstrated through this literature review, the literature on defining OPs for software components is scarce. Issues of relation between OPs at

the system level and OPs at the component level are still open and require further study. Experiments should be performed to validate hypothesized relationships between system-level and component-level OPs [Hamlet et al. 2001], and tools should be developed to perform the experiments. Rules must be defined to determine the level of component granularity. The issue of reusability of the OP of components should be studied further and mapping relations must be defined.

Originator: Methods for grouping and characterizing users and customers into meaningful groups should be explored further [Brooks and Memon 2007; Gittens et al. 2004; Juhlin 1992] and validated [Elbaum and Narla 2001]. In addition, one might investigate the use of models of human interaction with computers to characterize detailed behavior, especially if one wishes to characterize erroneous behavior [Shneideman and Plaisant 2010; Fabio et al. 2012]. See also the discussion under Input Data.

Software Scenarios: The use of techniques that define scenarios other than the ones proposed in the papers investigated here should be explored and the extent to which they can contribute to building an OP should be analyzed. Such techniques might include event trees, fault trees, event sequence diagrams, Petri-nets, and simulations of models (e.g., simulations of physical processes that interact with software systems). In general, OPs could be extracted from behavioral models of the system within the software and a more extensive analysis of OP-generation techniques from system models should be designed.

Mode: There are no open issues related to Mode.

Configuration: Mapping rules to transform the OP of one software version into the OP of another version of the same software [Brooks and Memon 2007] should be developed. The completeness of the operating environment configuration specification (including external hardware and software) of the application OP must also be evaluated [Whittaker and Voas 2000].

Critical Operations: Relationships between domain and critical operations should be examined. None of the work on safety classification appears in the existing OP literature. The notion of consequence of the failure of an operation is also not formally part of the OP.

Executive Scope: Information on faults originating from the operating system, hardware platform, and concurrent tasks is scarcely considered in OP development [Whittaker and Voas 2000]. Different methods that provide this information should be evaluated and feasibility analysis of inclusion of executive scope in OP development should be conducted.

External Errors: Little attention has been given in the literature to the systematic study of external errors. Because these are likely contributors to software failures, systematic approaches that support these aspects of OP definition should be developed (e.g., see Wei et al. [2010] for an approach to systematically analyze input failures for software used in safety critical applications typically studied in a risk-based framework, e.g., probabilistic risk assessment models).

Input Data: Input data has a strong relation to the originator. A better understanding of the originator may help characterize input data. Questions that much be answered include the degree to which a characterization of the data is required and in what context. Because it has the lowest granularity level, input data is possibly the most difficult to predict. In addition, there seems to be an interest in developing approaches for data grouping based on semantic characterization [Gittens et al. 2004].

Code Dependency:Very few papers discuss code dependency. Rules should be defined to specify when code should be used to develop the OP and when the benefits of doing so outweigh the disadvantages (i.e., the cost and the fact that the final OP would be available only in the later phases of software development).

Table IX. Pairs of Classes with Significant Correlations

Class	Class with Significant Correlation
Profile	Configuration
Abstraction Level	Input Data
	Tool Support
Mode	Input Data
Configuration	Critical Operations
	Executive Scope
	Lifecycle Phase
Critical Operations	Executive Scope

Field-of-Interest Dependency: FOI dependency must be clarified further. One should also explore the effects of domain characteristics on OP domain. Domain tailoring rules could be created.

Lifecycle Phase: Research challenges may include the quantification of benefits and costs of early OPs versus late OPs, the development of a process for (formal) continuous refinement of the OP during the development lifecycle, and the analysis of reusability aspects across systems and projects.

Tool Support: While tools have been developed to support certain aspects of OP definition and development, more efforts directed toward the development of efficient tools are warranted [Hamlet et al. 2001; Shukla et al. 2004a]. A comprehensive survey of existing tools and analysis of their contributions toward the building of OP should be undertaken. No real benchmarking of tools for OP construction exists, and not all profile types are supported by tools.

General: Other research issues include the following: (1) Currently, no measures of OP quality exist. If defined, such measures could include measures of completeness [Brooks and Memon 2007]. (2) Studies focused on the issue of OP scalability [Hamlet et al. 2001] should be undertaken, especially for large systems. (3) An efficient and cost-effective process of OP model construction should be defined for OP definition [Whittaker and Poore 1993].

OP Meta-model: The OP meta-model is a first step toward the standardization of the OP. Building tools conforming to a standard OP meta-model will ensure a unified OP construction and will enhance interoperability.

Correlation Analysis

It is also interesting to determine whether particular OP classes are correlated. Correlation coefficients between pairs of classes were calculated and pairs of classes that display significant correlations (95% confidence) [Cramer 1997; Rosenberg and Binkowski 2004] are given in Table IX.

The pairwise distribution of papers for each combination of significantly correlated classes is given in Table X.

As an example, a significant correlation exists between Profile and Configuration, which stems from the fact that profile developers will first and foremost focus on the set of operations that the software application should carry out rather than on issues related to configuration (which are related to the development of a configuration profile). Therefore, if a single profile is used, it will not be related to configuration.

As another example, a significant correlation exists between Critical Operations and Executive Scope. Critical Operations typically appear in applications whose failures have significant consequences. Therefore, it is expected that such applications would be required to function in stable, reliable, and sometimes real-time environments. This would require a characterization of this operating environment (i.e., the Executive Scope), hence the existence of the correlation.

Table X. Pairwise Distribution of Classes Significantly Correlated (Chi-Square Critical Values: For a 2×2 Matrix $c > 3.84$; for a 2×3 Matrix $c > 5.99$; for a 2×4 Matrix $c > 7.82$; for a 3×4 Matrix $c > 12.59$)

	ConfiA	ConfigUA
Single	1	11
Multi	4	1

	Data-In	Data-Ex	Auto	Non-Auto
Comp	3	0	0	3
Sys	3	10	10	3

	Data-In	Data-Ex
ModeA	4	1
ModeUA	2	9

	CriA	CriUA	ExeA	ExeUA	Early	Late
ConfiA	3	2	3	2	3	2
ConfigUA	0	12	0	12	11	0

	ExeA	ExeUA
CriA	2	1
CriUA	1	13

	Tree	S-Based	Set
Early	3.5	7.5	2
Late	0	0	2

	HW	SW	Human	Unspec.
Early	0.33	0.33	5.33	8
Late	1.33	0.33	0.33	0

It should be noted that while some significant correlations are expected to exist between pairs of classes, they may not appear in this body of work. For example, we would expect a significant correlation to exist between External Errors and Critical Operations.

6. CONCLUSION

This article is dedicated to the characterization of the OP model used to generate test cases. The OP model is described in terms of classes (i.e., dimensions of the classification) and a meta-model that describes the relationships between classes is defined. An example of such a relationship is the need for an OP to be configuration-aware prior to being able to be executive-aware, while the reverse is not true (Table V).

Musa [1993] defined the OP as a “quantitative characterization of how a system will be used.” Our review has shown that a variety of OP models can be constructed that are dependent on the field-of-interest of the application under test, the criticality of the application, the selected scope, and the degree of accuracy in replicating the environment in which the application will function, capture its dynamic properties, and so on. The OP classes provide a taxonomy that differentiates and analyzes OP characteristics from several perspectives: *common features* (profiles, structure, scenarios, etc.), *software boundary* (modeling of events and errors such as invalid inputs or operating system/hardware errors), *dependency* (code dependency, and field-of-interest dependency), and *development* (lifecycle phases, and tool support).

In the authors' opinion, as the complexity of software systems increases, development of an OP becomes increasingly challenging. The complexity arises particularly due to the increased interplay between the system components and varied nature of the system users (e.g., Web-based or distributed systems OPs). Thus, the characterization of the probability of occurrence of events becomes a difficult task. To resolve such issues, complex structures must emerge. The emergence of such new structures or the unconventional use of existing structures is observed in papers reviewed where authors have either introduced entirely new structures [Brooks and Memon 2007; Woit 1993] or force-fitted complex interaction probabilities to existing structures [Bousquet et al. 1998]. The complexity of characterizing probabilities of occurrence further increases in the early stages of software development. As an attempt to resolve this issue, Kumar [2008] has proposed the use of linguistic variables and fuzzy-logic based OPs.

The issue of an ideal OP as pointed out by [Juhlin 1992] is still unaddressed. It is obvious that an OP cannot be all-inclusive. As such, researchers have primarily tried to find pragmatic approaches best suited to the application of interest. However, which OP would be best suited to the application under consideration is unclear. As such, OP selection rules must be defined. To address the ideal OP issue further, various measures related to properties such as completeness, scalability, cost-effectiveness, and feasibility must be studied.

The chronological analysis of the paper does not show any specific trend in OP development. The concerns seem to be uniformly addressed over time. Many single-profile OPs, especially with state-based structures, can potentially be extended to multiple profiles; however, no such extension mechanisms or rules are provided. In 8 out of 17 papers, we see that the originator is unspecified. More research should be performed in characterizing the nature of the originator to build more accurate OPs. Only 2 out of 17 papers use a white-box approach to OP construction. Even though it is desirable to begin constructing OPs early in the development phase, extensions and update mechanisms should be defined to integrate code-level information late in the development. This will contribute toward completeness of the OP.

Overall, we believe that classifications and taxonomies developed from multiple angles and perspectives provide valuable and unique viewpoints and understanding of the use of OP. In this paper, effort has been given not only on describing, reviewing, and assessing the state of the art and practice but also on providing analyses oriented toward the development of future extensions and improvements of theories and techniques that involve OP model development.

APPENDIX A: APPENDIX TO METHODOLOGY

Table XI. Complete List of Paper Sources

Conferences	**Papers**
International Symposium on Software Reliability Engineering (ISSRE)	6
Reliability and Maintainability Symposium (RAMS)	2
International Symposium on Software Testing and Analysis (ISSTA)	1
Software Technology and Engineering Practice (STEP)	1
Asia-Pacific Software Engineering Conference (APSEC)	1
International Conference on Automated Software Engineering (ASE)	1
Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)	1
International Conference on Software Engineering (ICSE)	1
Journals	**Papers**
ACM Transactions on Software Engineering and Methodology (TOSEM)	1
IEEE Software	1
IEEE Transactions on Software Engineering	1
Informatica	1
Software: Practice and Experience	1

Table XII. Keywords Used to Operationalize the Classification Process

Class	Keywords	Comments/Insights
Profile	*customer profile, user profile, configuration profile, system-mode profile, data profile, process profile*	If the OP contains one profile, it was classified as *single-profile*; else, *multiple-profile*. The concept of multiple profiles led us to investigate different types of inputs and their providers. Because of the varied nature of providers, we created the *originator* class.
Structure	*Markov chain, state machine, statechart, set, tree, hierarchy, flow graph, graph, model*	The structures found with these keywords resulted in two important insights. First, *probability of occurrence* is a property of the structure and is not a distinct class. Second, *computability*, which accounts for usage dynamics, is a property of the modeling technique used. Upon realizing these dependencies, we removed these two classes from our original classification. Also, some unconventional and hybrid (state-based plus tree) structures were discovered. After analyzing these papers' context, we formed the class *field-of-interest*. We found that the structure and inputs of an OP are modified based on fields of interest.
Abstraction Level	*component, module*	
Originator	*user, event, customer, input, human*	The class was an outcome of the analysis of multiple profiles.
Scenario	*sequence, scenario, event sequence, execution sequence.*	
Mode	*mode, state, overload, administra–, startup, shutdown, nominal*	
Configuration	*operating system, configuration, hardware*	During our analysis of the *configuration* class, we discovered two categories of elements: (1) elements that belong to different layers of the computer running the software application, and (2) elements that are external to the computer. This led to the development of the *executive scope* class, broadly classified under the *boundary* class. The relationship is explored in detail in Figure 9.
Critical Operations	*critical, infrequent, high consequence, severity, rare*	
Input Data	*input, data, data type, data range, event, operation, message, data name*	During our analysis of inputs, we discovered that input data and input data type were formally defined in some papers. This explicit treatment of the input led to the *input data* class.
Executive Scope	*hardware, platform, CPU, RAM, OS*	If an OP explicitly considers a software application's executive layers (hardware platform), operating system, or other system or user tasks that concurrently execute and compete for executive resources, then the OP is *executive-aware*; else, it is *executive-unaware*.
External Errors	*error, illegal, faulty, invalid, external*	
Code Dependency	*source code, code, program*	
Field-of-Interest Dependency	*domain, synchronous, real-time, multimodal, GUI*	This class was an outcome of our analysis of the *structure* class.
Lifecycle Phase	*specification, code, requirements, design, test, implementation, life cycle*	
Tool Support	*jumbl, lutess, matelo, auto, tool, junit, Poseidon, rational, spectest, stateflow, manual–*	

APPENDIX B: APPENDIX TO META-MODELS

Table XIII. Operational Profile Meta-Model

	Elements	Description
1.	Operation Profile (OP)	An operational profile is a set occurrence probabilities that characterizes the usage of an application. **Attributes**: Executable: Boolean {default = "N"} Defines whether the OP relies on an executable model or not. If "Y," the OP relies on an executable model. If "N," the OP does not rely on an executable model. LPhase: string Defines the point in the development lifecycle phase at which the OP is first built. For instance, "early" (before implementation) or "late" (after implementation). ToolSupp: Boolean {default = "N"} Defines whether the OP approach is supported by tools. If "Y," a tool support is present. If "N," a tool support is not present.
2.	Profile	The profile is a cross-sectional usage view of the application along a given dimension. It includes the elements that make up that dimension and the occurrence probabilities of such elements during use. For example, the "functional dimension" view leads to the notion of a "functional profile" which is the set of functions that the application should implement and their occurrence probabilities. The "user dimension" leads to the notion of "user profile," which partitions the users of the application into distinct groups with various patterns of usage and provides the probability of a user belonging to such a group. A single or multiple profiles form(s) an operational profile.
3.	Structure	Structure defines the type of mathematical model used to describe the operational profile. Models identified in the literature so far include trees, Markov chains, Harel statecharts, probabilistic event flow graphs, and sets.
4.	Input	Input defines the set of inputs to the application. It includes events as well as actual data.
5.	Abstraction Level	Abstraction level defines whether the OP is built for a system or for a component of the system.
6.	System Level	System level represents cases where the OP is developed for the software as a whole.
7.	Component Level	Component level describes cases where the OP is developed for the software seen as a component of a larger system.
8.	Context	Context is defined as the mapping relationships between system-level OP and component-level OP. The component is assumed to belong to a system operating in an environment characterized by system-level OP.
9.	Executive Scope Profile	Executive scope profile is the portion of the OP that describes inputs generated by the executive layer (i.e., Operating system, computer hardware, and application NOP).
10.	External Error	External error defines whether or not the OP includes erroneous external inputs.
11.	Field-of-Interest	Field-of-interest refers to particular application area (e.g., web applications, embedded applications, multi-modal (here modal refers to speech, keyboard). Field-of-interest dependence determines whether a special type of OP should be built to allow inclusion of the particular characteristics of the application field of interest.
12.	Source Code	Source code of the software application for which the OP is being developed. The OP may or may not require access to the source code. If access to the source code is required the OP is labeled as white-box; otherwise, it is termed as black-box.
13.	Application OP	Application OP is the OP of the application under consideration.
14.	Critical Operation	Critical operation is defined as the set of low probability of occurrence operations or functions whose malfunction might lead to severe system-level consequences.
15.	Input Data	Input data is a type of input. Input data is includes variables with their names, values, types and the constraints they need to satisfy.
16.	Variable Name	Name of an input data variable.
17.	Values	Values taken an input data variable.
18.	Data Type	Data type (e.g., integer) of an input data variable.
19.	Input Data Constraint	Constraint imposed on the input data. An input data constraint may involve one or multiple input data variables.

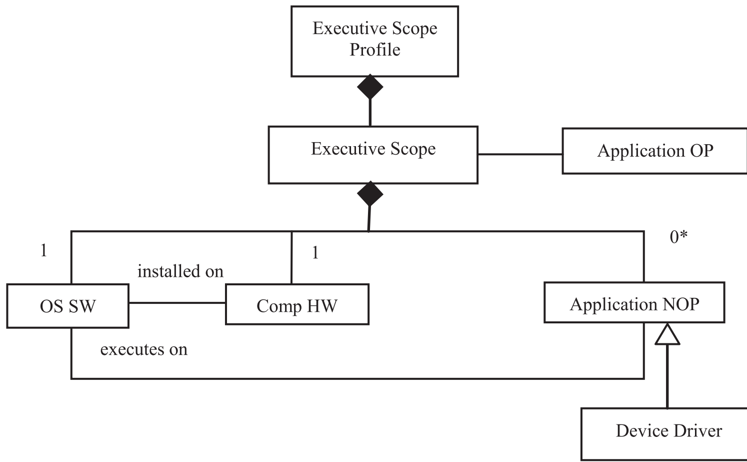

Fig. 8. Executive scope meta-model.

Table XIV. Executive Scope Meta-Model

	Element	Description
1.	Executive Scope Profile	See Table XIII.
2.	Executive Scope	The executive scope encompasses the software's executive layers such as the hardware platform (which provides basic software executive resources such as CPU and RAM) and the operating system software (which manages the basic executive resources and provides higher-level ones such as scheduling, synchronization, and file system).
3.	Application OP	See Table XIII.
4.	Application NOP	Application NOP is the OP of software applications distinct from but residing on the same computer platform as the application under consideration.
5.	Operating System Software (OS SW)	"A collection of software" [. . .] "that controls the execution of computer programs and provides such services as computer resource allocation, job control, input/output control, and file management in a computer system" (modified from [IEEE 24765]).
6.	Computer Hardware (CompHW)	"Physical equipment used to process, store, or transmit computer programs or data" [IEEE 24765].
7.	Device Driver	"A computer program that controls a peripheral device and, sometimes, reformats data for transfer to and from the device" [IEEE 24765].

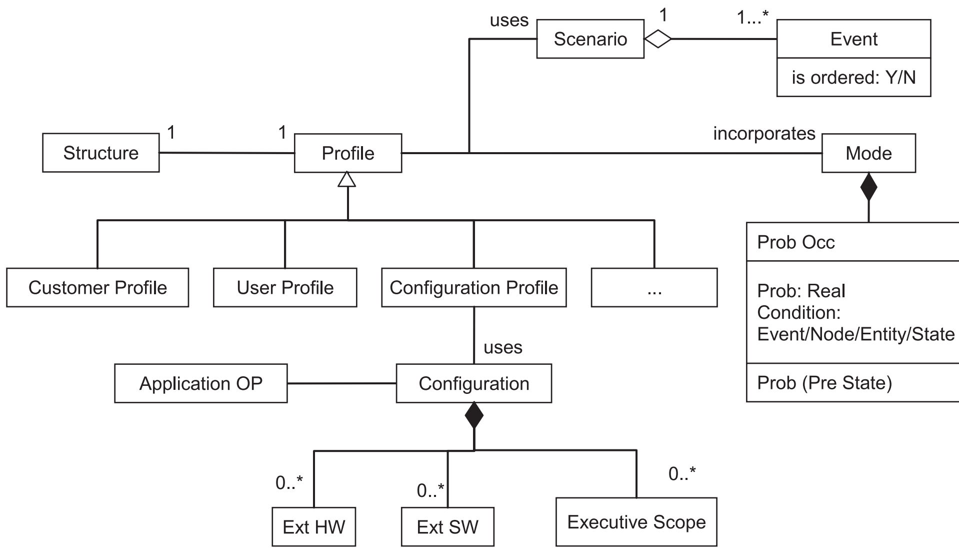

Fig. 9. Profile meta-model.

Table XV. Profile Meta-Model

	Elements	Description
1.	Profile	See Table XIII.
2.	Structure	See Table XIII.
3.	Configuration	"The arrangement of a computer system or component as defined by the number, nature, and interconnections of its constituent parts" [IEEE 24765]. A configuration could be physical or logical. It is the environment in which one or more usages make take place [Juhlin 1992].
4.	Configuration Profile	"The notion of configuration leads to the notion of configuration profile" [Juhlin 1992]. A configuration profile provides the probability of occurrence of various configuration of the application of interest.
5.	Executive Scope	See Table XIV.
6.	Application OP	See Table XIII.
7.	Probability of Occurrence (Prob Occ)	Probability of occurrence is the probability of occurrence of the elements used to define the OP (i.e., events, input data, operations, functions). Probability dependency is defined using attribute "condition" of the "probability of occurrence." Condition is meant to contain a set of events. The length of the set denotes the number of previous events to be considered and thus the Value of "N" is N-conditional. If the length is zero, the probability of occurrence is unconditional. This would be represented with a condition of "ϕ." The length determines to some extent the acceptable structure of the OP. A length of "1" entails a Markov model. **Attributes**: Prob: Real The value of the probability of occurrence of the associated element (i.e., event, node, or entity) Condition [0..$*$]: <enumeration> Specifies the set of conditional elements. The condition type is <enumeration>, which could be a set of events, nodes, entities, or states, etc. that is specific to the structure being used to define the profile. The <enumeration> is typically an ordered set. **Operations** Prob() This method calculates the probability value based on the condition specified. Typically the value is calculated from a known probability distribution function (pdf).

Continued

Table XV. Continued

	Elements	Description
8.	Scenario	Scenario is a sequence of user or externally generated inputs (mostly sequences of events) that accomplish a specific application goal. A profile incorporates the scenario information (scenario-aware) or not (scenario-unaware). **Constraints** **C1**: OP.Profile.Scenario Event.isOrdered = "Y"
9.	Mode	A mode represents a state or a set of states [Bousquet et al. 1998; Whittaker and Thomason 1994]. A mode can also be a set of functions or operations that are grouped for convenience in analyzing execution behavior [Musa 1993]. The modes form the system mode profile [Arora et al. 2005, Musa 1993] or the process mode profile [Gittens et al. 2004]. A profile could be mode aware or mode unaware.
10.	Event	An event is defined as an occurrence at a particular point in time that may lead to a change in the state of the application. **Attributes**: isOrdered: Boolean {default = N} Specifies whether the event is ordered. If "Y," the event is ordered. If "N," the event is not ordered
11.	Application NOP	See Table XIV.
12.	External Hardware (Ext HW)	External Hardware is a collection of hardware components external to the computer hardware under consideration. The computer hardware under consideration is the computer hardware on which the application under consideration is running.
13.	External Software (Ext SW)	External software is the collection of software components that runs on the external hardware and interfaces with the application under consideration.

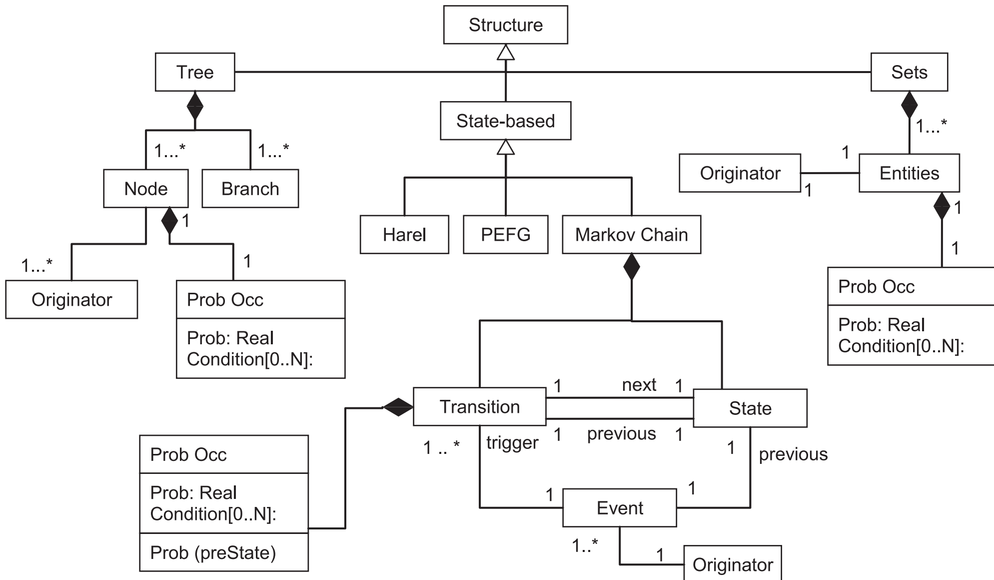

Fig. 10. Structure meta-model.

Table XVI. Structure Meta-Model

	Elements	Descriptions
1.	Structure	See Table XIII.
2.	Tree	This structure corresponds to a rooted labeled tree, that is, there is one root, nodes are labeled, and edges are directed away from the root.
3.	Node	Vertex of the tree structure.
4.	Branch	Edge of the tree structure.
5.	Originator	Originator characterizes the entity at the origin of inputs driving the software application. The application is either mostly driven by human events or less specifically by system inputs (i.e., human /HW/SW inputs).
6.	Probability of Occurrence	See Table XV. **Constraints** **C2**: OP.Profile.Structure.MarkovChain.Transition ProbabilityOfOccurrence.condition.size = [1] **C3**: OP.Profile.Structure.MarkovChain.Transition ProbabilityOfOccurrence.condition.type = previous state
7.	State-Based	Structure that uses states as primitive.
8.	Markov Chain	A Markov chain is collection of random variables X_t (where the index "t" runs through 0, 1, ... and X_t represents the state of the system at index "t") having the property that, given the present, the future is independent of the past.
9.	State	A state is defined as "the values assumed at a given instant by the variables that define the characteristics" of the application under consideration or of its components [IEEE 24765].
10.	Transition	A transition is defined as a change from one state to another. A transition is typically triggered by an event.
11.	Event	See Table XV.
12.	Probabilistic Event Flow Graph (PEFG)	"An Event flow graph (EFG) is specific model of the GUI for a particular application, representing all possible sequences of events that a user can execute on that GUI. Nodes in the EFG represent events, and directed edges represent the event-flow relationship between two events." The nodes are annotated with probability tables. [Brooks and Memon 2007]
13.	Harel State Charts	Harel statecharts "extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication" [Harel 1987].
14.	Set	Set is defined as a collection of distinct entities.
15.	Entity	Entity is defined as an element of interest. For example operations, events, input data etc.

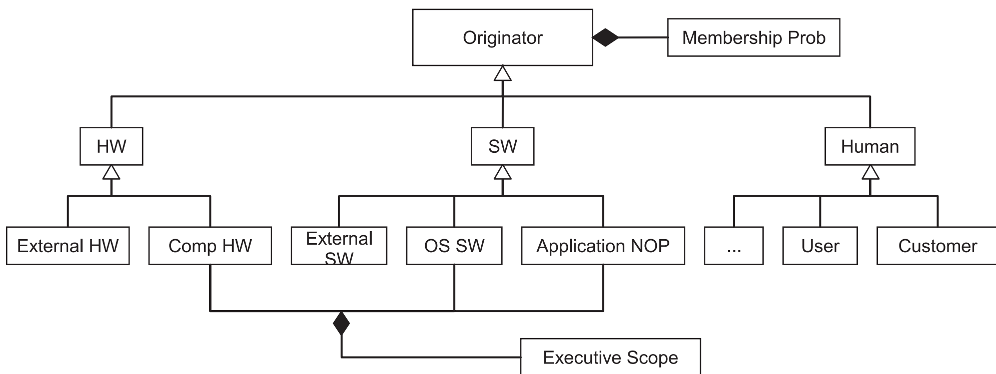

Fig. 11. Originator meta-model.

Table XVII. Originator Meta-Model

	Elements	Description
1.	Originator	See Table XVI.
2.	MembershipProbability	Probability that the originator belongs to a certain group. **Attributes**: Prob: Real The probability that the originator is a member of a certain group.
3.	Human	One or multiple human users of the application under consideration.
4.	User	“A user is a person, group, or institution that employs, not acquires, the system” [Musa 1993].
5.	Customer	A Customer is the person, group, or institution that is acquiring the system.
6.	Hardware (HW)	Hardware is a generalization of all the types of hardware’s with which the computer interfaces including the internal hardware of the computer system.
7.	Software (SW)	Software is a generalization of all the types of the software’s interacting with the computer system including the software’s installed on the computer.
8.	CompHW	See Table XIV.
9.	OS SW	See Table XIV.
10.	Application NOP	See Table XIV.
11.	Executive Scope	See Table XIV
12.	External Hardware (Ext HW)	See Table XV.
13.	External Software (Ext SW)	See Table XV.

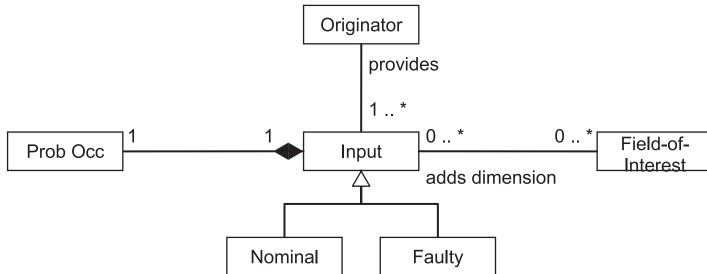

Fig. 12. Input meta-model.

Table XVIII. Input Meta-Model

	Elements	Description
1.	Input	See Table XIII.
2.	Probability of Occurrence	See Table XV.
3.	Field-of-interest	See Table XIII.
4.	Originator	See Table XVI.
5.	Nominal Input	A nominal input is an input to the application under consideration provided by an originator behaving normally.
6.	Faulty Input	A faulty input is an input to the application under consideration provided by an originator behaving abnormally/erroneously.

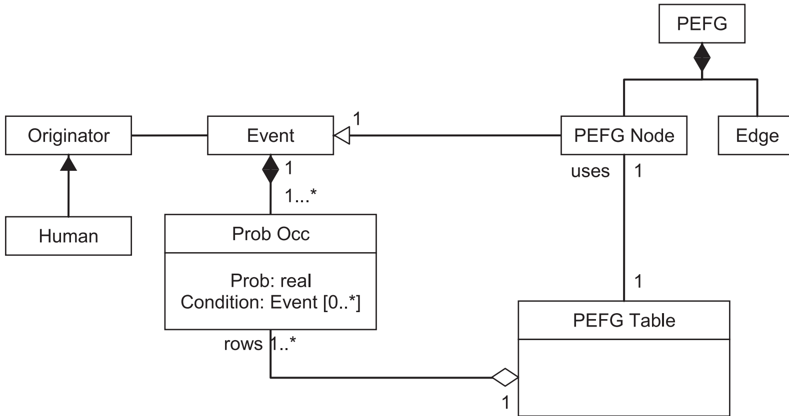

Fig. 13. Probabilistic Event Flow Graph (PEFG) meta-model.

Table XIX. Probabilistic Event Flow Graph Meta-Model

	Elements	Description
1.	PEFG	See Table XVI.
2.	Event	See Table XV.
3.	Edge	An edge in the PEFG is a directed edge used to represent an event-flow relationship. For example, an edge between e1 and e2 is used to indicate that event e2 may be invoked immediately after event e1 [Brooks and Memon 2007].
4.	PEFGNode	A PEFGNode represents an event.
5.	Originator	See Table XVI.
6.	Human	See Table XVII.
7.	PEFGTable(k)	A PEFGTable is the conditional probability table for an event **e** given subsequences of events up to length k.
8.	Probability of Occurrence	See Table XV.
		Constraints **C4**: OP.Profile.Structure.PEFG.Event ProbabilityOfOccurrence.condition.maxsize = k

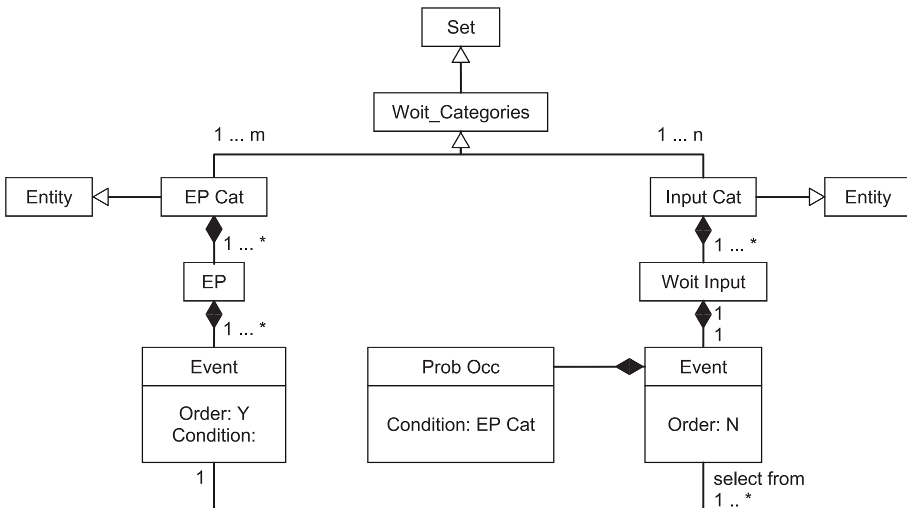

Fig. 14. Woit meta-model.

Table XX. Woit Meta-Model

	Elements	Description
1.	Set	See Table XVI.
2.	WoitCategory	A WoitCategory is defined as either an Execution Prefix Category or an Input Category.
3.	ExecutionPrefixCategory (EPCat)	An Execution prefix category is a subset of the set of prefixes such that for each EP Cat the user is able to define the probability of selecting the next input from each IC.
4.	InputCategory (IC)	An Input Category is a subset of the input domain such that for each EP Cat the user is able to define the probability of selecting the next input from each IC.
5.	ExecutionPrefix (EP)	An execution prefix is any subsequence of a module execution such that the subsequence begins with the module initialization or re-initialization.
6.	WoitInput	A WoitInput is defined as an event possibly accompanied by an argument (i.e., data).
7.	Event	See Table XV.
8.	Probability of Occurrence	See Table XV. **Contraints** **C5**: OP.Profile.Structure.WoitCategory.Event ProbabilityOfOccurrence.condition.type = EPCat **C6**: OP.Profile.Structure.WoitCategory.EPCat.EP Event.isOrdered = Y
9.	Entity	See Table XVI.

REFERENCES

S. Arora, R. B. Misra, and V. M. Kumre. 2005. Software reliability improvement through operational profile driven testing. In *Proc. of the 53rd Annual Reliability and Maintainability Symposium.* 621–627.

L. D. Bousquet, F. Ouabdesselam, and J.-L. Richier. 1998. Expressing and implementing operational profiles for reactive software validation. In *Proc. of the 9th International Symposium on Software Reliability Engineering (ISSRE'98).* 222–230.

P. A. Brooks and A. M. Memon. 2007. Automated GUI testing guided by usage profiles. In *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07).* 333–342.

K. Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. Sage, Thousand Oaks, CA.

J. Corbin and A. Strauss. 2008. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory* 3rd Ed. Sage, Thousand Oaks, CA.

D. Cramer. 1997. *Basic Statistics for Social Research: Step-by-Step Calculations and Computer Techniques Using Minitab*. Psychology Press.

C. Csallner and Y. Smaragdakis. 2004. JCrasher: An automatic robustness tester for Java. *Software Practice and Experience* 34, 11, 1025–1050.

W. Dulz and F. Zhen. 2003. MaTeLo—Statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3. In *Proc. of the 3rd International Conference on Quality Software.* 336–342.

S. Elbaum and S. Narla. 2001. A methodology for operational profile refinement. In *Proc. of the 2001 Annual Reliability and Maintainability Symposium (RAMS'01).* 142–149.

D. F. Fabio, A. Carlomusto, A. Petrillo, and A. Ramondo. 2012. Human reliability analysis: A review of the state of the art. *International Journal of Research in Management and Technology*, 2, 1.

M. Gittens, H. Lutfiyya, and M. Bauer. 2004. An extended operational profile model. In *Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE'04).* 314–325.

B. G. Glaser and A. L. Strauss. 2009. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, New York.

K. Gwet. 2002. Kappa statistic is not satisfactory for assessing the extent of agreement between raters. *Statistical Methods for Inter-rater Reliability Assessment* 1–6.

H. L. Guen and T. Thelin. 2003. Practical experiences with statistical usage testing. In *Proc. of the 11th Annual Workshop on Software Test and Reliability Estimation Process at Software Technology and Engineering Practice (STEP'03)*. 87–93.

H. L. Guen, R. Marie, and T. Thelin. 2004. Reliability estimation for statistical usage testing using Markov chains. In *Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*. 54–65.

D. Hamlet, D. Mason, and D. Woit. 2001. Theory of software reliability based on components. In *Proc. of the 23rd International Conference on Software Engineering (ICSE'01)*. 361–370.

J. M. Hammersley and D. C. Handscomb. 1964. Monte Carlo Methods (Methuen's Monographs on Applied Probability and Statistics). Methuen, London.

D. Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, 231–274.

B. Huang, M. Rodríguez, M. Li, J. Bernstein, and C. Smidts. 2011. Hardware error likelihood induced by the operation of software. *IEEE Transactions on Reliability* 60, 3, 622–639.

B. Huang, M. Rodríguez, M. Li, and C. Smidts. 2007. On the development of fault injection profiles. In *Proc. of the 53nd Annual Reliability and Maintainability Symposium (RAMS'07)*. 226–231.

IEEE/ISO/IEC 24765. 2010. *System and Software Engineering Vocabulary*. Standards.

M. Jørgensen and M. Shepperd. 2007. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering* 33, 33–53.

B. D. Juhlin. 1992. Implementing operational profiles to measure system reliability. In *Proc. of the 3rd International Symposium on Software Reliability Engineering (ISSRE'02)*. 286–295.

B. Kitchenham. 2004. *Procedures for Performing Systematic Reviews*. Technical Report. Keele University TR/SE-0401, NICTA Technical Report 0400011T.1, Keele University.

P. Koopman and J. Devale. 1999. Comparing the robustness of POSIX operating systems. In *Proc. of the 29th IEEE International Symposium on Fault-Tolerant Computing.* 30–37.

K. S. Kumar, R. B. Misra, and N. K. Goyal. 2008. Development of fuzzy software operational profile. *International Journal of Reliability, Quality, and Safety Engineering* 15, 581–597.

T. Muhr and S. Friese. 2004. *User's Manual for ATLAS. ti 6.0.* ATLAS. ti Scientific Software Development GmbH, Berline.

J. D. Musa. 1992. The operational profile in software reliability engineering: An overview. In *Proc. of the 3rd International Symposium on Software Reliability Engineering (ISSRE'92)*. 140–154.

J. D. Musa. 1993. Operational profiles in software-reliability engineering. *IEEE Software* 10, 2, 14–32.

F. Ouabdesselam and I. Parissis. 1995. Constructing operational profiles for synchronous critical software. In *Proc. of the 6th International Symposium on Software Reliability Engineering (ISSRE'95)*. 286–293.

K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. 2008. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*. 71–80.

S. J. Prowell. 2003. JUMBL: A tool for model-based statistical testing. In *Proc. of the 36th Hawaii International Conference on System Sciences (HICSS'03)*.

M. Popovic and J. Kovacevic. 2007. A statistical approach to model-based robustness testing. In *Proc. of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. 485–494.

A. Rosenberg and E. Binkowski. 2004. Augmenting the kappa statistic to determine interannotator reliability for multiply labeled data points. In *Proc. of HLT-NAACL 2004: Short Papers.* Association for Computational Linguistics, 77–80.

P. Runeson and C. Wohlin. 1993. Statistical usage testing for software reliability certification and control. In *Proc. of the 1st European International Conference on Software Testing, Analysis and Review (EuroSTAR'93)*. 309–323

J. Saldana. 2012. *The Coding Manual for Qualitative Researchers (No. 14)*. Sage.

B. Shneideman and C. Plaisant. 2010. *Designing the User Interface: Strategies for Effective Human Interaction*. Addison-Wesley.

R. Shukla, D. Carrington, and P. Strooper. 2004a. Systematic operational profile development for software components. In *Proc. of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*. 528–537.

R. Y. Shukla, P. A. Strooper, and D. A. Carrington. 2004b. A framework for reliability assessment of software components. In *Proc. of the 7th International Symposium on Component-based Software Engineering (CBSE7)*. 272–279.

J. Sim and C. C. Wright. 2005. The kappa statistic in reliability studies: use, interpretation, and sample size requirements. *Physical Therapy* 85, 3, 257–268.

C. Taylor, T. C. Gibbs and A. Lewins. 2005. *Quality of Qualitative Analysis*. Retrieved from http://onlineqda.hud.ac.uk/Intro_QDA/qualitative_analysis.php.

G. H. Walton, J. H. Poore, and C. J. Trammell. 1995. Statistical testing of software based on a usage model. *Software—Practice and Experience* 25, 1, 97–108.

Y. Wei, M. Rodríguez, and C. Smidts. 2010. PRA framework for software propagation analysis of failures. *Journal of Risk and Reliability* 224, 2, 113–135.

E. J. Weyuker. 1998. Testing component-based software: A cautionary tale. *IEEE Software* 15, 5, 54–59.

J. A. Whittaker and J. Voas. 2000. Toward a more reliable theory of software reliability. *IEEE Computer* 33, 12, 36–42.

J. A. Whittaker and J. H. Poore. 1993. Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology.* 2, 1, 93–106.

J. A. Whittaker. and M. G. Thomason. 1994. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering* 20, 10, 812–824.

D. M. Woit. 1993. Specifying Operational Profiles for Modules. In *Proc. of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'93)*. 2–10.

J. Yan, J. Wang, and H. W. Chen. 2004. Automatic generation of Markov chain usage models from real-time software UML Models. In *Proc. of the 4th International Conference on Quality Software (QSIC'04)*. 22–31.

F. Zhen and C. Peng. 2004. A system test methodology based on the Markov chain usage model. In *Proc. of the 8th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2004)*. 160–165.

Received January 2011; revised July 2013; accepted August 2013