Towards Understanding the Behavior of Classes Using Probabilistic Models of Program Inputs

Arbi Bouchoucha, Houari Sahraoui, and Pierre L'Ecuyer

DIRO, Université de Montréal, Canada
{bouchoar,sahraouh,lecuyer}@iro.umontreal.ca

Abstract. We propose an approach to characterize the behavior of classes using dynamic coupling distributions. To this end, we propose a general framework for modeling execution possibilities of a program by defining a probabilistic model over the inputs that drive the program. Because specifying inputs determines a particular execution, this model defines implicitly a probability distribution over the set of executions, and also over the coupling values calculated from them. Our approach is illustrated through two case studies representing two categories of programs. In the first case, the number of inputs is fixed (batch and command line programs) whereas, in the second case, the number of inputs is variable (interactive programs).

Keywords: Class role, dependency analysis, program behavior, Monte-Carlo simulation, probabilistic model.

1 Introduction

Program comprehension is an essential phase in software maintenance [7]. To implement new changes, software engineers have to acquire abstract knowledge on, among others, the program structure and the behavior, and the relationships among its elements [4]. This abstract knowledge helps relating a program implementation to conceptual knowledge about the application domain and, hence locates the elements affected by a change request. Understanding a complex program is similar to exploring a large city [1]. In both cases, having good maps (abstractions) facilitates considerably the comprehension. For cities, there is a good knowledge on what kind of useful information should be abstracted on maps such as streets, transportation indications, landmarks, etc. Landmarks (monuments, important buildings, train stations), for example, are used as references to quickly situate secondary elements. For software comprehension, the idea of landmarks was also used. Indeed in [11] and [13], key classes are identified to serve as starting points for program comprehension.

The identification of comprehension starting points is often based on coupling [11,13]. The rationale behind this decision is that elements that are tightly

[1] Leon Moonen, Building a Better Map: Wayfinding in Software Systems. Keynote talk, ICPC 2011.

V. Cortellessa and D. Varró (Eds.): FASE 2013, LNCS 7793, pp. 99–113, 2013.
© Springer-Verlag Berlin Heidelberg 2013

coupled to other elements are likely to implement the most important concepts of a program. Coupling can be estimated from a static analysis of the source code, independently of any execution, *i.e., static coupling*. However, this method could significantly over or under-estimate the coupling due to dynamic features such as polymorphism or dynamic class loading [1].

On the other hand, actual coupling between software elements could be captured at run time by a dynamic analysis, *i.e.,* considering what actually happens during the execution [2,12]. Thus, different executions of the same program usually lead to different values of the *dynamic coupling*. But then, from which execution(s) should the metric be computed? To circumvent the generalization issue, Arisholm et al. [2], pick an arbitrary set of executions and take the average of the coupling value over these executions. Similarly, Yacoub et al. [12] assign probabilities to a finite set of execution scenarios, compute the dynamic coupling for each scenario, and take the weighted average across scenarios as their final measure, where the weights are the probabilities. This represents the mathematical expectation of the metric under a probabilistic model where the number of possible realizations is finite. Such derivation methods certainly make sense if the set of chosen executions (and weights in the case of [12]) are representative of the variety of executions likely to be encountered when running the program. However, in practice, the number of possible executions is often extremely large, even infinite, and it may be difficult to directly assign a probability to each one. Moreover, perhaps more importantly, considering only a single value (or average) of coupling (static or dynamic) can hide a large amount of useful information on the variability of a class's behavior.

The purpose of this paper is to describe an approach for characterizing class behavior using dynamic coupling distributions. To this end, we propose a general framework to model execution possibilities by defining a probabilistic model over the inputs that drive the program. Because specifying the inputs determines a particular execution, this model defines implicitly a probability distribution over the set of executions, and also over the set of coupling values. In such a model, the distribution of the coupling values is in general too complicated to be closely approximated numerically, but it can be estimated via Monte Carlo simulation. Our approach is illustrated through two case studies representing two categories of programs. In the first case, the number of inputs is fixed (batch and command line programs) whereas, in the second case, the number of inputs is variable (interactive programs).

The remainder of this paper is organized as follows. In Section 2, we define how the probabilistic model is used to derive class coupling distributions over the executions. Then, Section 3 explains how a coupling distribution can be used to give insights of a class's behavior. In section 4, we illustrate our approach using two case studies corresponding to two categories of programs. Our approach is discussed and contrasted with the related work in Section 5. Finally, concluding remarks are given in Section 6.

2 Portraying Class Coupling

2.1 Approach Overview

When using deterministic algorithms, computer programs are driven by a set of external inputs that normally determine the entire execution sequence. We consider a computer program made up of several classes, say in Java for example, and a dynamic coupling metric at the class level whose value depends on the realizations of the input variables given to the program (see Figure 1). The total number of possibilities for these inputs (and then for the possible executions) is typically much too large to allow an explicit enumeration. Here we propose to define a probability distribution over the space of possible inputs. Once this distribution is determined, it can be used to generate representative sets of inputs. The coupling values corresponding to these inputs represent then the basis for the estimation of the coupling distribution of a class over the possible executions.

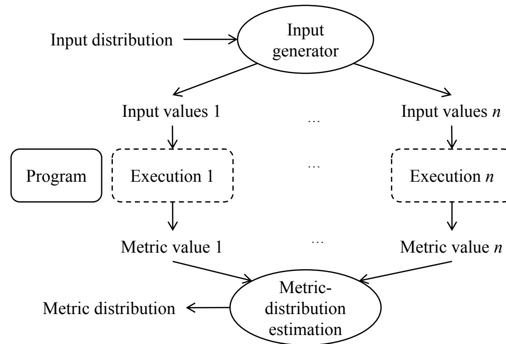


Fig. 1. Approach overview

2.2 Probabilistic Models for Input Data Generation

We consider two cases for how these input variables are defined or specified depending on the nature of programs:

(a) The number of inputs is fixed to d (a positive integer) and these inputs are represented by a random vector $\mathbf{X} = (X_1, \ldots, X_d)$. This case of inputs is found generally in batch and command-line programs where a set of parameters are specified for each execution.

(b) The number of inputs is variable (random) and the successive inputs can be seen as functions of the successive states of a Markov chain. This is the case of programs with interactions with a user or with the outside environment, where the probability distribution of the next input (and whether there is a next input or not) often depends on the values of the inputs that have already been given to the program so far.

Inputs Defined as a Random Vector. In the case of a random vector $\mathbf{X} = (X_1, \ldots, X_d) \in \mathbb{R}^d$, the input values can be given to the program before it starts its execution. In this model, we assume that $\mathbf{X}$ is a random vector with an arbitrary multivariate distribution over $\mathbb{R}^d$. This distribution could be discrete, continuous, or mixed, in the sense that for example some coordinates of $\mathbf{X}$ might have a normal or exponential distribution while others might take only integer values, or perhaps only binary values (0 and 1). The inputs, *i.e.,* coordinates of $\mathbf{X}$, are not assumed to be independent in general, but the situation where they are independent is a possibility; this is the simplest special case.

The multivariate random vector $\mathbf{X}$ has distribution F if for any $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$, we have $F(\mathbf{x}) = \mathbb{P}[\mathbf{X} \leq \mathbf{x}] = \mathbb{P}[X_1 \leq x_1, \ldots, X_d \leq x_d]$. The jth marginal distribution function is defined by $F_j(x_j) = \mathbb{P}[X_j \leq x_j]$. The random variables $X_1, \ldots, X_d$ are *independent* if and only if $F(X_1, \ldots, X_d) = F_1(x_1) \ldots F_d(x_d)$ for all $\mathbf{x} \in \mathbb{R}^d$. When $X_1, \ldots, X_d$ are not independent, a general way of specifying their joint (multivariate) distribution is via a copula [3].

A copula consists in specifying first a d-dimensional distribution whose marginals are uniform over the interval $(0, 1)$, but not independent in general. This distribution is called a *copula* (or *dependence function*). If $\mathbf{U} = (U_1, \ldots, U_d)$ denotes a random variable having this distribution, then for each j, we define

$$X_j = F_j^{-1}(U_j) = \inf\{x : F_j(x) \geq U_j\}.$$

The vector $\mathbf{X} = (X_1, \ldots, X_d)$ then has a multivariate distribution with the required marginals F_j, and a dependence structure determined by the choice of copula. That is, the marginal distributions are specified separately from the marginals. It is well known that any multivariate distribution can be specified in this way. Specific techniques for selecting a copula and generating random vectors from it are explained in [3,6], for example.

This case of input variables is found generally in batch and command-line programs where a set of parameters are specified for each execution. For example, when running *lpr* command in Linux, one should specify the printer name, the username, the number of copies, etc. When a parameter is not specified, *e.g.*, the printer name, this does not mean that one input is missing. It simply means that the default value will be used, here the default printer. Thus, the size of the input vector is always the same.

Inputs Modeled by a Markov Chain. The majority of programs nowadays do not have a fixed number of inputs, but the number of inputs (and their types) are themselves random variables. Consider for example, a program with a set of functions, each performed in a number of steps. Each step requires a certain number of parameters. The state of the execution (function and step) impacts the probability that a particular parameter is required, and if yes, the probability that this parameter takes a specific value. This is particularly true for programs that interact with a user or with the outside environment, where the probability distribution of the next input (and whether there is a next input or not) often depends on the values of the inputs that have already been given to the program

so far. In this type of situation, the input process can be modeled naturally as a discrete-time Markov chain.

A naive way of specifying such a discrete-time Markov chain model would be to assume that $\{X_j, j \geq 0\}$ is a Markov chain over the set of real numbers (or a subset thereof), where X_j represents the jth input to the program. However, this is not realistic, because the next input X_{j+1} usually depends not only on X_j, but also on the values of the previous inputs. Therefore, the process $\{X_j, j \geq 0\}$ thus defined would not be a Markov chain.

A proper way to get around this problem is to model the input process by a Markov chain $\{S_j, j \geq 0\}$ whose state S_j at step j contains more information than just X_j. Such a Markov chain can be defined by a stochastic recurrence of the form $S_j = \gamma_j(S_{j-1}, X_j)$, for some transition functions γ_j, and the jth input X_j is assumed to have a probability distribution that depends on S_{j-1}, and to be independent of $X_0, \ldots, X_{j-1}$ conditional on S_{j-1}. This assumption ensures that $\{S_j, j \geq 0\}$ is a Markov chain, which means that whenever we know the state S_j, knowing also $S_0, \ldots, S_{j-1}$ brings no additional useful information for predicting the behavior of any X_ℓ or S_ℓ for $\ell > j$. We also assume that this Markov chain has a random stopping time τ defined as the first time when the chain hits a given set of states Δ: $\tau = \inf\{j \geq 0 : S_j \in \Delta\}$. This τ represents the (random) number of inputs that the program requires.

Another interesting example where Markov chains are used to model the inputs of a software is described in [14]. In this work, the interaction with a web site is defined as a set of mouse clicks on links corresponding to the URLs of Web-site pages. The probability that a particular page is accessed depends on the other pages already accessed, *i.e.,* previous inputs. The Markov-chain model is used to measure the navigability of Web sites.

2.3 Dynamic-Coupling Distribution Estimation

Case of Random Vector. A *dynamic metric* φ can be seen as a function that assigns a real number to any possible execution of the program. But since the realized execution depends only on the realization of $\mathbf{X}$, we can view the metric as a function of $\mathbf{X}$, and write $\varphi : \mathbb{R}^d \to \mathbb{R}$. Then, $Y = \varphi(\mathbf{X})$ is a real-valued random variable whose distribution depends on the distribution of $\mathbf{X}$, perhaps in a complicated way. Thus, the distribution of Y will not be known explicitly in general. However, we can use Monte Carlo simulation to estimate this distribution. It consists in generating n independent realizations of $\mathbf{X}$, say $\mathbf{X}_1, \ldots, \mathbf{X}_n$, and then computing the n corresponding realizations of Y, say $Y_1, \ldots, Y_n$. Then the empirical distribution of $Y_1, \ldots, Y_n$ is used to estimate the true distribution of Y. As a byproduct, it permits one to estimate certain summary characteristics of this distribution, such as the mean, the variance, etc., and to compute *confidence intervals* on these numbers [9].

For example, one can estimate $\mu = \mathbb{E}[Y]$, the mean of Y, by the sample average $\bar{Y}_n = \sum_{i=1}^{n} Y_i$. To assess the accuracy of this estimator, one can also compute a confidence interval on μ, which is a random interval of the form $[I_1, I_2]$ where I_1 and I_2 are two random borders defined so that

$\mathbb{P}[I_1 \leq \mu \leq I_2] \approx 1 - \alpha$ where $1 - \alpha$ is a preselected confidence level. For example, if we assume that $\bar{Y}_n$ has a normal distribution (which is practically never exactly true but can be a good approximation when n is large, thanks to the central limit theorem), then the confidence interval has the form

$$[\bar{Y}_n - z_{1-\alpha/2} S_n/\sqrt{n},\ \bar{Y}_n + z_{1-\alpha/2} S_n/\sqrt{n}] \tag{1}$$

where S_n is the sample standard deviation of $Y_1, \ldots, Y_n$ and $z_{1-\alpha/2}$ satisfies $\mathbb{P}[Z \leq z_{1-\alpha/2}] = 1 - \alpha/2$, where Z is a standard normal random variable. Other techniques, such as bootstrap methods, for example, can be used when we think that the distribution of $\bar{Y}_n$ might not be close to normal. Confidence intervals on other quantities than the mean (for example, the variance of Y, or the correlation between two different metrics), can be computed in similar ways.

Of course, the whole empirical distribution itself always conveys more information for behavior understanding than the estimates of any of these statistics. For this reason, it is generally better in our opinion to study this distribution (for example in the form of a histogram) rather than (or in addition to) interprete, say, the average $\bar{Y}_n = (Y_1 + \cdots + Y_n)/n$ together with a confidence interval on the mathematical expectation $\mathbb{E}[Y]$.

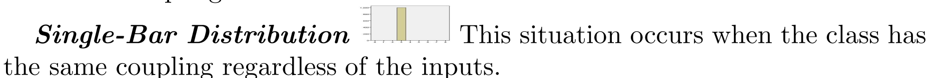
Case of Markov Chain. A dynamic metric here is defined as a function φ which assigns a real number $Y = \varphi(S_0, S_1, \ldots, S_\tau, \tau) \in \mathbb{R}$ to each realization $(S_0, S_1, \ldots, S_\tau, \tau)$.

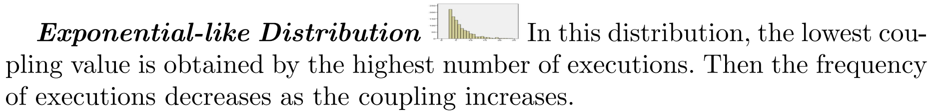
Again, if the Markov chain model is fully specified, we can simulate it and estimate the distribution of Y by the empirical distribution of n independent realizations $Y_1, \ldots, Y_n$, in the same way as in the random vector case.

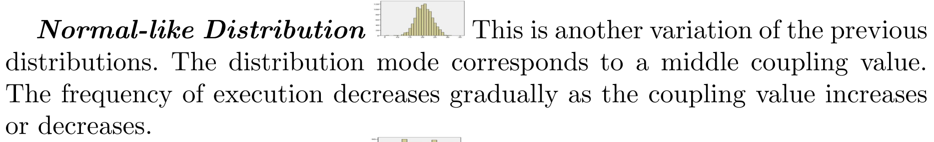
2.4 Examples of Coupling Distributions

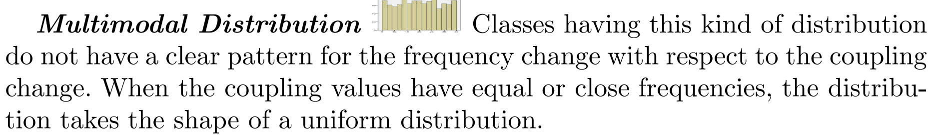
When considering dynamic coupling, different executions, corresponding to different inputs, could lead to different interactions between the considered class and the other elements of a program. Consequently, each execution results in a particular coupling value. After performing a relatively large sample of executions defined by the distribution of the inputs, those executions could be grouped according to their coupling values, which defines a distribution.

We conjecture that there is a causality chain between the input, the behavior, and the coupling value. Indeed, the input values may impact the execution control flow, *i.e.*, the class's behavior, which may lead to variations in the interactions between objects, and then in the class's coupling value. Note that variations in the interactions do not necessarily mean changes in the coupling value. The same value could be the result of different interaction sets. In the following paragraphs, we show examples of regular distributions that could be obtained. We discuss them in the basis of our conjecture. The relationship between the coupling values and the behavior will be discussed in Section 3.

Single-Bar Distribution  This situation occurs when the class has the same coupling regardless of the inputs.

Exponential-like Distribution  In this distribution, the lowest coupling value is obtained by the highest number of executions. Then the frequency of executions decreases as the coupling increases.

Normal-like Distribution  This is another variation of the previous distributions. The distribution mode corresponds to a middle coupling value. The frequency of execution decreases gradually as the coupling value increases or decreases.

Multimodal Distribution  Classes having this kind of distribution do not have a clear pattern for the frequency change with respect to the coupling change. When the coupling values have equal or close frequencies, the distribution takes the shape of a uniform distribution.

3 Understanding a Class's Behavior

3.1 Class Behavior

In the object-oriented programming paradigm, objects interact together in order to achieve different functionalities of a given program. Typically, the behavior of a program corresponds to the set of the implemented use cases. Each use case could have different scenarios depending on the inputs. Consider, for example, the use case of borrowing books in a library loan management system. The frequent and main scenario is to identify the reader, check his record, and register the loan. This scenario could be extended (*extend* relationship) by renewing the membership prior to the loan, for example, or truncated if the borrower exceeds the allowed number of books or if the concerned book cannot be borrowed (violation of use-case scenario *preconditions*).

These variations at the program level are reflected at the class level. As classes implement services that contribute to use cases. An alternative use-case scenario could require an additional service, do not use a service or use a variation of a service with respect to the main scenario. For example, when the loan is not registered, class *Book* will not decrease the number of available book copies. Therefore, a class could offer one or more services, each with possible variations. For a particular use case, the main scenario is the most frequently executed which in turn define a main and frequent contribution of a given class in this scenario. This contribution could vary according to the use case alternative scenarios.

3.2 Relating Coupling Distributions to Class Behavior

As mentioned earlier, we conjecture that there is a causality chain between the inputs, the class's behavior and the class's dynamic coupling. According to our probabilistic setting defined in Section 2, a set of representative executions is defined by the set of representative inputs. The execution sample produces a practical coupling distribution for each class of the program. The goal of our work is to use this distribution to understand the behavior of a class. Understanding a class's behavior in our setting means that we could identify the main

behavior and its variations by looking at its coupling distribution. Relating the coupling distribution to the behavior is only valid if we accept the three following hypothesis:

H_1***: the same behavior results in the same coupling value and conversely***: A class that executes one or many services in different executions corresponding to the same use-case scenarios will produce the same, or very close, coupling values. Conversely, two equal or very close coupling values corresponding to two executions indicate that these executions are likely to trigger the same class behavior.

H_2***: an extended behavior generates more or equal coupling than the original one***: When an alternative scenario AS extends a main scenario MS, the coupling value corresponding to AS is at least equal to the one of MS. This means that (almost) all the interactions in MS remain in AS and that the extension could add new interactions.

H_3***: a truncated behavior generates less or equal coupling than the original one***: When an alternative scenario AS is executed because of a precondition violation of the main scenario, the coupling value corresponding to AS is at most equal to the one of MS. This could be explained by the fact that most of the behavior of MS is not performed, which may cancel many interactions.

The above-mentioned hypotheses could be assessed automatically for any studied system. It is possible to check if executions belonging to the same block, *i.e.*, having the same coupling value, trigger the same set of method calls [2]. It is also possible to evaluate the similarity between executions belonging to contiguous blocks, corresponding to two successive coupling values. In the following paragraphs, we propose intra and inter-block similarity measures to group executions by behavior based on coupling.

Intra-block similarity or internal similarity is measured by evaluating the diversity of method calls inside the block. Formally, for a block b of executions having a coupling value c_b, the internal similarity is defined as $IS(b) = c_b/n_b$, where n_b is the number of different method calls observed in all the executions of b. The ideal situation ($IS(b) = 1$) is that all the executions in b trigger the same set of method calls. In that case, $c_b = n_b$. The more the values are close to 1, the more we consider that executions reflect the same behavior. If we assume that executions belonging to the same block concern the same behavior, the next step is to identify if contiguous blocks have the same behavior, thus forming a behavior region, or if an important modification is observed. Modifications include transitions from a truncated scenario to the main scenario within the same use case, main-scenario extension, and use case change.

Inter-block similarity or external similarity allows to measure the difference in behavior between execution blocks. In a first step, it is important to identify methods calls that are relevant in an execution block to exclude marginal calls that represent non significant behavior variations. Relevant method calls are

[2] For the sake of simplicity, we consider in this section that coupling between classes is the total of different afferent and efferent method calls. The similarity measures of the following paragraphs could be easily adapted to other dynamic coupling measures.

those that appear in the majority of the executions of a block b. A method call a is said to be relevant for a block b if it appears in at least n percent of the executions in b. n is a threshold parameter, usually set to 50% (half of the block executions). Once the set $Rel(b)$ of relevant method calls are identified for each block b, the following step is to determine the behavior regions by comparing contiguous blocks recursively. The first block b_1 (the one with the lowest coupling value) is automatically included in the first region R_1. Then for each block $b_i, i > 1$, we evaluate its similarity with the region R_j containing the previous blocks. If the similarity is above a given threshold value, then b_i is assigned to the same region R_j, if not, it forms a new region R_{j+1}. External similarity is calculated as follows:

$$ES(R_j, b_i) = \frac{|Rel(R_j) \cap Rel(b_i)|}{|Rel(R_j) \cup Rel(b_i)|} \quad (2)$$

where $Rel(R_j) = \bigcap_{b_k \in R_j} Rel(b_k)$.

When relating the coupling distributions to the behavior, the distribution examples given in Section 2.4, could be used as behavioral patterns. *Single-Bar* distribution defines an ***Assembly-chain*** pattern as the concerned class behaves in the same way regardless of the inputs. *Exponential-like* distribution is seen as a ***Clerk*** pattern. Like a clerk in an office, the class has one common behavior (main scenario of a use case), and this common behavior is gradually extended to deal with exceptional situations (alternative scenarios with extensions within the same use case). A third pattern ***plumber*** corresponds to the *Normal-like* distribution. Like for the *Clerk*, the common behavior is extended in some cases, but like for a plumber, some interventions do not require to perform this common behavior. This situation occurs when the main use-case scenario has preconditions that, for some executions, are not satisfied, which results in a truncated behavior, and then a lower coupling. Finally, we view a *Multimodal* distribution as a ***Secretary*** pattern. Classes having this kind of distribution are generally involved in different use cases. The choice of the behavior depends on the inputs, *e.g.*, utility classes. When the use cases have equal probabilities to be performed, the coupling distribution is uniform-like.

4 Illustrative Case Studies

4.1 General Setting and Implementation

To illustrate our approach, we present in this section the cases of two small Java programs: *Sudoku* (13 classes) and *Elevator* (eight classes) having inputs that are modeled by respectively a random vector and a Markov chain. For each program, we built a probabilistic input model according to the framework of Section 2. To simulate the inputs from the obtained input models, we have used the Java library *SSJ* [5], which stands for *Stochastic Simulation in Java*, and provides a wide range of facilities for Monte Carlo simulation in general. For each program, our simulation generated the input data for a sample of 1000 executions. For

each execution, the inputs were given using a class *Robot* that simulates the interaction with the GUI, and a trace was produced using the tool *JTracert*[3]. These traces were then used to calculate the class coupling metrics. For the sake of generality, we considered a different coupling metric for each program, *IC_OM*(*c*) (*Import Coupling of a class c for Objects with distinct Methods*) for Sudoku and *IC_CM*(*c*) (*Import Coupling of a class c for Classes with distinct Methods*). The definitions of these dynamic metrics are given in [2]. *SSJ* was also used to produce the distribution histograms.

4.2 Case 1: Sudoku Grid Generator

System Description and Input Probabilistic Model. Sudoku grid generator has 10 inputs. Nine of them are positions in the grid (fixed spots) where the digits 1 to 9 should be placed. We assume a 9×9 grid with cells numbered from 1 (top-left) to 81 (bottom-right). The tenth input is the level of difficulty of the grid to be generated, in a scale from 1 to 5. Starting from the first nine inputs, the program generates a grid by filling the remaining 72 cells to produce a correct solution if one exists. If not, it displays a message indicating that no solution was found for this input. When a solution is found, the level of difficulty is used to determine the number of cells to hide when displaying the puzzle.

With respect to our framework, the input vector is $\mathbf{X} = (X_1, \ldots, X_{10})$ where $X_1, \ldots, X_9 \in \{1, 2, \ldots, 81\}$, with $X_i \neq X_j$ whenever $i \neq j$, and $X_{10} \in \{1, 2, \ldots, 5\}$. Given this space of inputs, the number of possible executions (or realizations of the input vector) is $N = 81 \times 80 \times \ldots \times 73 \times 5 \approx 4.73354887 \times 10^{17}$. In our model, we assume that each of those input vector realizations has the same probability $1/N$. This means that the positions of the 9 fixed digits are selected at random, uniformly and independently, without replacement (so they are all distinct), and that the level of difficulty is also selected at random, uniformly and independently of the other inputs. The distribution of the input vector is then $F(X_1, \ldots, X_{10}) = F_1(x_1) \ldots F_{10}(x_{10})$. All $F_1(x_1), \ldots, F_9(x_9)$ are considered as uniform. We can reasonably consider that the five levels of difficulty have also equal chances to be selected ($F_{10}(x_{10})$ is a uniform distribution).

Distributions and Interpretation. Table 1 summarizes the main findings related to the coupling distributions. The first observation is that for all classes of the Sudoku program, the similarity in behavior is very high between executions producing the same coupling value (average IS per class ranging from 82% to 100%). This indicates that our first hypothesis about the correlation between the coupling and the behavior is true in this case. Regarding the distributions, we found that the 13 classes are instances of the patterns described in Section 3. All exponential-like distributions include three regions, except for *GridConfiguration* with two regions. In all cases, the first, which contains the higher number of executions corresponds to the main scenario of a use case and the other regions highlight successive extensions. For example, the first region of the class *Solver*

[3] `http://code.google.com/p/jtracert/`

Table 1. Statistics about *Sudoku* program.

Class Name	IS(%)	Distribution	Regions
Solver	88.71	*Exponential-like*	$\{b_5, b_6, b_7\}$, $\{b_8, b_9, b_{11}\}$, $\{b_{12}, b_{13}, b_{14}\}$
Sudoku (main)	96.38	*Exponential-like*	$\{b_4, b_5\}$, $\{b_6, b_7, b_8\}$, $\{b_9, b_{10}\}$
Util	100	*Uniform-like*	$\{b_1\}$, $\{b_2\}$, $\{b_3\}$, $\{b_4\}$
Valid	90.66	*Normal-like*	$\{b_1, b_2\}$, $\{b_3, b_4, b_5, b_6\}$, $\{b_7, b_8\}$
GridGenerator	90.67	*Exponential-like*	$\{b_4\}$, $\{b_5, b_6, b_7\}$, $\{b_8, b_9, b_{11}\}$
GridFrame	100	*Single Bar*	$\{b_3\}$
GridConfiguration	85.20	*Exponential-like*	$\{b_4\}$, $\{b_5, b_6\}$
Grid	93.42	*Normal-like*	$\{b_1\}$, $\{b_2, b_3, b_4\}$, $\{b_5\}$
Case	87.25	*Uniform-like*	$\{b_1, b_2, b_3\}$
BoxPanel	100	*Single Bar*	$\{b_1\}$
ButtonPanel	82.66	*Exponential-like*	$\{b_2\}$, $\{b_3, b_4, b_5\}$
InitSquare	100	*Single Bar*	$\{b_3\}$
GridPanel	88.27	*Single Bar*	$\{b_3, b_4\}$

corresponds to the standard process of searching for a solution when the user initializes the grid and asks for a valid solution (see Figure 2-left). This behavior is very frequent because in most of the cases, a solution is easy to find. The second region corresponds to an extended behavior of the first one. Indeed, when a solution is difficult to find or may not exist, the solver uses another resolution strategy based on back-tracking, and then calls other methods, especially those of the class *Valid*. The third region that we identified for the *Solver* class, corresponds to a surprisingly less-frequent case where the solver checks if an existing solution is unique or not. This task requires that from the *Solver* class to use new interactions with classes *Sudoku*, *Grid* and *Valid*. The two normal-like distributions also have three regions with the first (lower values) indicating a truncated behavior due to precondition violations, the second (middle values) representing the common behavior, and the third (highest values) including common behavior extensions. The two uniform-like distributions were not similar in behavior. Whereas *Util* with four different coupling values, defines clearly four different services (regions) with almost equal frequencies, *Case*, with three coupling values includes only marginal variations of the same behavior which lead to a single region. Finally, single-bar distributions have a unique coupling value defining a unique behavior. The only case with two coupling values exhibits unbalanced frequencies but with similar behavior.

4.3 Case 2 : Elevator System

System Description and Input Probabilistic Model. The second program that we considered in this study is a simple elevator operating system. To run the program, the user has to give the number of elevators and the number of floors. Then it is possible to enter as many times as desired the events calling elevators from a particular floor to go up or down, and selecting the destination floors. The program ends when the user enters the event stop. Obviously, the

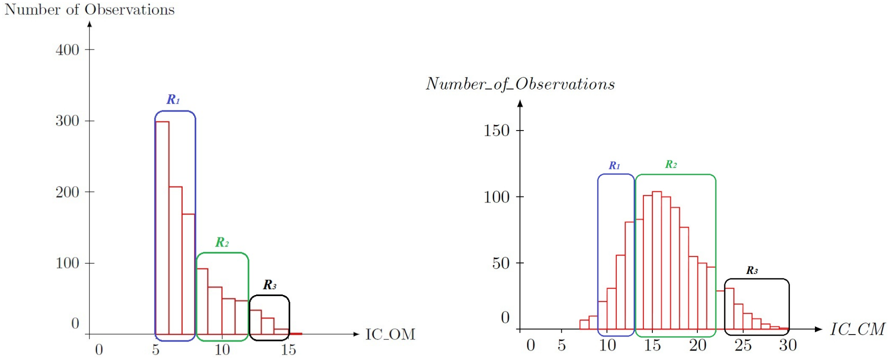


Fig. 2. Coupling distributions of classes *Solver* (left) and *ElevatorGroup* (right)

number of inputs cannot be fixed a priori. In that case, we model the program inputs by a Markov chain. We consider that there is an arbitrary number of subjects that use the elevators. Subject arrivals are random and mutually independent. Times between two successive arrivals are independent and identically distributed, and exponential with mean $1/\lambda$. In our simulation program, we took $\lambda = 1/2$. Each subject is modeled by a Markov chain that is triggered when he enters the building, and is stopped when he leaves it. The subjects' Markov chains are independent. A transition probability matrix is assigned to each subject. It defines the probabilities to travel between pairs of floors or to stay within the same floor. We also supposed that these Markov chains are homogenous, (i.e. the transition matrix doesn't change over time).

Distributions and Interpretation. Like for the first case, the average of internal similarities of all the classes are above 83% as shown in Table 2. Seven over the eight classes have average similarities of 93% or more. This confirms once again our intuition about the correlation between the coupling and the behavior uniformity. For the distributions, we observed some differences with the first case. One of them is a non regular distribution observed for the class *ArrivalSensor*. This distribution includes three blocks corresponding to three small coupling values, respectively 3 (for 20% of the executions), 4 (for 40%), and 5 (for 40%). This distribution could be considered, to some extent, as a single-bar one before the behavior in the three blocks is very similar. However, as we do not have a block that contains a clearly majority of executions, we classified it as "non regular". Another difference is that except for this marginal case, we did not observe single-bar distributions. This could be explained by the fact that the considered program is more complex than the one of Sudoku. This complexity introduces many variations in behavior.

The most frequent distribution is the exponential-like one, found for half of the classes. For example, class *Elevator* has three regions. The first region with the lower coupling values corresponds to the common elevator behavior with

Table 2. Statistics about *Elevator* program

Class Name	IS(%)	Distribution	Regions
ArrivalSensor	83.72	-	$\{b_3, b_4, b_5\}$
Elevator (main)	95.24	*Exponential-like*	$\{b_9, b_{10}, b_{11}, b_{12}\}$, $\{b_{13}, b_{14}, b_{15}, b_{16}, b_{17}\}$, $\{b_{18}, b_{19}, b_{21}, b_{23}, b_{24}\}$
ElevatorGroup	95.67	*Normal-like*	$\{b_9, b_{10}, b_{11}, b_{12}\}$, $\{b_{13}, b_{14}, b_{15}, b_{16}, b_{17}, b_{18}, b_{20}, b_{21}\}$, $\{b_{23}, b_{24}, b_{25}, b_{26}, b_{29}\}$
ElevatorControl	93.90	*Exponential-like*	$\{b_4, b_5, b_6, b_7\}$, $\{b_8, b_9, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}\}$, $\{b_{15}, b_{16}\}$
ElevatorInterface	96.57	*Normal-like*	$\{b_{10}, b_{12}, b_{13}, b_{14}\}$, $\{b_{15}, b_{16}, b_{18}, b_{19}, b_{20}, b_{22}, b_{23}, b_{24}\}$, $\{b_{25}, b_{26}, b_{27}, b_{28}\}$
Floor	98.11	*Exponential-like*	$\{b_{12}, b_{13}, b_{14}\}$, $\{b_{15}, b_{16}, b_{17}, b_{19}, b_{20}\}$, $\{b_{21}, b_{23}, b_{25}, b_{27}\}$
FloorControl	96.24	*Uniform-like*	$\{b_6, b_8, b_{10}\}$ $\{b_{12}, b_{14}, b_{16}\}$, $\{b_{18}, b_{20}, b_{22}\}$, $\{b_{24}, b_{26}, b_{28}\}$
FloorInterface	92.99	*Exponential-like*	$\{b_2, b_3, b_4\}$, $\{b_5, b_6, b_7, b_8\}$, $\{b_9, b_{11}\}$

interactions mainly with *Floor* and *ArrivalSensor*. The coupling increases to deal with exceptional situations such as considering a new elevator call during the movement, and less frequently to manage a high level of calls, which requires to create a queue and to start a new thread to manage the behavior concurrency.

The three remaining distributions are normal-like (two classes) and uniform-like ones. To illustrate how two distributions could be impacted by the same variation in a use case, consider the distribution of class *ElevatorGroup* (normal-like) shown in Figure 2-right. We identified three regions and three blocks that were not included in any region. This is a variation of our algorithm that consists in not creating regions for single blocks that have low similarities with previous and following blocks together with a very low number of executions. The first region corresponds to a minimal behavior that results from the violation of preconditions during the creating of the elevators. Indeed, during the creation, *ElevatorGroup* checks if the number of elevators and the number of floors are within a certain range. Then, it checks if the number of elevators is consistent with the number of floors. The second region that we identified corresponds to the common behavior of assigning calls to elevators, etc. Finally, the third region corresponds to an exceptional situation related to one of the class *Elevator* in the previous paragraph. This situation concerns the management of busy periods with a high number of calls.

5 Related Work and Discussion

The work proposed in this paper crosscuts several research areas. Compared to contributions in dynamic coupling calculations, our approach allows to select a representative set of executions. Indeed, Arisholm et al. [2] pick an arbitrary set of executions and take the average of the metric value over these executions. Yacoub et al. [12] assign probabilities to a finite set of execution scenarios, compute

the dynamic metric for each scenario, and take the weighted average across scenarios. Although assigning probabilities to executions is close to our approach, in practice, the number of possible executions is extremely large, which limits the applicability.

Our work defines probabilistic models for inputs that are used to generate representative samples of executions and then to better characterize the dynamic coupling. Stochastic simulation was used in software engineering, mainly to understand the development process (*e.g.*, [8]) or to characterize the evolution of a given program (*e.g.*, [10]). In both cases, the simulation is related to requirements and change request, but does not involve program inputs or class dependencies. The work of Zhou et al. [14] is maybe more or less closely related to our contribution. They propose a navigation model that abstracts the user Web surfing behavior as a Markov model. This model is used to quantify navigability. Modeling inputs as a Markov chain seems natural here because the inputs for Web sites are different from ones of classical software. Indeed, in this work, only mouse clicks on links are considered but not inputs using forms.

The work presented in this paper is an initial initiative to propose a framework for understanding the relationship between the coupling and the behavior of a class. Although the first findings are very encouraging, there are many open issues that need to be addressed. Firstly, in practice, it is difficult to define input distributions. When the program is in use, it is possible to record the inputs as the users provide them and after a certain period, estimate the distribution from the collected data. However, when the program is under development, *i.e.,* not released yet, these data are not available. Of course, one could decide theoretically that an input should have a particular distribution (say normal). Still, there is a need for estimating the distribution parameters (mean and variance in the case of normal distribution). Another problem concerns the nature of the input data. In our study, we considered inputs that take values in a finite set. In most of the programs, inputs could be strings with theoretically an infinite set of values such as person names, files, etc. The random generation of strings according to a particular distribution could be modeled easily. However, random generation of files, such as source code for compilers, is not an obvious task.

6 Conclusion

In this paper, we proposed a framework for modeling program inputs using a probabilistic setting. These models allow to derive class coupling metric distributions. We showed how these distributions could be used to understand the behavior of classes. We illustrated our approach with two small case studies. The first has a finite set of inputs, which are modeled by a random vector. In contrary, the second program has an infinite set of inputs that are modeled by a (homogenous) Markov chain. We observed in these cases that recurrent distribution patterns correspond to regular behavior schemes.

Our future work will be mainly dedicated to make our framework more effective. The issues to be addressed include scalability and support for input model

definition. We additionally intend to assess more dynamic coupling metrics, so that to improve the generalizability of our approach.

References

1. Allier, S., Vaucher, S., Dufour, B., Sahraoui, H.A.: Deriving coupling metrics from call graphs. In: Int. Work. Conf. on Source Code Analysis and Manipulation, pp. 43–52 (2010)
2. Arisholm, E., Briand, L.C., Foyen, A.: Dynamic coupling measurement for object-oriented software. IEEE Trans. Softw. Eng. 30(8), 491–506 (2004)
3. Asmussen, S., Glynn, P.W.: Stochastic Simulation. Springer (2007)
4. Biggerstaff, T., Mitbander, B., Webster, D.: The concept assignment problem in program understanding. In: Int. Conf. on Software Engineering, pp. 482–498 (1993)
5. L'Ecuyer, P.: SSJ: A Java Library for Stochastic Simulation (2008), Software user's guide, available at http://www.iro.umontreal.ca/~lecuyer
6. Nelsen, R.B.: An Introduction to Copulas. Lecture Notes in Statistics, vol. 139. Springer (1999)
7. Rajlich, V., Wilde, N.: The role of concepts in program comprehension. In: 10th International Workshop on Program Comprehension, pp. 271–278 (2002)
8. Setamanit, S., Wakeland, W., Raffo, D.: Planning and improving global software development process using simulation. In: Int. Workshop on Global Software Development for the Practitioner (2006)
9. Shao, J.: Mathematical Statistics. Springer (1999)
10. Stopford, B., Counsell, S.: A framework for the simulation of structural software evolution. ACM Trans. Model. Comput. Simul. 18(4), 1–36 (2008)
11. Tahvildar, L., Kontogiannis, K.: Improving design quality using meta-pattern transformations: a metric-based approach. J. Softw. Maint. Evol. 16(4-5), 331–361 (2004)
12. Yacoub, S.M., Ammar, H.H., Robinson, T.: Dynamic metrics for object oriented designs. In: METRICS 1999, pp. 50–61 (1999)
13. Zaidman, A., Demeyer, S.: Automatic identification of key classes in a software system using webmining techniques. J. Softw. Maint. Evol. 20(6), 387–417 (2008)
14. Zhou, Y., Leung, H., Winoto, P.: Mnav: A markov model-based web site navigability measure. IEEE Trans. Softw. Eng. 33(12), 869–890 (2007)