A Compositional Method for Reliability Analysis of Workflows Affected by Multiple Failure Modes

Salvatore Distefano
Università di Messina
Messina, Italy
Politecnico di Milano
Milano, Italy
sdistefano@unime.it,
distefano@elet.polimi.it

Antonio Filieri
Carlo Ghezzi
Raffaela Mirandola
Politecnico di Milano
Milano, Italy
filieri,ghezzi,mirandola@elet.polimi.it

ABSTRACT

We focus on reliability analysis for systems designed as workflow-based compositions of components. Components are characterized by their failure profiles, which take into account possible multiple failure modes. A compositional calculus is provided to evaluate the failure profile of a composite system, given failure profiles of the components. The calculus is described as a syntax-driven procedure that synthesizes a workflow's failure profile. The method is viewed as a design-time aid that can help software engineers reason about system's reliability in the early stage of development. A simple case study is presented to illustrate the proposed approach.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; D.2.8 [**Software Engineering**]: Metrics

General Terms

Reliability, workflow, components, services, probability

1. INTRODUCTION AND MOTIVATIONS

Component-based software engineering identifies in the concept of component as a building block the elementary unit in software architecture modeling, assessment, development, and management. This follows from three main principles of software engineering: separation of concerns, modularization, and reuse. According to such perspective, a software system is an aggregate of (software) modules or components. This paradigm is at the basis of modern *service-oriented architectures* (SOAs), in which the components assume the form of (Web) services selected and aggregated into composed processes. In the case of Web services, compositions are implemented through a *workflow language*, like BPM ([27]) or BPEL ([1]).

In the SOA case, application components are owned (developed, deployed, maintained, and operated) by possibly different stakeholders. They are viewed by clients as black boxes. In the rest of this paper, the term *component-based software* is used to indicate a wide range of software architectures, ranging from the case in which (commercial) off-the-shelf COTS components are integrated into the application to the case of service compositions. Accordingly, the term *component* is also used to denote a service. We further assume compositions to be described by an abstract workflow language, which is later described in Section 2. The language supports a minimal set of *structured* composition patterns. Other more complex patterns may be defined in terms of the structured set.

Non-functional properties of component-based software systems are a crucial design-time concern. Architectural decisions, including selection of components and the structure of the workflow, may significantly affect the qualities of the resulting system, such as their reliability, performance, or cost. This paper focuses on *reliability*. Early assessment of reliability at design time is one of the challenges of component-based architectures and a key factor to developing dependable software.

Component-based architectures allow designers to reason on systems' reliability at a higher level of abstraction. At the modeling level, components can be viewed as black-box units. Because components are subject to reuse, we may expect data to be available on their observed behavior, such as their failure profile. Based on a model of the architecture, which describes how components are connected together and interact, we expect well-founded methods to be available to software engineers to reason about satisfaction of the global system's reliability requirements.

We consider as a *failure* an observed condition or state that shows that a system does not meet its intended objective. Referring to [3], we acknowledge that each application domain has its own concerns about what should be considered as a failure and what can be the impact of each type of failure produced by a component in the execution flow through components. We define *failure mode* a possible way in which a component, or an aggregation of components can fail.

Avizienis et al. [3] clearly describes the need to deal with *multiple different failure modes*. A single Boolean domain (failure/no failure) is not expressive enough to represent important pathological behaviors. Moreover, in the same paper the authors also stress the importance of considering the *error propagation* process among system components. Nonetheless, few modeling approaches deal with error propagation across a component-based system (*e.g.*, [11, 23, 2]) and with multiple failure modes [13]. On the other hand, to get a complete view of the possible failure pathology of the whole system, it is important to take into account that components can experience a number of different failure modes and those failures

CBSE'11, June 20–24, 2011, Boulder, Colorado, USA.

can propagate in different ways across the execution flow, possibly spreading up to the application interface.

In [13] we already explored the notion of a component failure profile, which can include information about the emergence, propagation, and transformation of errors in the execution flow of a running system, eventually leading to a failure. Our past work was founded on the use of Discrete Time Markov Chains (DTMCs), a well-known stochastic model, which proved to be be applicable to reliability modeling and analysis (e.g., [17, 25]).

In this paper instead we provide a method that allows the evaluation of the failure profile of applications designed as structured workflow-based compositions, based on the failure profiles of the individual components. In our analysis, we assume that failures may only occur in the execution of a task and may only propagate through invocations of components; that is, the internal workflow operations do not generate failures. Our method can help the software engineer evaluate the effect of components' failures on the overall reliability of the workflow; for example, whether certain failures that may be generated by a component invocation eventually manifest themselves at the workflow interface or instead they only generate internal faulty states with no visible external effect.

The method we present in this paper differs from [13] in the modeling and analysis approach. Concerning modeling, it does not need to generate a DTMC in order to accomplish reliability verification. The analysis is performed directly on the abstract workflow. This may reduce the burden for designer, who is no longer required to deal with the generation of Markov models and Probabilistic Computation Tree Logic (PCTL) [22], although the transformation may be automated. Concerning analysis, this new approach is directly focused on the computation of closed formulae by means of algebraic operations. These formulae can then be analyzed in a number of way. Also, the entire analysis process is intrinsically compositional; it recursively constructs the reliability formulae for the whole structured workflow by composing the formulae for its constituent patterns.

This paper is organized as follows. In Section 2 we provide some background concepts on component-based systems and workflows, also characterizing the problem in more formal, algebraic terms and identifying the quantities to evaluate. Then, Section 3 specifies our analysis technique through algebraic rules that apply to simple sequential composition patterns. Section 4 discusses how the parallel composition pattern may be handled. Then Section 5 presents a syntax-driven algorithm that recursively analyzes complex component-based software workflows, starting from the aggregation rules of basic sequential and parallel workflows. An example taken from literature is therefore evaluated by applying the proposed technique as described in Section 6. Then, in Section 7 we try to provide a quick overview on related work, and finally, in Section 8, some remarks of the technique and future work are discussed.

2. PRELIMINARY CONCEPTS

Component-based systems are defined by their structures and their behaviors. As we mentioned, the behavior is assumed to be described by a workflow, which specifies the flow of control among components.

The Workflow Management Coalition defines a *business process* as [31] "a set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships." A *workflow* is instead "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." An *activity* (task) is a description of a piece of work that forms one logical step within a process. In this paper, activities correspond to invocations of external components. All other internal activities to the workflow (for example, in BPEL assignments to internal variables) are simply ignored, because we assume that they do not affect reliability.

An executing workflow is called an *instance process*. In our work, we assume that different instance processes of the same workflow do not interfere with each other at run time.

Workflows are specified by combining tasks through control structures that determine admissible execution paths [37]. A survey by van Der Aalst et al. in [32] identifies and characterizes several workflow patterns. For simplicity, in this paper we restrict our analysis to the three basic patterns of structured programming (*sequence*, *branch* and *loop*) for sequential parts. It is well-known that the three sequential patterns we select allow any other pattern to be described in terms of them [6]. The *fork-join* pattern is more complex. In fact, there are different kinds of synchronization possibilities for the *join* operation. For example, we may wish to specify that we wait for just the first k out of the total number of threads launched. In this case, we also need to make assumption on the completion order of the different threads. This problem will be discussed in more general terms later in Section 4. Our control flow structures are summarized in Table 1.

To support a formal definition of workflows, in the next subsection we introduce a reference grammar covering the basic constructs. The grammar can be easily extended to enrich the language with additional workflow patterns. Then, in Section 2.2, we formalize the reliability properties we are interested in.

2.1 Workflow syntax

Table 2 presents the grammar of the structured workflow language in BNF-like form. A *workflow* is defined as a list of statements. Each statement can be a *sequence*, a *branch*, or a *loop*. The *fork-join* construct supports parallel composition. We consider a *sequence* as a non-empty list of statements. The *branch* construct contains exactly two branches, the *if* and the *else*, both containing a list of statements, and a *condition*. The condition represents the probability to take the *if* branch. Notice that the list of statements can possibly be empty, allowing to define a single-edged *branch*. The *loop* construct is characterized by a list of statements, its body, and a condition – the probability of iteration. Since conditions c are probabilities, they are represented by real numbers in the range $0 \leq c \leq 1$.

As we said, in our context a *task*, referred to by an identifier, represents the invocation of an external component. The approach, however, can be generalized by assuming a domain-dependent notion of task. In this case, it would be up to the designer to decide which are the relevant software tasks to be used in the model, both in terms of granularity and relevance. For example, several operations can be aggregated in order to reduce the complexity of the reliability view of the system; or, on the other hand, the reliability view can be kept as close as possible to the functional view to simplify coherence checks. We may also assume that certain internal operations may fail, and therefore consider them as tasks. In general, tasks correspond to the atomic units of information upon which we wish to perform reliability analysis.

2.2 Reliability properties

We assume that tasks, when invoked, either respond by returning control and yielding correct data values or they fail. We distinguish between two kinds of failures. The former occurs when the task does not return control to the invoking task. That is, the invoked

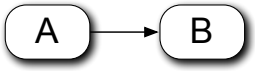
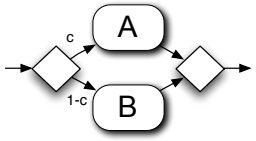
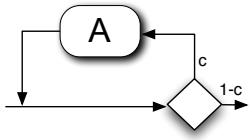
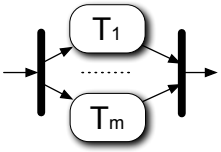
Control construct	Description	Graphical notation
Sequence	Ordered list of tasks.	
Branch	Conditional choice between two execution paths. One branch is chosen with probability *c* (the other branch is chosen with probability *1-c*).	
Loop	The body (task *A*) is iterated, at each step with probability c.	
Fork-Join	*Fork* indicates the starting point of multiple threads to be run in parallel. *Join* marks the synchronization point and can have application specific semantics.	

Table 1: Control patterns for workflow composition.

1: **workflow** → "begin" **stmtlist** "end"
2: **stmtlist** → ϵ
3: **stmtlist** → **stmt** **stmtlist**
4: **stmt** → **sequence**
5: **stmt** → **branch**
6: **stmt** → **loop**
7: **stmt** → **fork**
8: **sequence** → **task**
9: **sequence** → **task** ";" **sequence**
10: **branch** → "if" **condition** "then" **stmtlist** "else" **stmtlist** "endif"
11: **loop** → "while" **condition** "do" **stmtlist** "endwhile"
12: **fork** → "fork" **forklist** "join"
13: **forklist** → **stmtlist**
14: **forklist** → **stmtlist** "||" **forklist**
15: **condition** → $0 \leq value \leq 1$
16: **task** → *identifier*

Table 2: Reference grammar for structured workflow specification.

task appears to the invoker as a non-terminating task. This, in practice, corresponds to detecting an invocation timeout, which may for example occur because of message loss by the network or because of request dropout by the server that runs the invoked task. The latter instead corresponds to a specific named failure (or exception) transferred to the invoking task by the invoked component.

Accordingly, hereafter we consider the following task-level properties related with failures: *response probability* and *propagation matrix*. Response probability (α) represents the probability that an invoked task returns the control. Formally, the invocation of a task can be described as a Bernoulli test for its termination. α is then the expected value of such a test:

$$Pr(\text{task}_i \text{ terminates}) = \alpha^i$$

By generalization, the response probability of a combination of tasks represents the probability that the entire combination returns the control, and its value strictly depends on how the tasks are combined as will be shown in Section 3.

In this paper we consider α to be input and time-independent. This is a strong assumption we plan to relax in future work, by allowing dependency from input patterns as in [13]. We will also try to enhance the response probability distribution by making it time-dependent, i.e. defining $\alpha(t)$ as a boolean function varying during time. Both the extensions require an improvement of the mathematical procedures of Section 3.

The motivation for considering the propagation matrix can be traced back to Avizienis et al. [3], who notices that by only considering halting failures we would get approximate results, unable to capture critical reliability aspects in a number of domains. A task can return control to its invoker and still yield an *erroneous* output, which is viewed by the caller as a failure. In [3] the authors propose to consider at least timing and content failures. In our work, we assume that for each component that may be used in a composition the software designer knows its finite set of relevant failure patterns, called *failure modes*. As we mentioned, not

all possible failures generated by a component necessarily propagate through other components and eventually become visible as failures of the whole workflow. In fact, a failure generated by a component may be confined inside the workflow boundaries. In this work, we are interested in modeling how each component may react as it receives a certain input pattern by providing a probabilistic transfer function. Specifically, in this paper we consider linear transfer functions defined for each task by its *propagation matrix*.

The propagation matrix $\mathbf{P}^t$ of a task t is defined as a $n \times n$ matrix such that p^t_{ij} represents the probability that t, invoked with incoming error mode i, returns an output error mode j, provided that t returns. $\mathbf{P}^t$ is a right stochastic matrix, i.e. a square matrix each of whose rows contains only nonnegative real numbers summing to 1. For generality, we consider mode 0 to denote the correct behavior of the component (*correct mode*).

DEFINITION 2.1. *A task t is characterized by the pair* $< \alpha^t, \boldsymbol{P}^t >$, *where* α^t *and* $\boldsymbol{P}^t$ *are the response probability and the propagation matrix of t, respectively.*

As introduced in [9], the reliability of a component strictly depends on its usage. In our framework, the usage profile of a component, and thus that of the invocation of its tasks, has to be characterized with respect to the set of failure modes. Specifically we define the usage profile $\boldsymbol{\beta}$ as follows.

DEFINITION 2.2. *The usage profile* $\boldsymbol{\beta}^t$ *of a task t is a stochastic vector with n elements, such that* ${\beta^t}_i$ *represents the probability that the invocation of t carries error mode i.*

In other words, $\boldsymbol{\beta}^t$ can be seen as the probability mass function of a discrete random variable with n possible outcomes corresponding to the indices of its elements (numbered from 0). Such probability mass function has to be inferred from historical usage data or guessed on the basis of the expected users behavior.

The outcome of the invocation of a task t with usage profile $\boldsymbol{\beta}^t$ (assuming that t returns) can be obtained applying t's propagation function to $\boldsymbol{\beta}^t$: $\boldsymbol{\beta}^t \cdot \mathbf{P}^t$.

Following [13], we can now define a number of reliability-related properties:

- **Reliability**: the probability that a task t produces a correct output given that it was invoked without any erroneous pattern:
$$R^t = p^t_{0,0} \cdot \alpha^t$$
- **Failure probability** of error mode i, $0 < i \leq n$: probability that a task t invoked in correct mode returns with error mode i:
$$F^t_i = p^t_{0,i} \cdot \alpha^t$$
- **Robustness** with respect to error mode i, $0 < i \leq n$: probability that a task t, invoked with error mode i, masquerades the error and returns a correct output:
$$B^t_i = p^t_{i,0} \cdot \alpha^t$$
- **Susceptibility** with respect to error mode i, $0 < i \leq n$: probability that a task t, invoked with error mode i, produces an erroneous output with any error mode (i.e., susceptibility is the complement of robustness):
$$S^t_i = \alpha^t \cdot \sum_{j=1}^{n} p^t_{i,j} = 1 - p^t_{i,0} = 1 - B^t_i$$

By adding information concerning the usage profile, it is possible to define more properties. For example it is possible to define:

- **Proclivity** with respect to error mode i, $0 < i \leq n$: probability that a task t produces in output an error mode i given the usage profile β^t:
$$L^t_i = \alpha^t \cdot \sum_r \beta^t_r \cdot p^t_{r,i}$$

The above properties are just examples to illustrate, without any sake of completeness, the use of α and $\mathbf{P}$ to specify finer reliability-related properties.

3. PROPERTY COMPOSITION FOR SEQUENTIAL PATTERNS

In this section we introduce the mathematical procedures to compute both response probabilities and propagation matrices of each sequential workflow pattern, starting from their corresponding values for the composed tasks. The parallel composition pattern is described separately in Section 4. The formulae we present here and in Section 4 will be invoked by a recursive computation, as described in in Section 5, in order to support the evaluation of nested workflow patterns, as described by our grammar. The result will be a calculus for evaluating the overall response probability and propagation matrix.

3.1 Sequence

Given two tasks A and B connected in a sequence S as shown in Table 1, we wish to compute the overall values α^S and $\mathbf{P}^S$ given the values α^A, $\mathbf{P}^A$, and α^B, $\mathbf{P}^B$ for tasks A and B, respectively. Concerning the overall response probability, we observe that the sequence S returns control to the invoker if both A and B return the control. Thus:

$$\alpha^S = \alpha^A \cdot \alpha^B \tag{1}$$

As for the propagation matrix, we observe that the output of task A for a generic usage profile $\boldsymbol{\beta}$ can be computed as $\boldsymbol{\beta} \cdot \mathbf{P}^A$ and such an output is sent directly as input to task B. Hence the final outcome is $(\boldsymbol{\beta} \cdot \mathbf{P}^A) \cdot \mathbf{P}^B$. By exploiting the associative property of matrix product, it is easy to derive the propagation matrix for the sequence of two tasks as:

$$\mathbf{P}^S = \mathbf{P}^A \cdot \mathbf{P}^B \tag{2}$$

3.2 Branch

A branch (see Table 1) represents a point of probabilistic choice between two branches. Given two task choices A and B connected in a branching pattern Br, we wish to compute the overall values α^{Br} and $\mathbf{P}^{Br}$ for the composed pattern given the values α^A, $\mathbf{P}^A$, and α^B, $\mathbf{P}^B$ for tasks A and B, respectively.

The propagation matrix of a branch can be formalized though the total probability theorem:

$$\mathbf{P}^{Br} = c \cdot \mathbf{P}^A + (1 - c) \cdot \mathbf{P}^B \tag{3}$$

The same reasoning can be applied to computing the response probability:

$$\alpha^{\mathbf{Br}} = c \cdot \alpha^A + (1 - c) \cdot \alpha^B \tag{4}$$

3.3 Loop

The loop pattern in Table 1 has a condition that represents the probability of executing its body task A. We wish to compute the overall values α^L and $\mathbf{P}^L$ for the composed pattern L given the values α^A and $\mathbf{P}^A$ for task A.

The simplest way of reasoning about this construct is to look at it as a branch with condition c that can return the control to the beginning of the loop after each iteration. Thus, being $\mathbf{P}^L$ our unknown, by applying equation 3,

$$\mathbf{P}^L = (1-c)\cdot\mathbf{I} + c\cdot\mathbf{P}^A\cdot\mathbf{P}^L \tag{5}$$

where $\mathbf{I}$ represents the identity matrix.

With basic algebraic manipulations, equation 5 can be solved obtaining:

$$\mathbf{P}^L = (1-c)\cdot(\mathbf{I} - c\cdot\mathbf{P}^A)^{-1} \tag{6}$$

A similar reasoning leads to the computation of the response probability for a loop construct, as:

$$\alpha^L = \frac{1-c}{1-c\cdot\alpha^A} \tag{7}$$

4. PROPERTY COMPOSITION FOR PARALLEL PATTERNS

The parallel composition pattern is quite complex and deserves careful analysis. The main difficulty is that the same syntactic structural pattern can have different semantics, as anticipated in Section 2. To distinguish among them, the join operation that merges the different threads should in fact be enriched with specific, application-dependent semantic annotations that specify the intended meaning of the join. The intended meaning affects the way we can compute the response probability and the propagation matrix, given the values of the same attributes for the composing threads. In the sequel, we draw our attention to a number of typical cases.

Let us consider the parallel pattern as described in Table 1. Let α^i and $\mathbf{P}^i$ be the response probability and the propagation matrix for all threads T_i. From these, we discuss how to compute the values α^{FJ} and $\mathbf{P}^{FJ}$ associated with the fork-join pattern FJ.

Case 1: 1 out of m.

This case refers to a Fork-Join pattern where termination of *only one* task is sufficient for the parallel composition to terminate. In practice, this case may occur when different instances of the same task are launched in parallel on a number of different processing units, and we wait for termination of the first one. As an example, consider the download of a file from several mirror sites. Completion of one download fulfills our goal.

In Case 1, the response probability α for the parallel construct can be computed as the probability that at least one of the m threads returns:

$$\alpha^{FJ}_{1m} = 1 - \prod_{i=1}^{m}(1-\alpha^i) \tag{8}$$

In order to specify the propagation matrix, let us define the function $h : \{1..m\} \to [0,1]$ as the probability for a thread to complete before all the others. This corresponds to the probability of being the one waited for at the *join*, then the expected propagation matrix can be computed as:

$$\mathbf{P}^{FJ}_{1m} = \sum_{i=1}^{m} h(i)\cdot\mathbf{P}^i \tag{9}$$

In practice, the probability distribution of the completion of the different threads can be derived by an easier to define metric for the threads, i.e., their average execution time. If τ_i is the average execution time of thread T_i then (according to [10]) $h(i)$ can be computed as:

$$h(i) = \frac{\frac{1}{\tau_i}}{\sum_i \frac{1}{\tau_i}} \tag{10}$$

assuming that the execution time of each thread follows an exponential distribution.

Case 2: m out of m (any order).

This case refers to the situation in which the Fork-Join pattern terminates only when *all* the parallel threads return control. The arrival order of their responses instead is irrelevant.

Think for example of a map-reduce approach applied to image processing, where the initial image is split into m equal areas and sent to m parallel threads that apply the same filter to every piece and with the additional requirement that all the parts of the image must be processed. The join node simply collects all the results and puts them together to deliver the manipulated image to the final user. In this case the order of completion of the m threads does not matter, but just the fact that all of them complete the assigned job. Concerning the response probability, it is straightforward to consider the return of control of the parallel construct as the probability of the event that all of the m threads return the control to the invoker:

$$\alpha^{FJ}_{mm} = \prod_{i=1}^{m}\alpha^i \tag{11}$$

Concerning the propagation matrix, given the equal division of the image among the m tasks, it is reasonable to consider that for each out-going error mode of each single task, its impact on the global output is proportional to the affected fraction of the input processed by the task. Applying the same consideration to all of the m parallel threads, the expected propagation matrix can be approximated as:

$$\mathbf{P^{FJ}}_{mm} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{P}^i \tag{12}$$

Case 3: k out of m (any order).

Consider a variation of the previous case, in which the parallel composition terminates when k out of the m threads return the control. We still assume that the order in which they terminate does not matter.

For instance, consider a variation of the image processing example, in which the image processing can terminate as soon as the manipulation of any k out of the m subareas of the image are completed.

The *join* node can transfer control as soon as any group of k threads accomplish the task. Thus we can split the problem in two steps: 1) identify the group of k threads which accomplish the tasks before the others and 2) compute the properties of this lead-

ing group. Let C_k be the set of all the combinations of class k over the m threads, and let $h : C_k \rightarrow [0, 1]$ represent the probability for each group C_{ki} to be the first group to complete the assignments:

$$\alpha_{km}^{FJ} = 1 - \prod_{C_{ki} \in C_k} (1 - \prod_{j \in C_{ki}} \alpha^j) \tag{13}$$

With the same approach, $\mathbf{P}_{km}^{FJ}$ can be computed as:

$$\mathbf{P}_{km}^{FJ} = \sum_{C_{ki} \in C_k} h(C_{ki}) \cdot \frac{1}{k} \sum_{j \in C_{ki}} \mathbf{P}^j \tag{14}$$

5. PARSING THE WORKFLOW

Given the definition of the workflow language in Section 2.1, it is possible to define the algorithm that evaluates the overall response probability and the propagation matrix by augmenting the grammar with attributes. The resulting attribute grammar ([26, 28]) allows the response probability and the propagation matrix of a workflow to be computed as the workflow is parsed by a bottom-up parser[1]. Let α and $\mathbf{P}$ be the attributes representing the response probability and the propagation matrix. Each of the tasks of the workflow, which correspond to the leaves of the parse tree, has initial values associated with its own α and $\mathbf{P}$ attributes. Hereafter we provide the attribute rule that synthesizes the attributes for every node of the parse tree given the values of the attributes associated with its children nodes. The rules will be given for each grammar production.

The attribute rules associated with production rules 1, 4, 5, 6, 7, 8 simply copy the values of the attributes in the child node corresponding to the right-hand side nonterminal into the attributes associated with the parent node, corresponding to the left-hand side nonterminal. The evaluation of the attributes for the other rules is justified by the discussion in Section 3.

Rule 2 defines an empty statement list. By convention, we assume that the synthesized value of the α attribute is 1 (i.e., it always transfers control) and the value of the $\mathbf{P}$ attribute is the identity matrix (i.e., it perfectly propagates every incoming error mode).

Rule 3 defines a non-empty statement list. The values of the synthesized attributes are evaluated as follows:

$$\begin{aligned} stmtlist.P &= stmt.P \cdot stmtlist.P \\ stmtlist.\alpha &= stmt.\alpha \cdot stmtlist.\alpha \end{aligned} \tag{15}$$

Rule 9 defines a sequence as a list of tasks. The values of the synthesized attributes are evaluated as follows:

$$\begin{aligned} sequence.P &= task.P \cdot sequence.P \\ sequence.\alpha &= task.\alpha \cdot sequence.\alpha \end{aligned} \tag{16}$$

A *branch* is composed by two statement lists, selected by a condition c, whose value is a real number v such that $0 \leq v \leq 1$. Hence, the values of the attributes for rule 10 can be computed as:

$$\begin{aligned} branch.P &= c.value \cdot stmtlist_1.P + (1 - c.value) \cdot stmtlist_2.P \\ branch.\alpha &= c.value \cdot stmtlist_1.\alpha + (1 - c.value) \cdot stmtlist_2.\alpha \end{aligned} \tag{17}$$

The attributes for the *loop* construct (rule 11) can be computed as follows (where I represents the identity matrix):

$$\begin{aligned} loop.P &= (1 - c.value) \cdot (I - c.value \cdot stmtlist.P)^{-1} \\ loop.\alpha &= (1 - c.value)/(1 - c.value \cdot stmtlist.\alpha) \end{aligned} \tag{18}$$

Fork-Join constructs, depending on how the designer intends to solve nondeterminism, may assume different forms. For the sake of simplicity here we show Case 2 *m out of m* which does not require the use of complex data structures for the computation of attributes. The other cases may be defined accordingly.

A *fork* construct is characterized as a list of threads (rule 12). Besides $\mathbf{P}$ and α, *forklist* has an additional integer attribute m, which represents the number of parallel threads. *Forklist* is defined by rules 13 and 14. The evaluation of attributes for rule 13 is defined as:

$$\begin{aligned} forklist.P &= stmtlist.P \\ forklist.\alpha &= stmtlist.\alpha \\ forklist.m &= 1 \end{aligned} \tag{19}$$

The evaluation of attributes for rule 14 is defined as:

$$\begin{aligned} forklist.P &= stmtlist.P + forklist.P \\ forklist.\alpha &= stmtlist.\alpha \cdot forklist.\alpha \\ forklist.m &= 1 + forklist.m \end{aligned} \tag{20}$$

Then the *fork* construct (rule 12) synthesizes its attributes as:

$$\begin{aligned} fork.P &= (1/forklist.m) \cdot forklist.P \\ fork.\alpha &= forklist.\alpha \end{aligned} \tag{21}$$

An ANTLR[2] specification of this grammar, as well as a Java implementation of the compiler can be downloaded from `http://home.dei.polimi.it/filieri/cbse2011`.

6. AN EXAMPLE

In order to illustrate the proposed technique and to demonstrate its effectiveness through a case study, in this section we apply it to the evaluation of an example taken from [14], shown in Fig. 1. The example deals with a composed Web service supporting travel management, which offers several functionalities starting from the travel location input, such as booking services and notifications to users. The travel service workflow specification is shown in Fig. 1 through an activity diagram.

The workflow initially invokes a sequence of three services to identifying the travel requirements. Then, two tasks are performed in parallel: parking booking and the calling invited attendees, until they confirm the invitation by answering to the call. After their execution, the service starts arranging a meeting and subsequently notifies the commitment for the travel to the user. In case the user does not confirm the reception of the travel plan, the travel service keeps sending her messages until the reception is confirmed.

According to the grammar expressed is Section 2, the code corresponding to the above example is reported in Table 3.

Before introducing the numerical values, we can focus on the symbolic algebraic composition of tasks' attributes. The *fork-join* block here implements the m-out-of-m parallel pattern, because completion of both the parallel branches is needed to accomplish the request, and their output will be combined independently of their arrival order.

By means of our compiler we obtain the following expression for the response probability of the whole workflow W:

9[1]This is because the attriibute grammar only uses synthesized attributes.

9[2]ANTLR is a widely used language tool (http://www.antlr.org/)

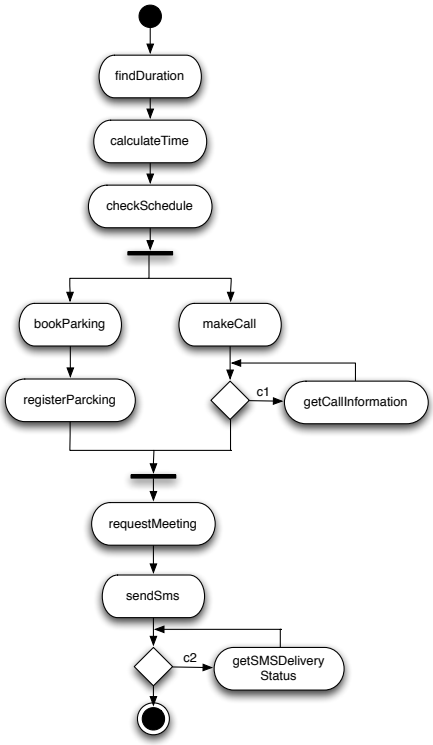


Figure 1: Example's workflow.

$$\begin{aligned}\alpha^W = \quad & \alpha^{findDuration} \cdot \alpha^{calculateTime} \\ & \cdot \alpha^{checkSchedule} \cdot \alpha^{bookParking} \cdot \alpha^{registerParking} \\ & \cdot \alpha^{makeCall} \cdot \frac{1-c1}{1-c1 \cdot \alpha^{getCallInformation}} \\ & \cdot \alpha^{requestMeeting} \cdot \alpha^{sendSms} \\ & \cdot \frac{1-c2}{1-c2 \cdot \alpha^{getSMSDeliveryStatus}}\end{aligned} \tag{22}$$

By setting $c1 = 0.333$ and $c2 = 0.6$ and the same reliability properties as in [14] we can obtain $\alpha^W = 0.775$, which is the same response probability of [14].

With respect to [14], we can further analyze the response of the system by looking at possible erroneous patterns contained in the system's response.

```
begin
  findDuration; calculateTime; checkSchedule;
  fork
    bookParking; registerParking;
  ||
    makeCall;
    while c1 do
      getCallInformation;
    endwhile
  ||
  join
  requestMeeting; sendSms;
  while c2 do
    getSMSDeliveryStatus;
  endwhile
end
```

Table 3: Source code of the example with the grammar of Section 2.

For this example we only define two failure modes, inspired by the treatment of [3], namely *content* and *timing*. *Content* failure represents the generic presence of an error in the answer that can derive, for example, from a logical error or from data corruption over the communication channels. *Timing* failure refers to an excessive delay in providing the response. We assume that *makeCall* and *findDuration* are managed through priority queues that make requests with high cumulated delay to pass over the line. Hence, these tasks are able to partially masquerade an incoming timing error.

The following formula 23 shows the propagation matrix for the example workflow:

$$\begin{aligned}\mathbf{P}^W = \quad & \mathbf{P}^{findDuration} \cdot \mathbf{P}^{calculateTime} \cdot \mathbf{P}^{checkSchedule} \\ & \cdot \tfrac{1}{2} \cdot [\mathbf{P}^{bookParking} \cdot \mathbf{P}^{registerParking} \\ & + \mathbf{P}^{makeCall}(1-c1) \cdot (I - c1 \cdot \mathbf{P}^{getCallInformation})^{-1}] \\ & \cdot \mathbf{P}^{requestMeeting} \cdot \mathbf{P}^{sendSms} \\ & \cdot ((1-c2) \cdot (I - c2 \cdot \mathbf{P}^{getSMSDeliveryStatus})^{-1})\end{aligned} \tag{23}$$

Let us consider that all the tasks have the same probability of introducing a content failure $p_{0,1} = 0.05$, and the same probability of introducing a timing failure equal to 0.04 ($p_{0,2}$). Both of these properties can be classified as *failure probabilities*, as for Section 2.2. Let us also assume that all the tasks propagate an incoming content failure with probability 1 ($p_{1,1}$), as well as they propagate a timing failure ($p_{2,2}$), with the exception of tasks *makeCall* and *findDuration*, which can masquerade an incoming failure (*robustness*, as in Section 2.2) mode with probability 0.75 ($p_{2,0}$).

By computing the propagation matrix for the example workflow $\mathbf{P}^W$, we obtain the following result:

$$\mathbf{P}^W = \begin{bmatrix} 0.4957 & 0.3030 & 0.2013 \\ 0 & 1.0000 & 0 \\ 0.4654 & 0.2221 & 0.3125 \end{bmatrix}$$

Looking at the resulting propagation matrix, we can notice that even if each single task has a relative low probability of introducing a content failure (0.05), composing them as in Figure 1 makes the entire system have a probability of producing a content failure equal to 0.303, which is considerable.

Considering the content failures, we can notice that, due to the fact that no components are able to compensate an incoming content failure, the entire system inherits such a property, revealed by the fact that $p^W_{1,1} = 1$.

Additionally, thanks to the ability of *makeCall* and *findDuration* to masquerade an incoming timing failure, the possible composition of the example system in a larger one would make the large system benefit from the masquerading capability of P^W, which is indeed $p^W = 0.4654$.

The closed-form expression of our reliability properties allows a deeper knowledge concerning the system under analysis, and thus further parametric analyses as well as sensitivity analyses could also be performed.

7. RELATED WORK

To the best of our knowledge, the issue of stochastic analysis of reliability for CB systems, taking into account multiple failure modes and their propagation inside the system has been considered only in [13]. Nevertheless there are a number of works strongly related to this. In the following we present a selection of related works to show on what our solution stands, and which is the starting point of this research.

With regards to architecture-based software reliability analysis several works have been proposed, some of which have been reviewed and classified in specific surveys [18, 24]. However, albeit error propagation is an important element in the chain that leads to a system failure, all existing approaches ignore it. In these approaches, the only considered parameters are the internal failure probability of each component and the interaction probabilities, with the underlying assumption that any error that arises in a component immediately manifest itself as an application failure, or equivalently that it always propagates (i.e. with probability one) up to the application outputs.

In the following, we shortly describe some of the works that mostly influenced the proposed solution. One of the first approaches to reliability that takes distance from debugging has been proposed in 1980 [9]. The approach got named from *user-oriented reliability*, which is defined as the probability that the program will give the correct output with a typical set of input data from the execution environment. The user-oriented approach is now the more widely adopted and it justifies the adoption of probabilistic methods as long as the system reliability depends on the probability that a fault gets activated during a run. The reliability of a system is computed as a function of both the reliability of its components and their frequency distribution of utilization, where the system is described by as a set of interacting modules which evolves as a stochastic Markov Process and the usage frequencies can be obtained from the structural description. In [36] the authors explore the possibility of transforming architecture expressed in three popular architectural styles into discrete Markov chains to be then analyzed by means of the approach proposed in [9]. Parametrized specification contracts, usage profile and reliability of required components as constituent factors for reliability analysis have been presented in [29]. Specifically, they consider components reliability as a combination of internal constant factors, such as reliability of the method body code, and variable factors, such as the reliability of external method calls. An approach for automatic reliability estimation in the context of self-assembling service-oriented computing taking into account relevant issues like compositionality and dependency on external resources has been proposed in [19].

The first and most significant attempt in obtaining the closed-form analytic solution of software architectures performance and reliability quantities, applying an iterative reduction algorithm taken and borrowed from the graph theory, is proposed in [4]. There are also several adaptation of such algorithm in different contexts such as, for example in discrete time Markov chain analysis [21]. In workflow context, the first attempt in such direction has been performed by Cardoso et al. [7, 8], which applied the reduction/aggregation techniques to some of the workflow patterns identified in [32]. But in the developed framework, failure modes propagation aspects have not been taken into account.

The concept of error propagation probability as the probability that an error, arising somewhere in the system, propagates through components, possibly up to the user interfaces has been introduced in [11]. The methodology of [11] assumes a single failure mode and provides tools to analyze how sensible the system is with respect to both failure and error propagation probability of each of its components. In [23], the authors proposed a notion of error permeability for modules as a basic characterization of modules' attitude to propagate errors. Also in this case, a single, non-halting failure mode is considered. Moreover, it is proposed a method for the identification of which modules are more likely exposed to propagated errors and which modules more likely produce severe consequences on the global system, considering the propagation path of their own failure. In [35, 34, 33] approaches based on fault injection to estimate the error propagation characteristics of a software system during testing are presented. In the context of safety some works exist dealing with multiple failure modes, see for example [20]. However they don't present any kind of stochastic analysis but only an examination of their possible propagation patterns.

With regard to the estimation of the propagation path probabilities, the basic information exploited by all the architecture-based methodologies is the probability that component i directly interacts with component j. At early design stages, where only models of the system are available, this information can be derived from software artifacts (e.g. UML interaction diagrams), possibly annotated with probabilistic data about the possible execution and interaction patterns [12]. A review and discussion of methodologies for the interaction probability estimate can be found in [18]. A more recent method has been discussed in [30], where a Hidden Markov model is used to cope with the imperfect knowledge about the component behavior. Once the interaction probabilities are known, the probability of the different error propagation paths can be estimated under the assumption that errors propagate through component interactions.

An important advantage of architectural analysis of reliability is the possibility of studying the sensitivity of the system reliability to the reliability of each component, as said in the Introduction. Although this advantage is widely recognized (e.g., [16]), few model-based approaches for computing the sensitivity of the system reliability with respect to each component reliability have been developed [9, 15]. A basic work for the sensitivity analysis of the reliability with respect to some system parameter was presented in [5], but it does not address specifically architectural issues. Moreover, all these models do not take into account the error propagation attribute and different failure modes.

Multiple failure modes and their propagation probabilities have been considered in [13]. This approach is based on DTMC models and it can be applied at early stages of software design providing a fine prediction model which can drive decisions about both architectural and behavioral aspects. The underlying model is also suitable for sensitivity analysis that establishes how much the global system reliability (for each failure mode) depends upon each model parameter.

The main contribution of our approach with regards to the literature is to provide an analytic technique for evaluating the propagation of different failure modes in component-based system and/or composed services starting from their workflow description. One of the main benefit of such technique is that it allows to obtain the closed-form solution of the failure modes propagation, expressed through a set of reliability quantities and measures specified in the paper (Section 3). The technique adopted is based on the aggregation-reduction approach well known in graph theory and also already applied in software/workflow contexts as mentioned above [4, 7, 8] but never applied, to the best of our knowledge, in failure propagation problems. With respect to [13], in this paper we provide a formalization based on a reference grammar defining the main workflow constructs, and a solution technique based on closed formulae that can be solved by means of algebraic operations. Besides, the entire analysis process is intrinsically compositional; it recursively builds the reliability formulae for the whole structured workflow by composing the formulae for its constituent patterns.

8. CONCLUSIONS

In this paper we presented a method for reliability analysis of systems designed as workflow-based compositions of components, where components are characterized by their failure profiles, which take into account possible multiple failure modes. The proposed

approach is described as a syntax-driven procedure that synthesizes a workflow's failure profile from the given failure profiles of the components. The method is viewed as a design-time aid that can help software engineers reason about system's reliability in the early stage of development.

The technique proposed in this paper can be improved along several directions. As we observed, parallel compositions can have a variety of application-dependent semantics. We intend to explore current industrial practices in order to identify practically relevant composition patterns and provide a formal specification of their reliability attributes. It may be interesting to define an ad-hoc formal specification language that can reduce the burden for designers. We also plan to explore the impact of embedding time dependency in the response probability function. This could help both to deal with timeouts and to automatically synthesize join synchronization points that depend on the parallel branches' response time. Finally, we are working on the implementation of our methodology on a real testbed, to assess its effectiveness through a more comprehensive set of real experiments.

9. ACKNOWLEDGEMENTS

Work partially supported by the Italian PRIN project D-ASAP and by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

10. REFERENCES

[1] A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, Sterling, D. König, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0. OASIS Committee Draft, May 2006.

[2] H. Ammar, D. Nassar, W. Abdelmoez, M. Shereshevsky, and A. Mili. A framework for experimental error propagation analysis of software architecture specifications. In *Proc. of International Symposium on Software Reliability Engineering*. IEEE, 2002.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1:11–33, January 2004.

[4] B. Beizer. *Micro-Analysis of Computer System Performance*. John Wiley & Sons, Inc., New York, NY, USA, 1978.

[5] J. T. Blake, A. L. Reibman, and K. S. Trivedi. Sensitivity analysis of reliability and performability measures for multiprocessor systems. In *SIGMETRICS*, pages 177–186, 1988.

[6] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9:366–371, May 1966.

[7] A. Cardoso. *Quality of Service and Semantic Composition of Workflows*. PhD thesis, Graduate School of the University of Georgia, Athens, Georgia, August 2002.

[8] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Web semantics: Science, services and agents on the world wide web; quality of service for workflows and web service processes. *Journal of Web Semantics, Elsevier*, 1(3):281–308, 2004.

[9] R. C. Cheung. A user-oriented software reliability model. *IEEE Trans. Softw. Eng.*, 6(2):118–125, 1980.

[10] E. Cinlar. Introduction to stochastic processes. *Englewood Cliffs*, 1975.

[11] V. Cortellessa and V. Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. *LNCS*, 4608:140, 2007.

[12] V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of uml based software models. In *Workshop on Software and Performance*, pages 302–309, 2002.

[13] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola. Reliability analysis of component-based systems with multiple failure modes. In L. Grunske, R. Reussner, and F. Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13238-4_1.

[14] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality prediction of service compositions through probabilistic model checking. In S. Becker, F. Plasil, and R. Reussner, editors, *Quality of Software Architectures. Models and Architectures*, volume 5281 of *Lecture Notes in Computer Science*, pages 119–134. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-87879-7_8.

[15] S. S. Gokhale and K. S. Trivedi. Reliability prediction and sensitivity analysis based on software architecture. In *ISSRE*, pages 64–78. IEEE Computer Society, 2002.

[16] S. S. Gokhale, W. E. Wong, J. R. Horgan, and K. S. Trivedi. An analytical approach to architecture-based software performance and reliability prediction. *Perform. Eval.*, 58(4), 2004.

[17] K. Goseva-Popstojanova, A. Mathur, and K. Trivedi. Comparison of architecture-based software reliability models. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 22 – 31, 2001.

[18] K. Goseva-Popstojanova and K. S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Perform. Eval.*, 45(2-3):179–204, 2001.

[19] V. Grassi. Architecture-based reliability prediction for service-oriented computing. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems III*, volume 3549 of *Lecture Notes in Computer Science*, pages 279–299. Springer Berlin / Heidelberg, 2005. 10.1007/11556169_13.

[20] L. Grunske and J. Han. A comparative study into architecture-based safety evaluation methodologies using aadl's error annex and failure propagation models. In *HASE*, pages 283–292. IEEE Computer Society, 2008.

[21] E. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric markov models. In C. Pasareanu, editor, *Model Checking Software*, volume 5578 of *Lecture Notes in Computer Science*, pages 88–106. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-02652-2_10.

[22] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[23] M. Hiller, A. Jhumka, and N. Suri. Epic: profiling the propagation and effect of data errors in software. *Computers, IEEE Transactions on*, 53(5):512 – 530, May 2004.

[24] A. Immonen and E. Niemel. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008.

[25] A. Immonen and E. NiemelŁ. Survey of reliability and availability prediction methods from the viewpoint of

software architecture. *Software and Systems Modeling*, 7:49–65, 2008.

[26] D. E. Knuth. Correction: Semantics of context-free languages. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[27] R. K. L. Ko. A computer scientist's introductory guide to business process management (bpm). *Crossroads*, 15:4:11–4:18, June 2009.

[28] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[29] R. Reussner, H. W. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3):241–252, 2003.

[30] R. Roshandel. Calculating architectural reliability via modeling and analysis. In *ICSE*, pages 69–71. IEEE Computer Society, 2004.

[31] W. M. C. Specification. *Workflow Management Coalition Terminology and Glossary (Document No. WFMC-TC-1011)*. Number 3.0. Workflow Management Coalition Specification, http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf, 1999.

[32] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14:5–51, 2003. 10.1023/A:1022883727209.

[33] J. M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. Software Eng.*, 18(8):717–727, 1992.

[34] J. M. Voas. Error propagation analysis for cots systems. *Computing and Control Engineering Journal*, 8(6):269–272, 1997.

[35] W. Abdelmoez et al. Error propagation in software architectures. In *METRICS '04*, pages 384–393, Washington, DC, USA, 2004. IEEE Computer Society.

[36] W.-L. Wang, Y. Wu, and M.-H. Chen. An architecture-based software reliability model. *Pacific Rim International Symposium on Dependable Computing, IEEE*, 0:143, 1999.

[37] S. White. *Process modeling notations and workflow patterns*, chapter Workflow Handbook, pages 265–294. Future strategies inc. lighthouse point, fl, usa., 2004. edition, 2004.