Reliability Analysis in Symbolic Pathfinder

Antonio Filieri[1]
Institute of Software Technology
University of Stuttgart
Stuttgart, Germany
antonio.filieri@informatik.uni-stuttgart.de

Corina S. Păsăreanu
Carnegie Mellon Silicon Valley, NASA Ames
Moffet Field, CA, USA
corina.s.pasareanu@nasa.gov

Willem Visser
Stellenbosch University
Stellenbosch, South Africa
wvisser@cs.sun.ac.za

***Abstract*—Software reliability analysis tackles the problem of predicting the failure probability of software. Most of the current approaches base reliability analysis on architectural abstractions useful at early stages of design, but not directly applicable to source code. In this paper we propose a general methodology that exploit symbolic execution of source code for extracting failure and success paths to be used for probabilistic reliability assessment against relevant usage scenarios. Under the assumption of finite and countable input domains, we provide an efficient implementation based on Symbolic PathFinder that supports the analysis of sequential and parallel programs, even with structured data types, at the desired level of confidence. The tool has been validated on both NASA prototypes and other test cases showing a promising applicability scope.**

I. Introduction

Pervasiveness of software systems in critical applications is stressing the need for methodologies and tools to do reliability assessment. With the term reliability we generically refer to the probability of the software to successfully accomplish its assigned task when requested by the user [1]. In reality most of the software we use daily is defective in some way, though it can most of the time do its job. Indeed, the presence of a defect in the code may never be realized if the input does not activate the fault [2]. For this reason, even incorrect software may be quite reliable under specific usage profiles.

Most of the available approaches for software reliability analysis are based on formal models derived by architectural abstractions [3], [4]. To deal with code, black-box [5] or some ad-hoc reverse engineering approaches have been proposed, e.g. [6]. Model-driven techniques have also been used to keep design models synchronized with the implementation [4], though their application scope is usually limited to specific domains, e.g. embedded systems.

In this paper we propose the systematic and fully automated use of symbolic execution to extract logical models of failure and successful execution paths directly from source code. In the past decade the use of symbolic execution as a means to analyze programs have steadily increased and it is now routinely used to find errors in code and to generate tests, see for example [7], [8], [9]. In this work we show how to take the path conditions generated by these tools and use them to estimate the reliability of the implemented software.

[1]Part of this work has been done while the author was with the DEEPSE group - DEI - Politecnico di Milano, Milan, Italy.

A path condition is a set of constraints on the inputs that, if satisfied by the input values, will allow the execution to follow the specific path through the code. We label each (terminating) execution path with either *success* or *failure*. Since the set of path conditions produced by symbolic execution is a complete partition of the input domain, given a probability distribution on the inputs we can compute the reliability of the software as the probability of satisfying any of the successful path conditions. The probability distribution on the input formalizes the expected *usage profile* that accounts for all the external interactions of the software, both with the user and with external resources such as remote components.

To account for non-termination in presence of loops we use bounded symbolic execution. In this case interrupted paths will be modeled by path conditions labeled as *grey*. For an input satisfying a grey path condition we cannot predict success nor failure. The resulting uncertainty will be used to define a precise confidence measure to assess the impact of the execution bounds and the consequent reliability prediction.

Although our approach can be customized for any symbolic execution system, we focus on Symbolic PathFinder (SPF) [9] that works at the Java byte code level. As failures, we can consider the typical errors reported by SPF (e.g. assert violations, null-pointer exceptions, race violations or deadlocks) but also more general properties of interest, not necessarily related to low-level bugs present in the code (which presumably could be corrected before code release). For example, in Section VIII-A we describe the reliability analysis for a NASA software component that monitors for flight-rule violations during flight. The analyzed code has no software bugs, but we characterize as failure the paths that lead to mission abort due to flight-rule violations.

We focus on programs whose inputs range over finite discrete domains. This restriction allows us to make use of model counting procedures for general and efficient algorithms to compute probabilities. Our implementation supports linear integer arithmetic (which is the most common theory handled by symbolic execution tools), complex data-structures, loops and also concurrency. For multi-threaded programs the actual reliability depends both on the usage scenario and on the scheduling policy. In this case we identify the best and worst schedule for a given usage profile, that lead respectively to the highest and lowest reliability achievable for that usage.

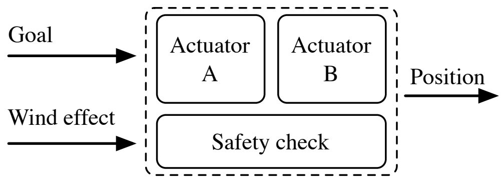

Fig. 1. Running example structure.

The main contributions of the paper are: (1) calculating reliability for source code, (2) a confidence measure for results computed from a bounded analysis, (3) support for multi-threading, (4) support for complex data-structures, (5) an efficient implementation and (6) an evaluation of the approach on NASA's Onboard Abort Executive.

In the rest of the paper we first define a running example that we will use throughout the paper and in Section II we give some background on symbolic execution and probability theory. In the following three sections we will show how to calculate reliability for non-looping (III), looping (IV) and multi-threaded programs (V) each time elaborating on the running example to illustrate the techniques. In Section VI we then show how reliability can be computed when symbolic data-structures are given as input followed by details of the implementation of our general approach (VII). The paper finishes with the validation of the reliability calculations on NASA's Onboard Abort Executive (VIII-A) and the example of a broken Binary Tree container (VIII-B), followed by related work (IX) and conclusions (X).

A. Example Analysis

We illustrate our reliability analysis on an example modeling a simplified flap controller of an aircraft. The controller is composed of two actuators and a safety check to avoid overrun of the flap. The user sets the goal position for the flap and the actuators move toward it from the current position. The actuation power of each controller is hindered by the effect of the wind, which could push on both head or tail. The high level structure of the flap controller is shown in Figure 1.

The position of the flap is identified by an integer value between 0 and 15. Goal is an integer value that states the next flap position and is provided by the user (we assume that any goal position can be requested with the same probability). The wind effect represents the action of the wind and could be any integer between -15 and 15, depending on wind direction.

An actuation step of Actuator A moves the position of the flap toward the goal position by 10 units; Actuator B by 1 unit. We will refer to 10 and 1 as the *actuator strengths* of A and B, respectively. The actual movement depends also on the wind effect which is added to the effect of the actuator. Safety check represents an invariant to be verified after each actuation. The invariant asserts that the value of the output position is actually between 0 and 15.

We will describe parts of this system along with the presentation of reliability analysis for different control structures and for multi-threading. In the latter case, we will show how concurrency can be managed with schedules leading to better or worse reliability.

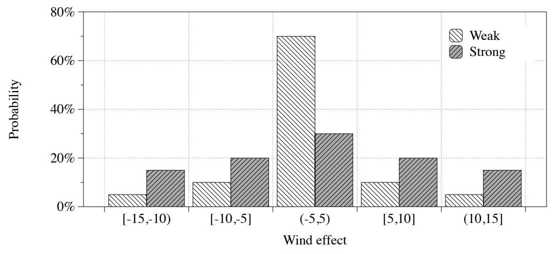

Fig. 2. Wind effect profiles.

The effect of wind is uncertain and will be profiled through a random variable resembling specific operation scenarios. Figure 2 shows the two profiles we will use. The ranges of values for the wind effect are reported on the x-axis, and the corresponding vertical bars represent the probability of a value in the range to be inserted as input. In this example, we assume symmetric distributions for the two profiles. Weak wind is more concentrated around zero, while strong wind is more likely to produce extreme values. The analysis will estimate the mission success probability of our toy system in the two different wind conditions.

II. Background

In this section we recall some basic notions concerning symbolic execution [10] and probability theory [11]. Further extensions will be provided in subsequent sections as they are needed. For simplicity, in this section we will assume all the variables in our program to be integers from a finite domain and all the conditions appearing in a branch or a loop statement to be expressed as linear constraints. We will later show how to relax these assumptions to make our framework more general. We will focus here on symbolic execution for Java, as performed by Symbolic PathFinder (SPF).

A. Symbolic Execution

Symbolic execution is an extension of normal execution in which the semantics of the basic operators of a language is extended to accept symbolic inputs and to produce symbolic formulas as output [10]. The behavior of a program P is determined by the values of its inputs. Such a behavior can be defined through a state transition systems where the execution of an instruction identifies the next reachable states.

Definition 1 (State of a program)**.** *The state s of a sequential program is defined by the tuple (IP, V, PC) where:*

- *IP represents the next instruction to be executed.*
- *V is a mapping from each variable v_i of the program to its symbolic value, i.e. a symbolic expression.*
- *PC is the* path condition. *PC is a boolean expression over the symbolic inputs σ_i.A path condition is a conjunct of terms plus the constants true and false $= \neg$true as short form for a tautology and an unsatisfiable condition.*

The current state s and the next instruction IP define the set of transitions from s. Without going into the details of every Java instruction, we informally define these transitions depending on the type of instruction pointed to by IP.

Assignment. At bytecode level we consider an assignment as setting the value of the area of memory corresponding to the left hand side variable $v_i \in V$, while at source code an assignment requires also to evaluate the right hand side statement. The execution of an assignment leads to a new state where *IP* is incremented to point to the next instruction and V is updated to map v_i to its new symbolic value. *PC* does not change.

Branch. The evaluation of an *if-then-else* instruction (on condition c) introduces two new transitions. The first leads to the state s_1 where IP_1 points to the first instruction of the *then* block and $PC_1 = PC \wedge c$. The second leads to a state s_2 where IP_2 points to the first instruction of the *else* block and $PC_2 = PC \wedge \neg c$. If the PC associated with a branch is not satisfiable, symbolic execution will not follow the branch.

Loop. A *while* loop is unfolded by SPF until its condition evaluates to false or the exploration depth limit is reached[1]. Analogous transformations are applied to the other loop constructs.

Method invocation. Method invocations are managed by means of macro expansion, i.e. the execution jumps to the invoked subprogram and executes it, making the necessary assignments to represent method's arguments. The state s^1 will have the same *PC*, IP^1 will point to the first instruction of the method and V^1 will contain now the new assignments to the formal parameters of the method.

The initial state of a program is $s_0 = (IP_0, V_0, PC_0)$, where IP_0 points to the first instruction of the main method, V_0 contains the arguments of main, if any, and $PC_0 = true$. A program may have also one or more terminal states. They represent terminal condition of the program such as the exit of the program or an uncaught exception making the program terminate abruptly.

In Section V we will extend the definition of symbolic execution to the case of multi-threading.

B. Probability Theory

In this section we recall some fundamental notions of probability theory for finite spaces. For an extensive exposition, the interested reader could refer, for example, to [11].

The possible outcomes of an experiment are called *elementary events*. For example, the rolling of a 6-sided dice may produce the elementary events 1,2,3,4,5, and 6. Elementary events have to be *atomic*, i.e. the occurrence of one of them excludes the occurrence of any other. The set of all elementary events is called a *sample space*. A set of elementary events is called an *event*. In this paper, we consider only finite and countable sample spaces, meaning that the underlying set of elementary events is countable and finite.

Definition 2 (Probability distribution)**.** *Let S be the sample space of an experiment. A* probability distribution *on S is any function*

$$Pr : \mathscr{P}(S) \to [0,1] \cap \mathbb{R}$$

that satisfies the following conditions (probability axioms):

- $Pr(\{x\}) \geq 0$ *for every elementary event x*
- $Pr(S) = 1$
- $Pr(A \cup B) = Pr(A) + Pr(B)$ *for all events* $A, B \subseteq S$ *with* $A \cap B = \emptyset$

The pair (S, Pr) *constitutes a* probability space.

Definition 3 (Conditional probability)**.** *Let* (S, Pr) *be a probability space. Let A and B be events* $(A, B \subseteq S)$*, and let* $Pr(B) \neq 0$.

The conditional probability *of the event A given that the event B occurs is:*

$$Pr(A|B) = \frac{Pr(A \cap B)}{Pr(B)}$$

$Pr(A|B)$ *is also referred to as* probability of A given B.

Given the definition of conditional probability, the following law holds:

Definition 4 (Law of total probability)**.** *Let* (S, Pr) *be a probability space and* $\{B_n : n = 1, 2, 3, \ldots\}$ *be a finite partition of S. Then, for any event A:*

$$Pr(A) = \sum_n Pr(A|B_i) \cdot Pr(B_i)$$

The law of total probability can also be stated for conditional probabilities:

$$Pr(A|X) = \sum_n Pr(A|X \cap B_i) \cdot Pr(B_i|X)$$

where B_i are defined as in Definition 4 and X does not invalidate the assumptions of Definition 3.

III. Computing Reliability

In this paper we consider reliability (*Rel*) as the probability that the program accomplishes its execution without hitting any failure, under specific usage assumptions. A failure could be any observable error for SPF, such as a failed assertion or an uncaught exception. Since SPF operates at Java bytecode level, the definition of failure events can be very detailed and flexible.

The main flow of our analysis chain is shown in Fig. 3. The input of the process is the Java source code. This is symbolically executed by SPF, whose output is a set of *path conditions* (PCs); a path condition is a set of constraints on the program inputs whose satisfaction leads either to the occurrence of a failure event or to success (i.e. termination without failures).

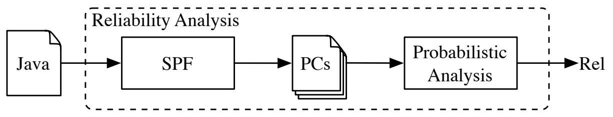

Fig. 3. Reliability analysis chain.

Let us assume for now that the symbolic execution of the program always terminates (we will relax this assumption in the next section). We classify the PCs produced by

[1]The choice of an exploration depth limit and its refinement is further investigated in Section IV

SPF in the two sets $PC^s = \{PC^s_1, PC^s_2, \ldots, PC^s_m\}$ and $PC^f = \{PC^f_1, PC^f_2, \ldots, PC^f_p\}$ according to the fact that they lead to success or failure, respectively.

Note that all path conditions identified by SPF define disjoint input sets and, because of the termination assumption, they cover the whole input domain. Therefore, the path conditions define a complete partition of the input domain [10], [9].

We further assume that all the input variables range over finite discrete domains, whose joining is generically indicated as D. We profile the expected usage for the program through a *usage profile* (UP). UP is a set of pairs $\langle c_i, p_i \rangle$ where c_i is a *usage scenario* defined as a (constraint representing a) subset of D and p_i ($p_i \geq 0$) is the probability that a user input belongs to c_i. We further require, for simplicity, $\{c_i\}$ to be a complete partition of D, and thus $\sum_i p_i = 1$ (checked automatically by our implementation). Intuitively, UP is the distribution over the input space. Notice that c_i could contain even a single element of D, allowing for the finest grained specifications of UP.

Throughout the paper we will use constraints and the sets characterized by them interchangeably. For example if the input domain of variable x is $\{0,1,2,3\}$, we may use the constraints characterizing this set $x \geq 0 \wedge x \leq 3$.

Given the output of SPF, reliability can be redefined as the probability of executing the program (P) with an input satisfying any of the successful path conditions, given the usage profile UP. This definition can be formalized as:

$$Rel = Pr^s(P) = \sum_i Pr(PC^s_i \mid UP) \tag{1}$$

An analogous definition can be provided for failure probability $Pr^f(P)$ and it is straightforward to prove that $Pr^s(P) + Pr^f(P) = 1$.

A. Computing Reliability using Model Counting

Let us now look more closely at how to compute the actual value of Rel (or, conversely, the failure probability of P). Since UP defines a partition of the input domain, from the law of total probability [11]:

$$Pr(PC \mid UP) = \sum_i Pr(PC \mid c_i) \cdot p_i \tag{2}$$

Furthermore, from the definition of conditional probability: $Pr(PC|c_i) = Pr(PC \wedge c_i)/Pr(c_i)$. In order to use model-counting techniques (e.g. [12], [13]) for the computation of the conditional probabilities, let us define for a constraint c the function $\sharp(c)$ that returns the number of elements of D satisfying c. $\sharp(\cdot)$ is always a finite non negative integer because we assumed D finite and countable. Under this same assumption, $Pr(c)$ is, by definition [11], $\sharp(c)/\sharp(D)$ (where $\sharp(D)$ is the size of the domain that we implicitly assumed not null).

Applying the same argument to the combination of equations (1) and (2) we obtain:

$$\begin{aligned} Rel = Pr^s(P) &= \sum_i Pr(PC^s_i \mid UP) = \\ &= \sum_i \sum_j Pr(PC^s_i \mid c_j) \cdot p_j = \sum_i \sum_j \frac{\sharp(PC^s_i \wedge c_j)}{\sharp(c_j)} \cdot p_j \end{aligned} \tag{3}$$

The implementation of $\sharp(\cdot)$ depends on D, that is on the types of the input variables, and will be described in Section VII, with some insight on its computational complexity.

B. Example

From our example of Section I-A, let us consider the safety check. After every step of an Actuator, the position of the flap is updated from its previous value adding the effects of the actuator and the wind. After each update, the current position is checked against the safety invariant, as in Listing 1.

Listing 1. Safety invariant.

```
flapPosition = flapPosition + actuatorEffect +
    windEffect;
if (flapPosition > MAX_POSITION || flapPosition
    < MIN_POSITION) {
    throw new OverrunException();
}
```

The probability of raising an OverrunException depends on both the actuator strength and the wind profile. Let us assume that only actuator B (strength 1) operates a single action according to listing 1. In case of *weak wind* (cf. Fig. 2), the reliability of the system is 0.60197132; in case of *strong wind* it drops down to 0.51881720. If instead of actuator B we use actuator A (strength 10), the reliability for weak and strong wind is 0.79247312 and 0.52204301, respectively. Notice that none of our actuators can converge to the goal in a single step for all the values of wind effect, but the stronger actuator is more effective in keeping the flap position in a safe range even for stronger wind (according to the invariant). In sections IV-A and V-A we will first allow a single actuator to operate as many actions as needed, and then we will study the complete system involving the concurrent use of the two actuators.

IV. Looping Constructs

In the previous section we assumed the termination of symbolic execution. In general, the presence of loop constructs may lead to infinite computation and requires convenient analysis strategies. The solution provided by SPF is based on *bounded symbolic execution* [9]: a bound is set for the exploration depth (i.e. the number of transitions executed); when the bound is reached the exploration backtracks. In this setting the symbolic execution is no longer complete and, besides *success* and *failure* paths, a new set of paths is collected for executions interrupted before reaching an error or completing the run. We call this set of paths *grey* and label the corresponding path conditions as $PC^g_1, PC^g_2, \ldots, PC^g_q$. For the set of grey path conditions it is possible to define $Pr^g(P)$ analogously to the other sets:

$$Pr^g(P) = \sum_i Pr(PC^g_i | UP) \tag{4}$$

The three sets $\{PC^s\}$, $\{PC^f\}$, and $\{PC^g\}$ are disjoint, and constitute a complete partition of the entire domain D [10]. Hence it is straightforward to prove that $Pr^s(P)+Pr^f(P)+Pr^g(P)=1$. The intuitive meaning of $Pr^g(P)$ is to quantify the ratio of elements of D for which neither success nor failure have been revealed at the current exploration depth. This information is a measure of the confidence we can put on our reliability estimation:

$$\mathit{Confidence} = 1 - Pr^g(P)$$

$\mathit{Confidence} = 1$ means that the symbolic execution is complete, i.e. for each element of D we can state if it leads to a success or a failure. Smaller values of *Confidence* may reveal a too small bound for the exploration depth and thus suggest the need for a deeper analysis. Confidence can thus be used for iterative refinement of symbolic execution. Indeed, this measure is by construction a non decreasing function of the exploration depth, hence the exploration depth can be increased until the desired confidence goal has been reached. Notice that, if for very large values of the exploration depth the confidence keeps a steady value it could be the bad smell of an infinite loop that should be further investigated by the designers.

A. Example

Let us now allow an Actuator to run as many actions as possible, that is until the goal comes closer that its actuation strength. The body of the loop is constituted by an actuator action, defined as in Sect. III-B:

Listing 2. Actuator loop.

```
while (abs(goalPosition - flapPosition)>=
    ACTUATOR_STRENGTH) {
    actuatorEffect = sgn(goalPosition -
        flapPosition)*ACTUATOR_STRENGTH;
    flapPosition = flapPosition + actuatorEffect
         + windEffect;
    if (flapPosition > MAX_POSITION ||
        flapPosition < MIN_POSITION) {
        throw new OverrunException();
    }
}
```

The loop in Listing 2 may not terminate because of the wind effect (i.e. a wind effect greater than the actuator strength inhibits the ability to converge to the goal). The results of the analysis using actuator A only (strength 10) with different exploration depths is shown in Table I for the profiles of the wind effect:

TABLE I
ANALYSIS CONFIDENCE.

Exploration depth	Weak wind	Strong wind
10	$Conf = 0.87580646$ $Pr^s = 0.59677419$	$Conf = 0.85806452$ $Pr^s = 0.48387096$
30	$Conf = 0.97741936$ $Pr^s = 0.65358422$	$Conf = 0.95483871$ $Pr^s = 0.54623656$
50	$Conf = 0.98440861$ $Pr^s = 0.65734767$	$Conf = 0.96881721$ $Pr^s = 0.55376344$
70	$Conf = 0.98924731$ $Pr^s = 0.65949820$	$Conf = 0.97849463$ $Pr^s = 0.55806452$

Increasing the exploration depth, the confidence grows revealing more accurate reliability predictions. Notice that, due to the possibility of infinite loops, the confidence cannot reach 1 because in some cases the symbolic execution does not terminate. Indeed, analyzing, for example, the weak profile with exploration depth 500 still leads to $Conf = 0.9892473118$ with $Pr^s = 0.6594982078853047$.

V. MULTI-THREADING

Multi-threading introduces non determinism in the choice of the next thread that could access the CPU. Different choices may affect the occurrence of failures, and thus the reliability of the program. Without making any assumption on the way the next thread is chosen (i.e. without assuming a next-choice distribution nor specific scheduling polices) we want to identify the best possible sequence of choices, that is the one leading to the highest reliability (and the worst one). The benefit of our approach is twofold: first, any possible thread scheduler will provide a reliability between the worst and the best case; second, inspection of the best case could possibly provide insights for the design of a scheduler able to improve software reliability for specific usage profiles. The latter research path is still under investigation and in this paper we will focus on the identification of the best and worst cases.

First, we need to extend the definition of symbolic execution provided in Sect. II-A by replacing IP with a set of pairs $\langle t_i, IP_i\rangle$, where each t_i identifies an active thread and IP_i represents the next instruction to be executed by thread t_i. A *schedule* σ is a sequence $t_i, t_j, \ldots, t_k$ defining the order of access to the CPU for all the active threads. Notice that for a given schedule the multi-threaded program is reduced to a sequential one whose next instruction corresponds to the next instruction of the next thread to be executed, according to the schedule. SPF allows one to symbolically execute multi-threaded programs and to produce a set of pairs $\langle \sigma_i, PC_i\rangle$ where PC_i are path conditions classifiable as *success*, *failure*, or *grey* as in the previous sections, and σ_i is the schedule associated with the specific execution.

Although for each schedule σ_i in the SPF output it is possible to analyze the corresponding PCs to compute its Pr^s, Pr^f, and Pr^g it is not always true that their sum is equal to 1. Indeed it could be the case that for some inputs the execution terminates earlier than for others. For example, a specific value of the wind effect may lead to the failure of actuator A and the abort of the system, while another may allow a further move to be performed, for example by actuator B (assuming the actuators run concurrently). Though the two schedules would be reported by SPF as different, it is the case that the first one is a prefix of the second. Generalizing, certain input values may lead to early termination of the program (either success of failure) while others may let the execution continue. Based on this observation we can record the schedules produced by (bounded) symbolic execution into a prefix tree and define the *maximal schedules*:

Definition 5 (Maximal schedule)**.** *A* maximal schedule *(Σ) is a thread schedule that is not a prefix of any other schedule in the paths reported by a (bounded) symbolic execution.*

For a maximal schedule Σ_i we can define the set of path conditions $PC^* = \cup_{\sigma_j \in pre(\Sigma_i)} PC_j$, where $pre(\Sigma_i)$ is the set of all the prefixes of Σ_i including the maximal schedule. The intuition behind the construction of PC^* is the assumption of Σ_i as the "prescribed" schedule and then the accounting for possible early terminations of the execution captured in the PCs of its prefixes. For a maximal schedule it is immediate to notice that PC^* covers the entire domain D. Indeed, either the execution has been terminated because of the exploration bound or no more instructions can be executed by SPF.

For each maximal schedule Σ_i we can now define:

$$Pr^s(P, \Sigma_i) = \sum_j Pr(PC_j^{*s} \mid UP) \qquad (5)$$

where PC^{*s} is the subset of PC^* leading to success. Analogous definitions can be stated for PC^{*f} and PC^{*g}. Since PC^* covers the entire input domain, we have that $Pr^s(P, \Sigma_i) + Pr^f(P, \Sigma_i) + Pr^g(P, \Sigma_i) = 1$ for every Σ_i.

Maximal schedules can now be ordered according to their reliability obtaining the best and worst schedules and the corresponding reliability values. In the ordering, a special role is played by the grey area. Indeed it can be seen either in an optimistic or a pessimistic way: in the former we add the value of grey to success while in the latter to failure. Notice that grey area keeps its role of measuring the confidence we can entrust in the analysis, hence the same considerations of Sect. IV hold. Finally, it could be the case that there is no single best (or worst) schedule, but more than one reach the maximum possible reliability. In this case the designer might pick the most suitable one, according to the application domain.

The main issue in dealing with multi-threading is the computational complexity of both SPF and the probability analysis because of the number of schedules to analyze is exponential in the number of the active threads. This problem can be alleviated by means of partial order reduction (POR)[14]. The idea behind POR it to exploit the commutativity of concurrently executed instructions, which result in the same state when executed in different orders. In particular, two symbolic paths that are in the same partial order have logically equivalent path conditions [15] and thus we conjecture they lead to the same probability results.

A. *Example*

Concluding our flap controller example from Sect. IV-A, let us now allow the two actuators to run in two concurrent threads. The *run* method of both of them is the same as Listing 2, with strength 10 for actuator A and 1 for actuator B. We bound the analysis at exploration depth 20 and, in order to rank the different schedules, we conservatively consider the grey area as failure. The reason for this small value for exploration depth lies in the complexity of symbolic execution. Indeed, the number of paths to be stored in memory grows exponentially with the number of threads, saturating our 4Gb availability for depth 25. The PCs collected are 5234, with SPF time of 3 minutes and probabilistic analysis of 10 seconds (thanks to the high reuse of previous computations as will be explained in Sect. VII). For the sake of space, we report the results for the weak wind profile only; the case of strong wind leads to analogous observations.

The best schedule is M,M,M,A,A,B (where M stands for the main thread) with reliability 0.14854167 and confidence 0.5154166667. We reported only the shortest schedule, all the equivalent ones show the same pattern: M, as many actions of actuator A as possible and then as many actions of B as needed. Notice that this pattern is known in control allocation theory for aeronautics systems as Daisy Chain, and is actually applied for flap control [16]. Hence, in this example, our analysis did not only provide reliability estimation but suggested also a scheduling policy for the two actuators that corresponds to the one designed by control engineers.

The worst schedule is instead, for example, $M,A,M,A,B,A,A,B,A,B,A,B,A,A,M,A$ with reliability 0.0 and confidence 0.04166666667 (this is the shortest worst case schedule). Our conservative assumption played a significant role in the ranking because of the large grey area assumed as failure. On the other hand, the worst scheduling policy in this case keeps bouncing between actuators A and B reducing the effectiveness of control. Indeed, beside the role of grey area, within 20 exploration steps the flap never reached the goal.

VI. Input Data Structures

We now describe how the reliability analysis is extended to also handle programs that take as input structured data types (e.g., lists or trees).

A. *Usage Profiles for Data Structures*

Usage profiles for data structures are defined with the help of Java predicates (i.e. boolean methods) that define data structure properties that partition the input state space. For example, for a program with an input list, the UP may specify that the input list is acyclic 90% of the time (and cyclic 10%). As before, we restrict ourselves to finite input domains, which for data structures will also limit the number of possible heap nodes in the input.

SPF can analyze programs with unbounded data structures as inputs, using *lazy initialization* [9]. The result of symbolic execution is a set of paths, each characterized by a path condition that encodes both numeric and heap constraints.

Listing 3. swapNode Example

```
class Node {
  int elem;
  Node next;

  Node swapNode() {
      if (elem*elem > next.elem) {
          Node t = next;
          next = t.next;
          t.next = this;
          return t;
      }
      return this;
   }
}
```

As an example, consider the Java code in Figure 3 that declares a class *Node* implementing linked lists. The fields *elem* and *next* represent, respectively, the node's integer value and a reference to the next node in the list. The method *swapNode* destructively updates its input list, referenced by implicit parameter *this*, according to a non-linear condition on the first two nodes. Symbolic execution results in eight symbolic paths, due to the condition and the different aliasing possibilities in the input (e.g. $PC_1 : in.elem * in.elem > in.elem \wedge in.next = in \wedge in \neq null$, $PC_2 : in.elem * in.elem) \leq in.elem \wedge in.next = in \wedge in \neq null$, etc.). Seven paths are successful and one leads to failure (null dereference for $PC_3 : in.next = null \wedge in \neq null$). There are no grey paths (since there are no loops).

B. Model Counting for Data Structures

Though the procedure described so far can be applied on any finite and countable input domain, the case of data structures deserves special attention in the definition of the counting procedure $\sharp(\cdot)$. Indeed, in the worst case a complete (and expensive) enumeration of all the possible input instances might be performed. To deal with this issue, we propose to use Korat [17], a tool that performs constraint-based generation of structurally complex test inputs for Java programs. Korat provides efficient generation, and therefore also counting, of input data structures that satisfy a complex predicate within pre-defined bounds. The predicate is written as a boolean method called *repOk*, whose body can embed any arbitrary complex computation. The finitization of the input domain is instead accomplished by specific Korat methods to specify bounds on the size of input data structures as well as on the domain of primitive fields. Thus we can encode the constraints provided by symbolic execution together with the constraints from the usage profile as a *repOK* predicate and run Korat to count the data structures that satisfy the constraints for the given finitization.

Reliability for the List example: As an example of the use of Korat, let us compute the reliability of our list program. Assume a usage profile that specifies that the input list is acyclic with probability 0.9 and it is cyclic with remaining probability 0.1. As there is only one failure symbolic path (revealed by a null pointer exception in the evaluation of the if condition), it is simpler to compute the failure probability and thus the corresponding reliability. The path condition for the failure path, as revealed by SPF, is $input \neq null \wedge input.next = null$.

Since the path condition for the failure constraint is only satisfiable for acyclic lists, we get the probability of failure $Pr^f(P)$ as:

$$0.9 \cdot \frac{\sharp(input \neq null \wedge input.next = null \wedge acyclic(input))}{\sharp(acyclic(input))}$$

Korat computes $\sharp(input \neq null \wedge input.next = null \wedge acyclic(input)) = 10$ and $\sharp(acyclic(input)) = 1111111$, for lists having up to 6 nodes and elem between 1 and 10, giving probability of failure 0.0000081. Thus, since the execution always terminates due to finitization, the probability of success is 1- 0.0000081= 0.9999919.

C. Non-linear and Floating-point Constraints

Korat can handle arbitrary numeric constraints on the fields in the input data structures, including non-linear integer constraints and constraints involving floating point numbers, as in the example above; for floating points, the finitization should include discretization criteria. Thus Korat could be used as an "universal" model counting procedure that can handle arbitrary constraints expressed as Java predicates. On the other hand, for special input domains more efficient procedures can be substituted to Korat as will be discussed in the next section.

We also note that it is the responsibility of the user to write the complex (Java) predicates in the usage profile and to ensure that these predicates are disjoint. To ease this burden we have defined patterns for some commonly used predicates (such as *acyclic* and *size* for linked lists) that can be used and modified easily. In the future we would like to explore established logics to simplify the specification task.

VII. Implementation

Reliability analysis is performed in two phases (cf. Sect III). First SPF collects path conditions leading to success, failure, and grey conditions; second the probabilistic analysis is performed. In this section we will describe the implementation of probabilistic analysis (for SPF implementation see [9]).

The purpose of the probabilistic analysis phase is to compute $Pr(A)$, where A is a set of constraints on the input variables of the program. The complexity is in terms of the number of variables and the number of constraints composing A. For large problems, in either of the two dimensions, approaching the problem as a whole could be time consuming. In this section we will show a divide and conquer strategy to improve time efficiency.

The central idea is that constraints in A identify a dependency relation (*dep*) among the constrained variables that can be formalized as follows (let x, y, and z be variables in A):

- $\forall x \; dep(x,x)$
- $\forall x,y$ if x and y appear in the same constraint, then $dep(x,y)$
- $\forall x,y,z \; dep(x,y) \wedge dep(y,z) \implies dep(x,z)$

The intuitive meaning of the *dep* relation is that if $dep(x,y)$ then the values assumed by x affect the values that can be assumed by y. For example, from $\{x > 5 \wedge y = x + 5\}$ we deduce that the value of y is affected by the values of x, and vice versa. The relation *dep* is an equivalence relation, thus it induces a partition on the set of variables appearing in A. For this reason, we can rewrite A as the conjunction of the subsets $A_{[v]}$, each of whom collects all the constraints involving a variable in the equivalence class of *dep* represented by v. Such conjuncts are logically separated, hence it can be proved that $Pr(A) = \prod_v Pr(A_{[v]})$. For example, let A be $\{x > 2 \wedge y < 5\}$, its probability can be computed as $Pr(A) = Pr(x > 2) \cdot Pr(y < 5)$.

If $A_{[v]}$ is a set of linear integer constraints, we add a further *normalization* step to remove inequalities and redundant constraints. The normal form obtained is composed by the disjunction of non overlapping constraint sets describing exactly the same event space of $A_{[v]}$. For example, $A = \{x > 2 \wedge x > 4 \wedge x < 10 \wedge x \neq 7\}$ would be transformed into the equivalent disjunction: $\{x \geq 5 \wedge x \leq 6\} \cup \{x \geq 8 \wedge x \leq 9\}$. This step allows to both *simplify* the set of constraints by removing redundancies and to (possibly) split $A_{[v]}$ into a number of smaller subsets (let us call them $A^i_{[v]}$). The sets $A^i_{[v]}$ are non overlapping by construction, hence we have that: $Pr(A_{[v]}) = \sum_i Pr(A^i_{[v]})$. This simplification is performed through the external tool Omega [18].

Model counting is now used to compute the probability value of each $A_{[v]}$ ($A^i_{[v]}$, respectively), in the assumed domain, as explained in Section III. This operation is performed through the external tool LattE [12] for integer linear constraints because of its efficiency on this special case. For more general constraints or data structures an analogous simplification can be defined and the counting procedure can be performed through Korat [17], as shown in Section VI.

The reduction of A into a set of sub-problems $A^i_{[v]}$ has three valuable benefits. First, the time complexity of LattE is polynomial in the number of variables and for Korat it is up to exponential in the same measure: each $A_{[v]}$ usually involves a subset of the variables appearing in A, speeding up the execution time of the external tools. Second, both the split from A to $A_{[v]}$ and from each of them to $A^i_{[v]}$ allow a natural parallelization in a map-reduce fashion. Third, the reduction of the constraints into sub-problems enhance the reuse of previous computations through caching of the results; indeed the same subset of constraints may appear in many path conditions, because, for example, they share the same prefix. Reuse can lead to a significant improvement in the probabilistic analysis time (see Sect. VIII). The effect of reuse is even more evident in case of multi-threading, where different schedules may lead to similar path conditions, and in case of large usage profiles, where several usage scenarios may share a large set of constraints, as will be shown in the next section.

In the worst case, simplification does not improve execution time and the elements of A have to be counted all at once. But in our experience, real software is far from the worst case and simplification leads to a significant speed-up (cf. Sect. VIII).

As a final remark notice that the two phases are completely independent. By replacing SPF with a symbolic execution engine for another language it is possible to reuse our implementation for the reliability analysis of programs written in any other language.

Our tool implementation can be donwloaded from [19], as well as the source code and usage profiles of our example.

VIII. Validation

A. On-board Abort Executive (OAE)

We applied the reliability analysis to a Java model of a NASA software component that was originally written as a prototype for the Crew Exploration Vehicle's ascent abort handling, the Onboard Abort Executive (OAE)[20]. The OAE monitors the status of the vehicle during the ascent phase of flight and it checks a set of *flight rules* that are supposed to be invariant during the ascent. Whenever a flight rule is violated the OAE decides that an abort is required and it selects the abort mode which is safest for the astronauts. The OAE receives its inputs (e.g., current altitude, launch vehicle internal pressures, etc.) from sensors and other software components. The analyzed code is approximately 1400 lines of code, it has a large input space (36 input variables) and complicated logic. We note that the component did not have any errors, so instead of computing the probability of reaching an error (or conversely the probability of the component behaving correctly) we compute the probability of an abort (or conversely the probability of mission success).

1) Domain: Some of the range restrictions on the inputs were directly provided by the domain experts, while others were determined from the simulation data used for testing the flight software. For each input, we determined the minimum (min) and maximum (max) values and we used these values to encode the ranges $[min - \delta \ldots max + \delta]$ as the domain for the analysis. We used the extra quantity δ to increase the chances that symbolic execution would analyze failure cases.

2) Usage Profile: Table II shows the results of analyzing three different usage profiles for the OAE. The first profile was to simply look at a uniform distribution of values for each variable within its domain. To see how the analysis scales we then considered a profile where for one variable (*thrust*) a Gaussian (normal) distribution of values was used and lastly a case where a Gaussian distribution for two variables (*thrust* and *tank_pressure*) was used. The Gaussians were produced by discretizing the ranges into 5 segments for *thrust* and 4 for *tank_pressure*, which led to 5 and 20 usage profile constraints for respectively the one and two variable cases.

3) Results: The analysis was run on a Red Hat Linux 64bit machine with 3.7Gb of memory and a 2.8Ghz Intel i7 CPU.

From the results in Table II one should first notice that since we run the reliability analysis after symbolic execution, the total paths ("Paths") and the paths that lead to an abort ("Aborts") doesn't change for different profiles. Similarly the time for symbolic execution ("SPF") is essentially the same for all runs (the small variance is natural in Java). Since the number of profile constraints increase between the three analyses (1, 5 and 20 respectively) the number of times we need to get a model counting result also increase ("Counts") which in turn results in an increase in the time spent calculating the probabilities ("Probs"). However, notice that the time spent simplifying constraints with Omega and doing the model counting with LattE is pretty much invariant for the different profiles. This is due to the caching described in Section VII. The number of times a reliability result would need to be calculated will always be the number of paths times the number of profile constraints, however, from Section VII we know that we first split the total path constraint into independent parts, then apply Omega for simplification and only then call LattE. Caching is applied at all steps following the dependency step, thus both Omega and LattE results can be reused. Because of the nature of symbolic execution (where

TABLE II
ONBOARD ABORT EXECUTIVE PROFILES

Profile	Paths	Aborts	SPF (ms)	Counts	Probs (ms)	Omega (ms)	LattE (ms)	Reliability
Uniform (1)	3754	432	8329	27320	5991	1285	2488	0.99999998180802
1 Gaussian (5)	3754	432	8074	136600	7471	1229	2474	0.99999998180663
2 Gaussian (20)	3754	432	8176	546400	12298	1244	2537	0.99999997427198

prefixes to a path are shared) there is very high reuse of previously calculated results (although not shown it is more than 99.9% for both the Omega and LattE caches).

The actual chance of an abort happening is clearly extremely small and the code seems to be very robust with regards to the different profiles. It is interesting to note that the result of our analysis could be further used to help estimating the reliability not only of the OAE component but also of the vehicle monitored by the OAE, since we essentially compute the probability of a flight rule being violated, and the rules should be invariant for the whole flight.

B. Binary Tree

For the next set of experiments we use the Binary Tree implementation previously studied in [21] and [13]. This code has an error in the `delete` operation, that is triggered when non-leaf nodes are deleted. What we want to study here is the reliability of the code, i.e. the probability of avoiding the error, under different usage profiles. We limit the values in the container to the range $[0 \dots 9]$. The actions we can perform are `add` and `delete` of an element. Lastly, we only look at sequences of 5 actions after which we check an assertion to determine if the error was triggered. Since our sequences are of finite length, there cannot be any *grey* paths here, all paths will be either correct or they will trigger the error.

TABLE III
BINARY TREE RELIABILITY PROFILES

Actions	Values	Reliability
Uniform	Uniform	0.99459
75% Delete	Uniform	0.99888
25% Delete	Uniform	0.99174
No Delete Last	Uniform	0.99850
No Delete Last 2	Uniform	1.00000
Uniform	1% Ordered	0.99280
Uniform	30% Ordered	0.99490
Uniform	50% Ordered	0.99636
Uniform	70% Ordered	0.99781
Uniform	99% Ordered	0.99992
Uniform	100% Ordered	1.00000

The usage profile can vary in the probability of the actions being performed and the values being used. Table III shows the reliability results obtained with various different usage profiles for actions and values. Note that the examples are small and as such we do not report timing (< 20 seconds in all cases).

The first case considered is if both the actions and the values are chosen uniformly from their respective domains. The reliability result is the same as reported by the tool[2] from [13] where only uniform probabilities are considered. Next, inspired by the fact that we know the error is within the `delete` action, we consider a case where we delete 75% of the time (and therefore `add` 25% of the time) and then a case where we delete only 25% of the time. Our intuition might suggest that the former case should be more unreliable, but in fact the more one delete the more reliable the code is. The reason for this seeming anomaly is that the error is not *just* due to delete, it is also when and how much you delete, as shown by the case where we specify a profile that will never have a `delete` action in the last entry, or the last two entries, in the sequence. In the latter case, the reliability is 1.0.

[2]Available from http://probsym.googlecode.com

Next we consider adjusting the profile of the values we call `add` and `delete` with, but we keep the choice between these two actions uniform (i.e. each with 0.5 probability). Specifically we focus on how ordered the values are that we call the actions with. We consider sequences of length 5, hence if the value v_i is the parameter to call i in the sequence $(i:1..5)$, we can express the profile as (using the 30% ordered case):

```
usageProfile{
  v1<=v2 && v2<=v3 && v3<=v4 && v4<=v5  : 3/10;
!(v1<=v2 && v2<=v3 && v3<=v4 && v4<=v5) : 7/10;
}
```

The results show that the code becomes more reliable when the ordering of the values increases. In fact it will be completely reliable if the values are ordered. The reason for this is that in the ordered case, one will only delete the last value added, and the actual error in this code is only triggered when deleting a non-leaf node in the tree (and the last value added will always be at a leave node).

IX. RELATED WORK

Reliability analysis is considered a key element in the design of software systems. The high availability of monitoring data allows, nowadays, to compute valuable and accurate characterization of both the actual software behavior and the usage profile. The original approaches tended to consider the software system as a black-box and to observe its reliability from the outside [22]. Most recent trends are focused on modular or component-based systems [23] and assume an architectural-level perspective [4]. With our work we further open the view on the internal structure of the software, down to source code. Previous approaches deal with code artifacts passed through an intermediate abstract representation (e.g. as sketched in [1] or applied in [6]). A number of architectural styles can be mapped to analytical models (e.g. [24]); there have been defined parametric contracts for software architectures (e.g. [25]); the issues of error propagation and transformation in complex architectures has been analyzed ([26]). The idea to extrapolate models from code is effective for a number of purposes. Nonetheless, the coherency between models and code and among different models (e.g., concerning different measures of the same related phenomena) could be hard to maintain. Models could also add a semantic gap between the developer and the artifacts she is used to reason on. Direct syntax-driven approaches have not been so popular. In [27], a syntax-driven approach has been proposed to deal with workflows specified by structured languages. This methodology is based on attribute grammars LR parsing and exploits attribute synthesis mechanisms to compose reliability estimates provided for single tasks up through the entire workflow.

On a technical level a closely related work is that of [13] where model counting was used within an extension of SPF to calculate path probabilities. However they did not consider usage profiles and therefore cannot calculate reliability as described here; they also didn't consider multi-threading nor structures. Although our approach is cast in terms of an extension to SPF, it should be noted that it can be applied to any symbolic execution approach (e.g. PEX [8] and KLEE [7]) where we have access to a path condition, thread schedule and whether the path led to a failure, success or unknown.

Probabilistic model checking [28] is also related to what we do here, with the main difference being that we calculate probabilities for the system based on the usage profiles, whereas for probabilistic model checking the input is a system with the probabilities for each transition already provided. Another difference is that we take as input Java code, whereas probabilistic model checking tends to analyze models of real systems. An exception is [29] and [30] that takes as input Java code annotated with probabilities, but their goal is to determine the progress of the model checking and not reliability analysis.

X. Conclusions and Future Work

We presented an approach to calculate software reliability, under a specific usage profile, directly from the source code. The current implementation supports linear integer arithmetic operations using the LattE model counter and an extension using Korat to count structures is under development. Many areas for future work exist: (a) Section VI-C describes extensions to the data-structure approach, (b) more failure properties (e.g. temporal properties), (c) usage profiles for sequences where probability distributions depend on previous events, (d) partial order reduction for multi-threading, and (e) runtime analysis to derive profiles directly from running systems.

Acknowledgment

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

[1] R. Cheung, "A user-oriented software reliability model," *IEEE Trans. Soft. Eng.*, vol. SE-6, no. 2, pp. 118—125, 1980.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11—33, 2004.

[3] K. Goseva-Popstojanova, A. Mathur, and K. Trivedi, "Comparison of architecture-based software reliability models," in *ISSRE*, 2001, pp. 22—31.

[4] A. Immonen and E. Niemela, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," *Software and Systems Modeling*, vol. 7, pp. 49—65, 2008.

[5] J. Musa, "Operational profiles in software-reliability engineering," *IEEE Software*, vol. 10, no. 2, pp. 14—32, March 1993.

[6] K. Goseva-Popstojanova, M. Hamill, and R. Perugupalli, "Large empirical case study of architecture-based software reliability," in *ISSRE*, Nov 2005, pp. 52—61.

[7] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209—224.

[8] N. Tillmann and J. De Halleux, "Pex: white box test generation for .net," in *TAP*. Springer, 2008, pp. 134—153.

[9] S. Anand, C. Păsăreanu, and W. Visser, "Jpfse: A symbolic execution extension to java pathfinder," ser. LNCS, vol. 4424. Springer, 2007, pp. 134—138.

[10] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385—394, Jul 1976.

[11] W. Pestman, *Mathematical Statistics*, ser. De Gruyter Textbook. De Gruyter, 2009.

[12] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, "Effective lattice point counting in rational convex polytopes," *Journal of Symbolic Computation*, vol. 38, no. 4, pp. 1273—1302, 2004.

[13] J. Geldenhuys, M. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 166—176.

[14] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. LNCS, 1996, vol. 1032.

[15] K. Sen, "Scalable automated methods for dynamic program analysis," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2006.

[16] W. Levine, *The Control Handbook, Second Edition: Control System Applications, Second Edition*, ser. The Electrical Engineering Handbook Series. Taylor and Francis, 2009.

[17] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 123–133, Jul. 2002.

[18] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "The omega calculator and library," *College Park, MD*, vol. 20742, p. 18, 1996.

[19] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analyzer for SPF," http://filieri.dei.polimi.it/publications/2013-icseSPF/.

[20] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *ISSTA*, 2008.

[21] W. Visser, J. Geldenhuys, and M. Dwyer, "Green: Reduce, reuse and recycle constraints in program analysis," in *FSE*. ACM, 2012.

[22] H. Pham, "Recent studies in software reliability engineering," in *Handbook of Reliability Engineering*, H. Pham, Ed. Springer, 2003, pp. 285—302.

[23] M. Palviainen, A. Evesti, and E. Ovaska, "The reliability estimation, prediction and measuring of component-based software," *Journal of Systems and Software*, vol. 84, no. 6, pp. 1054—1070, 2011.

[24] W.-L. Wang, Y. Wu, and M.-H. Chen, "An architecture-based software reliability model," in *Dependable Computing, 1999. Proceedings. 1999 Pacific Rim International Symposium on*, 1999, pp. 143—150.

[25] R. Reussner, H. W. Schmidt, and I. Poernomo, "Reliability prediction for component-based software architectures," *Journal of Systems and Software*, vol. 66, no. 3, pp. 241—252, 2003.

[26] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes," in *Component-Based Software Engineering*, ser. LNCS. Springer, 2010, vol. 6092, pp. 1—20.

[27] S. Distefano, A. Filieri, C. Ghezzi, and R. Mirandola, "A compositional method for reliability analysis of workflows affected by multiple failure modes," in *CBSE*. ACM, 2011, pp. 149—158.

[28] M. Kwiatkowska, "Quantitative verification: models techniques and tools," in *ESEC-FSE*. ACM, 2007, pp. 449—458.

[29] X. Zhang and F. van Breugel, "Model checking randomized algorithms with java pathfinder," in *QEST*, sept. 2010, pp. 157—158.

[30] ——, "A progress measure for explicit-state probabilistic model-checkers," in *Automata, Languages and Programming*, ser. LNCS. Springer, 2011, vol. 6756, pp. 283—294.