Incremental Syntactic-Semantic Reliability Analysis of Evolving Structured Workflows

Domenico Bianculli[1], Antonio Filieri[2], Carlo Ghezzi[3], and Dino Mandrioli[3]

[1] University of Luxembourg, domenico.bianculli@uni.lu
[2] University of Stuttgart, antonio.filieri@informatik.uni-stuttgart.de
[3] Politecnico di Milano, {ghezzi,mandrioli}@elet.polimi.it

Abstract. Modern enterprise information systems are built following the paradigm of service-orientation. This paradigm promotes workflow-based software composition, where complex business processes are realized by orchestrating different, heterogenous components. These workflow descriptions evolve continuously, to adapt to changes in the business goals or in the enterprise policies.
Software verification of evolving systems is challenging mainstream methodologies and tools. Formal verification techniques often conflict with the time constraints imposed by change management practices for evolving systems. Since changes in these systems are often local to restricted parts, an incremental verification approach could be beneficial.
In this paper we focus on the probabilistic verification of reliability requirements of structured workflows. We propose a novel incremental technique based on a syntactic-semantic approach. Reliability analysis is driven by the syntactic structure (defined by an operator-precedence grammar) of the workflow and encoded as semantic attributes associated with the grammar. Incrementality is achieved by coupling the evaluation of semantic attributes with an incremental parsing technique. The approach has been implemented in a prototype tool; preliminary experimental evaluation confirms the theoretical speedup over a non-incremental approach.

1 Introduction

Enterprise information systems are realized nowadays by leveraging the principles of service-oriented architecture [30]. This paradigm fosters the design of systems that rely on *workflow-based* composition mechanisms, like those offered by BPEL, where complex applications are realized by integrating different, heterogenous services, possibly from different divisions within the same organization or even from third-party organizations. These workflows often realize crucial business functions; their correctness and reliability is of ultimate importance for the enterprises.

Moreover, these systems represent an instance of *open-world software* [3] where, because of the intrinsic dynamicity and decentralization, service behaviors and interactions cannot be fully controlled or predicted. These characteristics, when bundled with the inherent need for enterprise software to evolve (e.g., to adapt to changes in the business goals or in the enterprise policies), require to rethink the various engineering phases, for dealing with the phenomenon of software evolution; in this paper we focus on the verification aspect.

Incremental verification has been suggested as a possible approach to dealing with evolving of software [35]. An incremental verification approach tries to reuse as much as possible the results of a previous verification step, and accommodates within the verification procedure—possibly in a "smart" way—the changes occurring in the new version. By avoiding re-executing the verification process from scratch, incremental verification may considerably reduce the verification time. This may be appealing for adoption within agile development processes. Incremental verification may speed up change management, which may be subject to severe time constraints. Moreover, incremental verification helps software engineers reason and understand the effects and the implications of changes.

In this paper we propose a novel incremental technique for performing probabilistic verification of reliability requirements of structured workflows. Our technique follows a syntactic-semantic approach: reliability verification is *driven* by the structure of the workflow (prescribed by a formal grammar) and *encoded* as synthesis of semantic attributes [31], associated with the grammar and evaluated by traversing the syntax tree of the workflow. The technique is realized on top of SiDECAR [4, 5] (Syntax-DrivEn inCrementAl veRification), our general framework to define verification procedures, which are automatically enhanced with incrementality by the framework itself. The framework is based on operator precedence grammars [20], which allow for re-parsing, and hence semantic re-analysis, to be confined within an inner portion of the input that encloses the changed part [2]. This property is the key for an efficient incremental verification procedure: since the verification procedure is encoded within attributes, their evaluation proceeds incrementally, hand-in-hand with parsing. We report on the preliminary evaluation of the tool implementing the proposed technique; the results shows a significant speedup over a non-incremental approach.

The rest of the paper is structured as follows. Section 2 introduces some background concepts on operator precedence grammars and attribute grammars. Section 3 shows how our framework exploits operator precedence grammars to support syntactic-semantic incremental verification. Section 4 details our incremental reliability verification technique. In Sect. 5 we present the preliminary experimental evaluation of the approach. Section 6 surveys related work. Section 7 provides some concluding remarks.

2 Background

Hereafter we briefly recall the definitions of operator precedence grammars and attribute grammars. For more information on formal languages and grammars, we refer the reader to [25] and [10].

2.1 Operator precedence Grammars

A *context-free (CF)* grammar G is a tuple $G = \langle V_N, V_T, P, S \rangle$, where V_N is a finite set of non-terminal symbols; V_T is a finite set of terminal symbols, disjoint from V_N; $P \subseteq V_N \times (V_N \cup V_T)^*$ is a relation whose elements represent the rules of the grammar; $S \in V_N$ is the axiom or start symbol. We use the following naming convention, unless otherwise specified: non-terminal symbols are enclosed within chevrons, such as $\langle A \rangle$;

$$
\begin{array}{lll}
\langle S\rangle & ::= \langle A\rangle & \{value(\langle S\rangle) = value(\langle A\rangle)\} \\
\langle S\rangle & ::= \langle B\rangle & \{value(\langle S\rangle) = value(\langle B\rangle)\} \\
\langle A_0\rangle & ::= \langle A_1\rangle \text{ `+' } \langle B\rangle & \{value(\langle A_0\rangle) = value(\langle A_1\rangle) + value(\langle B\rangle)\} \\
\langle A\rangle & ::= \langle B_1\rangle \text{ `+' } \langle B_2\rangle & \{value(\langle A\rangle) = value(\langle B_1\rangle) + value(\langle B_2\rangle)\} \\
\langle B_0\rangle & ::= \langle B_1\rangle \text{ `*' `n'} & \{value(\langle B_0\rangle) = value(\langle B_1\rangle) * eval(\text{`n'})\} \\
\langle B\rangle & ::= \text{`n'} & \{value(\langle B\rangle) = eval(\text{`n'})\}
\end{array}
$$

(a)

	`n`	`*`	`+`
`n`		$\gtrdot$	$\gtrdot$
`*`	$\doteq$		
`+`	$\lessdot$	$\lessdot$	$\gtrdot$

(b)

Fig. 1. (a) Example of an operator grammar (‘n’ stands for any natural number), extended with semantic attributes; (b) its operator precedence matrix

terminal ones are enclosed within single quotes, such as ‘+’ or are denoted by lowercase letters at the beginning of the alphabet $(a,b,c,\ldots)$; lowercase letters at the end of the alphabet $(u,v,x,\ldots)$ denote terminal strings; ε denotes the empty string. For the notions of *immediate derivation* ($\Rightarrow$), *derivation* ($\stackrel{*}{\Rightarrow}$), and the language $L(G)$ generated by a grammar G please refer to the standard literature, e.g., [25].

A rule is in *operator form* if its right hand side (rhs) has no adjacent non-terminals; an *operator grammar (OG)* contains only rules in operator form.

Operator precedence grammars (OPGs) [20] are defined starting from operator grammars by means of binary relations on V_T named *precedence*. Given two terminals, the precedence relations between them can be of three types: *equal-precedence* ($\doteq$), *takes-precedence* ($\gtrdot$), and *yields-precedence* ($\lessdot$). The meaning of precedence relations is analogous to the one between arithmetic operators and is the basic driver of deterministic parsing for these grammars. Precedence relations can be computed in an automatic way for any operator grammar. We represent the precedence relations in a $V_T \times V_T$ matrix, named *operator precedence matrix (OPM)*. An entry $m_{a,b}$ of an OPM represents the set of operator precedence relations holding between terminals a and b. For example, Fig. 1b shows the OPM for the grammar of arithmetic expressions depicted at the left side of Fig. 1a. Precedence relations have to be neither reflexive, nor symmetric, nor transitive, nor total. If an entry $m_{a,b}$ of an OPM M is empty, the occurrence of the terminal a followed by the terminal b represents a malformed input, which cannot be generated by the grammar.

Definition 1 (Operator Precedence Grammar). *An operator grammar G is an operator precedence grammar if and only if its OPM is a conflict-free matrix, i.e., for each* $a,b \in V_T, |m_{a,b}| \leq 1$.

Definition 2 (Fischer Normal Form, from [10]). *An OPG is in Fischer Normal Form (FNF) if it is invertible, the axiom* $\langle S\rangle$ *does not occur in the right-hand side of any rule, no empty rule exists except possibly* $\langle S\rangle \Rightarrow \varepsilon$*, the other rules having* $\langle S\rangle$ *as left-hand side (lhs) are renaming, and no other renaming rules exist.*

The grammar of Fig. 1a is in FNF. In the sequel, we assume, without loss of generality, that OPGs are in FNF. Also, as is customary in the parsing of OPGs, the input strings are implicitly enclosed between two ‘#’ special characters, such that ‘#’ yields

precedence to any other character and any character takes precedence over '#'. The key feature of OPG parsing is that a sequence of terminal characters enclosed within a pair $\lessdot \gtrdot$ and separated by $\doteq$ uniquely determines a rhs to be replaced, with a shift-reduce algorithm, by the corresponding lhs. Notice that in the parsing of these grammars non-terminals are "transparent", i.e., they are not considered for the computation of the precedence relations. For instance, consider the syntax tree of Fig. 2a generated by the grammar of Fig. 1a: the leaf '6' is preceded by '+' and followed by '∗'. Because '+' $\lessdot$ '6' $\gtrdot$ '∗', '6' is reduced to $\langle B \rangle$. Similarly, in a further step we have '+' $\lessdot$ $\langle B \rangle$ '∗' $\doteq$ '7' $\gtrdot$ '∗' and we apply the reduction $\langle B \rangle \Rightarrow \langle B \rangle$ '∗' '7' (notice that non-terminal $\langle B \rangle$ is "transparent") and so on.

2.2 Attribute Grammars

Attribute Grammars (AGs) have been proposed by Knuth as a way to express the semantics of programming languages [31]. AGs extend CF grammars by associating *attributes* and semantic functions to the rules of a CF grammar; attributes define the "meaning" of the corresponding nodes in the syntax tree. In this paper we consider only *synthesized* attributes, which characterize an information flow from the children nodes (of a syntax tree) to their parents; more general attribute schemas do not add semantic power [31].

An AG is obtained from a CF grammar G by adding a finite set of attributes SYN and a set SF of semantic functions. Each symbol $X \in V_N$ has a set of (synthesized) attributes $SYN(X)$; $SYN = \bigcup_{X \in V_N} SYN(X)$. We use the symbol α to denote a generic element of SYN; we assume that each α takes values in a corresponding domain T_α. The set SF consists of functions, each of them associated with a rule p in P. For each attribute α of the lhs of p, a function $f_{p\alpha} \in SF$ synthesizes the value of α based on the attributes of the non-terminals in the rhs of p. For example, the grammar in Fig. 1a can be extended to an attribute grammar that computes the value of an expression. All nodes have only one attribute called *value*, with $T_{value} = \mathbb{N}$. The set of semantic functions SF is defined as in the right side of Fig. 1a, where semantic functions are enclosed in braces next to each rule. The $+$ and $*$ operators appearing within braces correspond, respectively, to the standard operations of arithmetic addition and multiplication, and $eval(\cdot)$ evaluates its input as a number. Notice also that, within a rule, different occurrences of the same grammar symbol are denoted by distinct subscripts.

3 Syntactic-semantic Incrementality

Our incremental technique for probabilistic verification of reliability requirements of structured workflows is realized on top of SiDECAR [4], our general framework for incremental verification. The framework exploits a syntactic-semantic approach to define verification procedures that are encoded as semantic functions associated with an attribute grammar. In this section we show how OPGs, equipped with a suitable attribute schema, can support incrementality in such verification procedures in a natural and efficient way.

3.1 The Locality Property and Syntactic Incrementality

The main reason for the choice of OPGs is that, unlike more commonly used grammars that support deterministic parsing, they possess and benefit from the *locality property*, i.e., the possibility of starting the parsing from any arbitrary point of the sentence to be analyzed, independent of the context within which the sentence is located. In fact for OPGs the following proposition holds.

Proposition 1. If $a\langle A\rangle b \stackrel{*}{\Rightarrow} asb$, then, for every t,u, $\langle S\rangle \stackrel{*}{\Rightarrow} tasbu$ iff $\langle S\rangle \stackrel{*}{\Rightarrow} ta\langle A\rangle bu \stackrel{*}{\Rightarrow} tasbu$. As a consequence, if s is replaced by v in the context $[\![ta,bu]\!]$, and $a\langle A\rangle b \stackrel{*}{\Rightarrow} avb$, then $\langle S\rangle \stackrel{*}{\Rightarrow} ta\langle A\rangle bu \stackrel{*}{\Rightarrow} tavbu$, and (re)parsing of $tavbu$ can be stopped at $a\langle A\rangle b \stackrel{*}{\Rightarrow} avb$.

Hence, if we build—with a bottom-up parser—the derivation $a\langle A\rangle b \stackrel{*}{\Rightarrow} avb$, we say that a *matching condition* with the previous derivation $a\langle A\rangle b \stackrel{*}{\Rightarrow} asb$ is satisfied and we can replace the old subtree rooted in $\langle A\rangle$ with the new one, independently of the global context $[\![ta,bu]\!]$ (only the local context $[\![a,b]\!]$ matters for the incremental parsing).

For instance, consider the string and syntax tree of Fig. 2a. Assume that the expression is modified by replacing the term '6*7*8' with '7*8'. The corresponding new subtree can clearly be built independently within the context $[\![\text{'+'},\text{'\#'}]\!]$. The matching condition is satisfied by '+'$\langle B\rangle$'#' $\stackrel{*}{\Rightarrow}$ '+''6''*''7''*''8''#' and '+'$\langle B\rangle$'#' $\stackrel{*}{\Rightarrow}$ '+''7''*''8''#'; thus the new subtree can replace the original one without affecting the remaining part of the global tree. If, instead, we replace the second '+' by a '*', the affected portion of syntax tree would be larger and more re-parsing would be necessary[4].

In general, the incremental parsing algorithm, for any replacement of a string w by a string w' in the context $[\![t,u]\!]$, automatically builds the minimal "sub-context" $[\![t_1,u_1]\!]$ such that for some $\langle A\rangle$, $a\langle A\rangle b \stackrel{*}{\Rightarrow} at_1wu_1b$ and $a\langle A\rangle b \stackrel{*}{\Rightarrow} at_1w'u_1b$.

The locality property[5] has a price in terms of generative power. For example, the LR grammars traditionally used to describe and parse programming languages do not enjoy it. However they can generate all the deterministic languages. OPGs cannot; this limitation, however, is more of theoretical interest than of real practical impact. Large parts of the grammars of many computer languages are operator precedence [25, p. 271]; a complete OPG is available for Prolog [7]. Moreover, in many practical cases one can obtain an OPG by minor adjustments to a non operator-precedence grammar [20].

In the current SiDECAR prototype, we developed an incremental parser for OPGs that exhibits the following features: linear complexity in the length of the string, in case of parsing from scratch; linear complexity in the size of the *modified subtree(s)*, in case of incremental parsing; $O(1)$ complexity of the matching condition test.

3.2 Semantic Incrementality

In a bottom-up parser, semantic actions are performed during a reduction. This allows the re-computation of semantic attributes after a change to proceed hand-in-hand with

[4] Some further optimization could be applied by integrating the matching condition with techniques adopted in [23] (not reported here for brevity).

[5] The locality property has also been shown to support an efficient parallel parsing technique [2], which is not further exploited here.

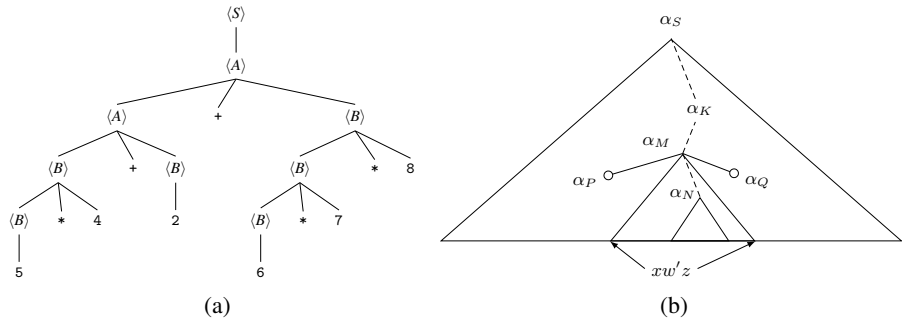


Fig. 2. (a) Abstract syntax tree of the expression '`5*4+2+6*7*8`'; (b) Incremental evaluation of semantic attributes on a generic syntax tree

the re-parsing of the modified substring. Suppose that, after replacing substring w with w', incremental re-parsing builds a derivation $\langle N \rangle \stackrel{*}{\Rightarrow} xw'z$, with the same non-terminal $\langle N \rangle$ as in $\langle N \rangle \stackrel{*}{\Rightarrow} xwz$, so that the matching condition is verified. Assume also that $\langle N \rangle$ has an attribute α_N. Two situations may occur related to the computation of α_N:

1. The α_N attribute associated with the new subtree rooted in $\langle N \rangle$ has the same value as before the change. In this case, all the remaining attributes in the rest of the tree will not be affected, and no further analysis is needed.

2. The new value of α_N is different from the one it had before the change. In this case (see Fig. 2b) only the attributes on the path from $\langle N \rangle$ to the root $\langle S \rangle$ (e.g., $\alpha_M, \alpha_K, \alpha_S$) may change and in such case they need to be recomputed. The values of the other attributes not on the path from $\langle N \rangle$ to the root (e.g., α_P and α_Q) do not change: there is no need to recompute them.

4 Incremental Reliability Analysis of Structured Workflows

In this section we define our procedure for incremental reliability analysis of structured workflows. As mentioned in the previous section, SiDECAR requires the verification procedure to be encoded as an attribute grammar schema. We assume that the structured workflows are written in a tiny and simple language called *Mini*, whose OPG is shown in Fig. 3. It is a minimalistic language that includes the major constructs of structured programming and allows for expressing the *sequence*, *exclusive choice*, *simple merge*, and *structured loops* patterns, from van der Aalst's workflow patterns collection [36].

The verification procedure is based on our previous work [13], which supports the analysis of workflow constructs similar to those in *Mini*, in a non-incremental way; we refer the reader to [13] for the technical choice behind the analysis itself. Moreover, for the sake of readability and to reduce the complexity of attribute schemas, *Mini* workflows support only (global) boolean variables; we model invocation of external services as boolean functions with no input parameters. We remark that more complex analyses and workflow languages (see, for example, the extension of [13] in [12] for

support of BPEL business processes, including parallelism and nested workflows) could be supported with richer attribute schemas.

Reliability is a "user-oriented" property [8]; i.e., a software may be more or less reliable depending on its use. If user inputs do not activate a fault, a failure may never occur even in a software containing defects [1]; on the other hand, users may stress a faulty component, leading to a high frequency of failure events. Here we consider reliability as the probability of successfully accomplishing an assigned task, when requested.

To show the benefits of incrementality, we will apply the verification procedure to analyze two versions of the same example workflow (shown in Fig. 4a). They differ in the assignment at line 3, which determines the execution of the subsequent *if* statement, with implications on the results of the two analyses. Figure 4b depicts the syntax tree of version 1 of the workflow, as well as the subtree that is different in version 2; nodes of the tree have been numbered for quick reference.

The following notation is introduced to specify the attribute schema of the verification procedure. For a *Mini* workflow, let F be the set of functions; V the set of variables defined within the workflow; E the set of boolean expressions that can appear as the condition of an *if* or a *while* statement in the workflow. An expression $e \in E$ is either a combination of boolean predicates on variables or a placeholder predicate labeled $*$.

To model the probabilistic verification procedure, first we assume that each function $f \in F$ has a probability $Pr_S(f)$ of successfully completing its execution. If successfully executed, the function returns a boolean value. We are interested in the returned value of a function in case it appears as the rhs of an assignment because the assigned variable may appear in a condition. The probability of assigning *true* to the lhs variable of the statement is the probability that the function returns *true*, which is the product $Pr_S(f) \cdot Pr_T(f)$, where $Pr_T(f)$ is the *conditioned* probability that f returns *true* given that it has been successfully executed. For the sake of readability, we make the simplifying assumption that all functions whose return value is used in an assignment are always successful, i.e., have $Pr_S(f) = 1$. Thanks to this assumption the probability of f returning *true* coincides with $Pr_T(f)$ and allows us to avoid cumbersome, though conceptually simple, formulae in the following development.

For the conditions $e \in E$ of *if* and *while* statements, $Pr_T(e)$ denotes the probability of e to be evaluated to *true*. In case of an *if* statement, the evaluation of a condition e leads to a probability $Pr_T(e)$ of following the *then* branch, and $1 - Pr_T(e)$ of following

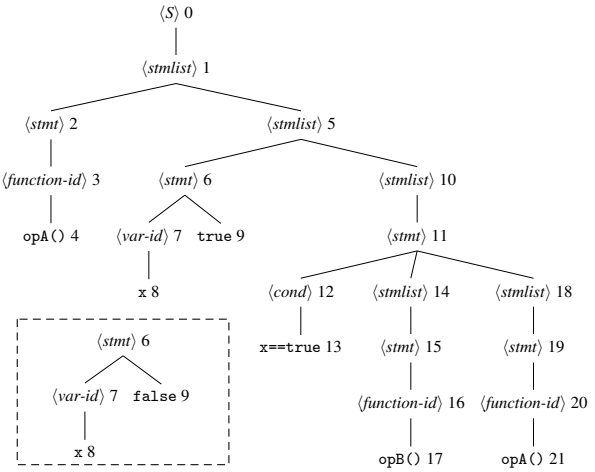
⟨*S*⟩ ::= '`begin`' ⟨*stmtlist*⟩ '`end`'
⟨*stmtlist*⟩ ::= ⟨*stmt*⟩ '`;`' ⟨*stmtlist*⟩ | ⟨*stmt*⟩ '`;`'
⟨*stmt*⟩ ::= ⟨*function-id*⟩ '`(`' '`)`' | ⟨*var-id*⟩ '`:=`' '`true`' | ⟨*var-id*⟩ '`:=`' '`false`'
| ⟨*var-id*⟩ '`:=`' ⟨*function-id*⟩ '`(`' '`)`'
| '`if`' ⟨*cond*⟩ '`then`' ⟨*stmtlist*⟩ '`else`' ⟨*stmtlist*⟩ '`endif`'
| '`while`' ⟨*cond*⟩ '`do`' ⟨*stmtlist*⟩ '`endwhile`'
⟨*var-id*⟩ ::= ...
⟨*function-id*⟩ ::= ...
⟨*cond*⟩ ::= ...

Fig. 3. The grammar of the *Mini* language

```
1 begin
2  opA();
3  x := true;
4  if (x==true)
5   then opB();
6   else opA();
7  endif;
8 end
```

```
1 begin
2  opA();
3  x := false;
4  if (x==true)
5   then opB();
6   else opA();
7  endif;
8 end
```

(a) Version 1 (top)
Version 2 (bottom)



(b) The syntax tree of version 1 of the example workflow; the subtree in the box shows the difference (node 9) in the syntax tree of version 2

Fig. 4. The two versions of the example workflow and their syntax tree(s)

the *else* branch. For *while* statements, $Pr_T(e)$ is the probability of executing one iteration of the loop. The probability of a condition to be evaluated to *true* or *false* depends on the current usage profile and can be estimated on the basis of the designer's experience, the knowledge of the application domain, or gathered from previous executions or running instances by combining monitoring and statistical inference techniques [18].

The value of $Pr_T(e)$ is computed as follows. If the predicate is the placeholder $*$, the probability is indicated as $Pr_T(*)$. If e is a combination of boolean predicates on variables, the probability value is defined with respect to its atomic components (assuming probabilistic independence among the values of the variables in V):

- $e =$ `"v==true"` $\implies Pr_T(e) = Pr_T(v)$
- $e =$ `"v==false"` $\implies Pr_T(e) = 1 - Pr_T(v)$
- $e = e_1 \wedge e_2 \implies Pr_T(e) = Pr_T(e_1) \cdot Pr_T(e_2)$
- $e = \neg e_1 \implies Pr_T(e) = 1 - Pr_T(e_1)$

The initial value of $Pr_T(v)$ for a variable $v \in V$ is undefined; after the variable is assigned, it is defined as follows:

- `v:=true` $\implies Pr_T(v) = 1$
- `v:=false` $\implies Pr_T(v) = 0$
- `v:=f()` $\implies Pr_T(v) = Pr_T(f)$

The reliability of a workflow is computed as the *expected probability value* of its successful completion. To simplify the mathematical description, we assume independence among all the failure events.

The reliability of a sequence of statements is essentially the probability that all of them are executed successfully. Given the independence of the failure events, it is the product of the reliability value of each statement.

For an *if* statement with condition e, its reliability is the reliability of the *then* branch weighted by the probability of e to be *true*, plus the reliability of the *else* branch weighted by the probability of e to be *false*. This intuitive definition is formally grounded on the law of total probability and the previous assumption of independence.

The reliability of a *while* statement with condition e and body b is determined by the number of iterations k. We also assume that $Pr_T(e) < 1$, i.e., there is a non-zero probability of exiting the loop, and that $Pr_T(e)$ does not change during the iterations. The following formula is derived by applying well-known properties of probability theory:

$$E(Pr_S(\langle while \rangle)) = \sum_{k=0}^{\infty} (Pr_T(e) \cdot Pr_S(b))^k \cdot (1 - Pr_T(e)) = \frac{1 - Pr_T(e)}{1 - Pr_T(e) \cdot Pr_S(b)}$$

A different construction of this result can be found in [13].

We are now ready to encode this analysis through the following attributes:
- $SYN(\langle S \rangle) = SYN(\langle stmlist \rangle) = SYN(\langle stmt \rangle) = \{\gamma, \vartheta\}$;
- $SYN(\langle cond \rangle) = \{\delta\}$;
- $SYN(\langle function\text{-}id \rangle) = SYN(\langle var\text{-}id \rangle) = \{\eta\}$;

where:
- γ represents the reliability of the execution of the subtree rooted in the node the attribute corresponds to.
- ϑ represents the knowledge acquired after the execution of an assignment. Precisely, ϑ is a set of pairs $\langle v, Pr_T(v) \rangle$ with $v \in V$ such that there are no two different pairs $\langle v_1, Pr_T(v_1) \rangle, \langle v_2, Pr_T(v_2) \rangle \in \vartheta$ with $v_1 = v_2$. If $\nexists \langle v_1, Pr_T(v_1) \rangle \in \vartheta$ no knowledge has been gathered concerning the value of a variable v_1. If not differently specified, ϑ is empty.
- δ represents $Pr_T(e)$, with e being the expression associated with the corresponding node.
- η is a string corresponding to the literal value of an identifier.

The actual value of γ in a node has to be evaluated with respect to the information possibly available in ϑ. For example, let us assume that for a certain node n_1, $\gamma(n_1) = .9 \cdot Pr_T(v)$. This means that the actual value of $\gamma(n_1)$ depends on the value of the variable v. The latter can be decided only after the execution of an assignment statement. If such assignment happens at node n_2, the attribute $\vartheta(n_2)$ will contain the pair $\langle v, Pr_T(v) \rangle$. For example, let us assume $Pr_T(v) = .7$; after the assignment, the actual value of $\gamma(n_1)$ is refined considering the information in $\vartheta(n_2)$, assuming the numeric value .63. We use the notation $\gamma(\cdot) \mid \vartheta(\cdot)$ to describe the operation of refining the value of γ with the information in ϑ. Given that $\gamma(\cdot) \mid \emptyset = \gamma(\cdot)$, the operation will be omitted when $\vartheta(\cdot) = \emptyset$.

The attribute schema for the Mini language is defined as follows:

1. $\langle S \rangle$ ::= '`begin`' $\langle stmtlist \rangle$ '`end`'
   $\gamma(\langle S \rangle) := \gamma(\langle stmtlist \rangle)$
2. (a) $\langle stmtlist_0 \rangle$ ::= $\langle stmt \rangle$ '`;`' $\langle stmtlist_1 \rangle$
   $\gamma(\langle stmtlist_0 \rangle) := (\gamma(\langle stmt \rangle) \cdot \gamma(\langle stmtlist_1 \rangle)) \mid \vartheta(\langle stmt \rangle)$
   (b) $\langle stmtlist \rangle$ ::= $\langle stmt \rangle$ '`;`'
   $\gamma(\langle stmtlist \rangle) := \gamma(\langle stmt \rangle)$
3. (a) $\langle stmt \rangle$ ::= $\langle function\text{-}id \rangle$ '`(`' '`)`'
   $\gamma(\langle stmt \rangle) := Pr_S(f)$ with $f \in F$ and $\eta(\langle function\text{-}id \rangle) = f$

(b) $\langle stmt \rangle$ ::= $\langle var\text{-}id \rangle$ ':=' 'true'
$\gamma(\langle stmt \rangle) := 1$, $\vartheta(\langle stmt \rangle) := \{\langle \eta(\langle var\text{-}id \rangle), 1 \rangle\}$

(c) $\langle stmt \rangle$ ::= $\langle var\text{-}id \rangle$ ':=' 'false'
$\gamma(\langle stmt \rangle) := 1$, $\vartheta(\langle stmt \rangle) := \{\langle \eta(\langle var\text{-}id \rangle), 0 \rangle\}$

(d) $\langle stmt \rangle$ ::= $\langle var\text{-}id \rangle$ '=' $\langle function\text{-}id \rangle$ '(' ')'
$\gamma(\langle stmt \rangle) := 1$, $\vartheta(\langle stmt \rangle) := \{\langle \eta(\langle var\text{-}id \rangle), Pr_T(\eta(\langle function\text{-}id \rangle)) \rangle\}$ with $f \in F$ and $\eta(\langle function\text{-}id \rangle) = f$

(e) $\langle stmt \rangle$ ::= 'if' $\langle cond \rangle$ 'then' $\langle stmtlist_0 \rangle$ 'else' $\langle stmtlist_1 \rangle$ 'endif'
$\gamma(\langle stmt \rangle) := \gamma(\langle stmtlist_0 \rangle) \cdot \delta(\langle cond \rangle) + \gamma(\langle stmtlist_1 \rangle) \cdot (1 - \delta(\langle cond \rangle))$

(f) $\langle stmt \rangle$ ::= 'while' $\langle cond \rangle$ 'do' $\langle stmtlist \rangle$ 'endwhile'
$$\gamma(\langle stmt \rangle) := \frac{1 - \delta(\langle cond \rangle)}{1 - \delta(\langle cond \rangle) \cdot \gamma(\langle stmtlist \rangle)}$$

4. $\langle cond \rangle$ ::= ...
$\delta(\langle cond \rangle) := Pr_T(e)$, with $\eta(\langle cond \rangle) = e$

We now show how to perform probabilistic verification of reliability properties with SiDECAR on the two versions of the example workflow of Fig. 4a. In the steps of attribute synthesis, for brevity, we use numbers to refer to corresponding nodes in the syntax tree of Fig. 4b. As for the reliability of the two functions used in the workflow, we assume $Pr_S(opA) = .97$, $Pr_S(opB) = .99$.

Example Workflow - Version 1 Given the abstract syntax tree in Fig. 4b, evaluation of attributes leads to the following values (shown top to bottom, left to right, with η attributes omitted):

$\gamma(2) := .97;$ | $\gamma(14) := \gamma(15);$ | $\gamma(10) := \gamma(11);$
$\gamma(6) := 1;$ | $\gamma(19) := .97;$ | $\gamma(5) := (\gamma(6) \cdot \gamma(10)) \mid \vartheta(6) = .99;$
$\vartheta(6) := \{\langle x, 1 \rangle\};$ | $\gamma(18) := \gamma(19);$ | $\gamma(1) := \gamma(2) \cdot \gamma(5) = .9603;$
$\delta(12) := Pr_T(\texttt{"x==true"});$ | $\gamma(11) := .99 \cdot \delta(12)$ | $\gamma(0) := \gamma(1) = .9603.$
$\gamma(15) := .99;$ | $\quad + .97 \cdot (1 - \delta(12));$ |

The resulting value for $\gamma(0)$ represents the reliability of the workflow, i.e., each execution has a probability equal to .9603 of being successfully executed.

Example Workflow - Version 2 Version 2 of the example workflow differs from version 1 only in the assignment at line 3, which leads the incremental parser to build the subtree shown in the box of Fig. 4b. Because the matching condition is satisfied, this subtree is hooked into node 6 of the original tree. Re-computation of the attributes proceeds upward to the root, leading to the following final values (shown top to bottom, left to right, with η attributes omitted):

$\gamma(6) := 1;$ | $\gamma(5) := (\gamma(6) \cdot \gamma(10)) \mid \vartheta(6) = .97;$ | $\gamma(0) := \gamma(1) := .9409.$
$\vartheta(6) := \{\langle x, 0 \rangle\};$ | $\gamma(1) := \gamma(2) \cdot \gamma(5) = .9409;$ |

Thus, our incremental approach requires to reparse only 3 nodes and reevaluate only 5 attributes instead of the 13 ones computed in a full, non-incremental (re)parsing (as for Version 1).

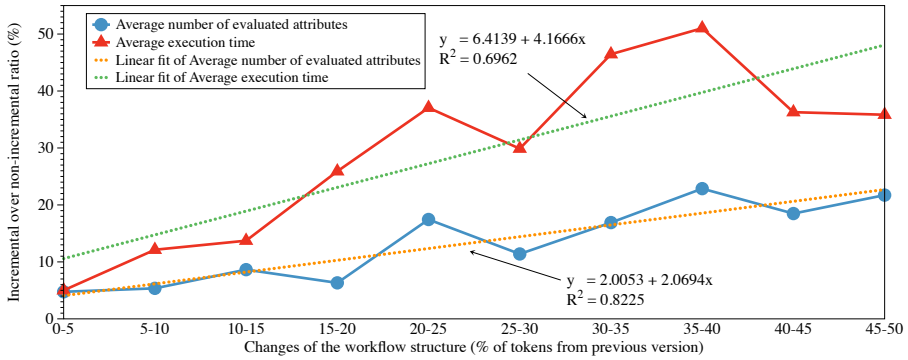


Fig. 5. Comparison between the incremental verification approach and the non-incremental one

5 Evaluation

To show the effectiveness of our approach, we performed a preliminary experimental evaluation using a prototype developed in Java, on a Intel Xeon E31220 3.10Ghz CPU, with 32Gb of RAM, running Ubuntu Server 12.04 64bit. We generated 56 random *Mini* workflows, each one with about 10000 tokens. For each workflow we randomly generated 30 subsequent versions, applying a series of deletions and/or insertions of syntactically valid code snippets, ranging in total from 5% to 50% of the workflow size. We run the probabilistic verification procedure defined above on all generated versions, both in an incremental way and in a non-incremental one. For each run, we measured the *number of evaluated attributes* and the *execution time*. Figure 5 shows the average of the ratio between the performance of the incremental approach over the non-incremental one, for both metrics. The results show that the execution time of our incremental approach is linear with respect to the size of the change(s), as expected from Sect. 3.1: the smaller the changes on the input program are, the faster the incremental approach is than the non-incremental one. This preliminary evaluation shows a 20x speedup of the incremental approach over the non-incremental one, for changes affecting up to 5% of the input artifact (having a total size of just 10^4 tokens); in the case of changes affecting about 50% of the code, we measured a 3x speedup. Since for large, long-lasting systems it is expected that most changes only involve a small fraction of the code, the gain of applying our incremental approach can be significant.

The parsing algorithm used within our framework has a temporal complexity (on average) linear in the size of the modified portion of the syntax tree. Hence any change in the workflow has a minimal impact on the adaptation of the abstract syntax tree too. Semantic incrementality allows for minimal (re)evaluation of the attributes, by proceeding along the path from the node corresponding to the change to the root, whose length is normally logarithmic with respect to the length of the workflow description. Notice that even if the change in a statement affects the execution of another location of the code (e.g., an assignment to a global variable), such dependency would be automatically handled in the least common ancestor of the two syntactic nodes. Such common

ancestor is, in the worst case, the root, resulting in the cost for the change propagation (in terms of re-evaluation of the attributes) being still logarithmic in the length of the workflow description.

We also analyzed each version of each workflow with Prism v.4.1, a probabilistic model checker. Our incremental verification approach was, on average, 4268 times faster than Prism, with a speedup of at least 1000x in about 35% of the workflows versions. We remark that Prism is a general-purpose verification tool that supports various types of input models (more complex than those needed to model structured workflows). Moreover, one can verify with Prism several properties more expressive than the simple reliability. However, the reason for this comparison is that many reliability analysis approaches (see also next section) make use of probabilistic model checking, which ultimately impacts on their performance.

6 Related Work

Reliability analysis of workflow has been widely investigated in the last decade. Most of the proposed approaches are based on algebraic methods [28], graph manipulation [11], or stochastic modeling [24, 33, 21, 8, 34, 27]. To the best of our knowledge, the only approach explicitly formalized by means of an attribute grammar is [13]. Nevertheless, only few approaches provide incrementality, at least to some extent; they are mainly grounded in the concepts of change encapsulation and of change anticipation [22].

Incrementality by *change encapsulation* is achieved by applying compositional reasoning to a modularized system using the assume-guarantee [29] paradigm. This paradigm views systems as a collection of cooperating modules, each of which has to guarantee certain properties. The verification methods based on this paradigm are said to be compositional, since they allow reasoning about each module separately and deducing properties about their integration. If the effect of a change can be localized inside the boundaries of a module, the other modules are not affected, and their verification does not need to be redone. This feature is for example exploited in [9], which proposes a framework for performing assume-guarantee reasoning in an incremental and fully automatic fashion.

Approaches based on *change anticipation* assume that the artifact under analysis can be divided into static (unchangeable) parts and variable ones, and rely on *partial evaluation* [14] to postpone the evaluation of the variable parts. Partial evaluation can be seen as a transformation from the original version of the program to a new version called *residual program*, where the properties of interest have been partially computed against the static parts, preserving the dependency on the variable ones. As soon as a change is observed, the computation can be moved a further step toward completion by fixing one or more variable parts according to the observations.

The above approaches, however, are based on the assumption that engineers know a priori the parts that are most likely subject to future evolution and can encapsulate them within well-defined borderlines. Our approach, instead, does not make any hypothesis on where changes will occur during system's life: it simply evaluates a posteriori their scope within system's structure as formalized by the syntax tree. This should be par-

ticularly beneficial in most modern systems that evolve in a fairly unpredictable way, often without a unique design responsibility.

Focusing on incremental probabilistic verification, the three main techniques supporting incremental verification of stochastic models (e.g., Markov Chains) are *decomposition* [32], which belongs to the class of change encapsulation, and *parametric analysis* [11, 26] and *delta evaluation* [33], which can be classified as change anticipation techniques. The first decomposes the input model into its strongly connected components (SCCs), allowing verification subtasks to be carried on within each SCC; local results are then combined to verify the global property. By defining a dependency relation among SCCs, when a change occurs, only the SCCs depending on the changed one have to be verified. The benefits of incrementality in this case depend on the quality of the SCC partition and the corresponding dependency relation. In the case of parametric analysis the probability value of the transitions in the model that are supposed to change are labeled with symbolic parameters. The model is then verified providing results in the form of closed mathematical formulae having the symbolic parameters as unknowns. As the actual values for the parameters become available (e.g., during the execution of the system), they are replaced in the formulae, providing a numerical estimation of the desired property (e.g. system reliability). Whenever the values of the parameters change, the closed formula obtained by the preprocessing phase can be reused, with significant improvements of the verification time [17, 15]. The main limitation of this approach is that a structural change in the software (i.e., not describable by a parameters assignment) invalidates the results of the preprocessing phase, requiring the verification to start from scratch, with consequent degradation of the analysis performance. Delta evaluation is concerned with incremental reliability analysis based on conveniently structured Discrete Time Markov Chains (DTMC). The structure of those model follows the proposal by [8], where each software module (represented by a state of the DTMC), can transfer the control to another module, or fail by making a transition toward an absorbing failure state, or complete the execution by moving toward an absorbing success state. Assuming that a single module failure probability changes at a time, only few arithmetic operations are needed to correct the previous reliability value. Despite its efficiency, delta evaluation can only deal with changes in a modules failure probability, providing no support for both structural changes and changes in the interaction probabilities among modules. Finally, in [28] service compositions are formalized through a convenient algebraic structure and an incremental framework is used to compose local results into a global quantitative property, in an assume-guarantee flavor. The approach is widely applicable for verification of component-based systems, and it has been applied for reliability analysis. The compositionality entailed by the assume-guarantee infrastructure can be recasted into our syntactic-semantic approach.

7 Conclusion and Future Work

Incrementality is one of the most promising means to dealing with software evolution. In this paper we addressed the issue of incrementality in the context of the probabilistic verification of reliability requirements of structured workflows. We defined a novel incremental technique based on a syntactic-semantic approach: the verification procedure

is encoded as synthesis of semantic attributes associated with the grammar defining the structure of workflows. As confirmed by the preliminary experimental evaluation, the execution time of our incremental approach is linear with respect to the size of the change(s). When changes involve only a small fraction of the artifact to analyze, our approach can provide a significant speedup over a non-incremental approach.

In the future, we plan to extend our approach to support richer workflow languages, such as BPEL and BPMN, as well as other types of verification procedures. The first direction will require to express the grammar of the workflow languages in an OPG form, with the possible caveat of reducing the readability of the grammar and impacting on the definition of the attribute schemas. As for the second direction, we plan to investigate richer attribute schemas, to support both new language features and different verification algorithms (e.g., to support more realistic assumptions on the system under verification as well as state-of-the-art optimizations and heuristics). Finally, we plan to apply our approach to the related problem of probabilistic symbolic execution [19, 6, 16]. In all these scenarios incrementality would be automatically provided by our SiDECAR framework, without any further effort for the developer.

Acknowledgments. This work has been partially supported by the European Community under the IDEAS-ERC grant agreement no. 227977-SMScom and by the National Research Fund, Luxembourg (FNR/P10/03). We thank Alessandro Maria Rizzi for helping with the implementation of the prototype.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. 1(1), 11–33 (2004)
2. Barenghi, A., Viviani, E., Crespi Reghizzi, S., Mandrioli, D., Pradella, M.: PAPAGENO: a parallel parser generator for operator precedence grammars. In: Proc. of SLE 2012. LNCS, vol. 7745, pp. 264–274. Springer (2012)
3. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issues and challenges. IEEE Computer 39(10), 36–43 (2006)
4. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D.: A syntactic-semantic approach to incremental verification (2013), `http://arxiv.org/abs/1304.8034`
5. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D.: Syntactic-semantic incrementality for agile verification. Sci. Comput. Program. N/A(N/A) (2013), DOI:10.1016/j.scico.2013.11.026
6. Borges, M., Filieri, A., d'Amorim, M., Păsăreanu, C.S., Visser, W.: Compositional solution space quantification for probabilistic software analysis. In: Proc. of PLDI 2014. pp. 123–132. ACM (2014)
7. de Bosschere, K.: An operator precedence parser for standard Prolog text. Softw. Pract. Exper. 26(7), 763–779 (1996)
8. Cheung, R.: A user-oriented software reliability model. IEEE Trans. Soft. Eng. SE-6(2), 118–125 (1980)
9. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Proc. of TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer (2003)
10. Crespi Reghizzi, S., Mandrioli, D.: Operator precedence and the visibly pushdown property. J. Comput. Syst. Sci. 78(6), 1837–1867 (2012)

11. Daws, C.: Symbolic and parametric model checking of discrete-time markov chains. In: Proc. of ICTAC 2004. LNCS, vol. 3407, pp. 280–294. Springer (2005)
12. Distefano, S., Ghezzi, C., Guinea, S., Mirandola, R.: Dependability assessment of web service orchestrations. IEEE Trans. Rel. PrePrint(N/A) (2014), DOI:10.1109/TR.2014.2315939
13. Distefano, S., Filieri, A., Ghezzi, C., Mirandola, R.: A compositional method for reliability analysis of workflows affected by multiple failure modes. In: Proc. of CBSE 2011. pp. 149–158. ACM (2011)
14. Ershov, A.: On the partial computation principle. Inform. Process. Lett. 6(2), 38–41 (1977)
15. Filieri, A., Ghezzi, C.: Further steps towards efficient runtime verification: Handling probabilistic cost models. In: Proc. of FormSERA 2012. pp. 2–8. IEEE (2012)
16. Filieri, A., , Păsăreanu, C.S., Visser, W., Geldenhuys, J.: Statistical symbolic execution with informed sampling. In: Proc. of SIGSOFT '14/FSE-22. ACM (2014)
17. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proc. of ICSE 2011. pp. 341–350. ACM (2011)
18. Filieri, A., Ghezzi, C., Tamburrelli, G.: A formal approach to adaptive software: continuous assurance of non-functional requirements. Formal Asp. Comput. 24(2), 163–186 (2012)
19. Filieri, A., Păsăreanu, C.S., Visser, W.: Reliability analysis in symbolic pathfinder. In: Proc. of ICSE 2013. pp. 622–631. IEEE Press (2013)
20. Floyd, R.W.: Syntactic analysis and operator precedence. J. ACM 10, 316–333 (1963)
21. Gallotti, S., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Quality prediction of service compositions through probabilistic model checking. In: Proc. of QOSA 2008. LNCS, vol. 5281, pp. 119–134. Springer (2008)
22. Ghezzi, C.: Evolution, adaptation, and the quest for incrementality. In: Proc. of the 17th Monterey Workshop. LNCS, vol. 7539, pp. 369–379. Springer (2012)
23. Ghezzi, C., Mandrioli, D.: Incremental parsing. ACM Trans. Program. Lang. Syst. 1(1), 58–70 (1979)
24. Goseva-Popstojanova, K., Mathur, A., Trivedi, K.: Comparison of architecture-based software reliability models. In: Proc. of ISSRE 2001. pp. 22–31. IEEE (2001)
25. Grune, D., Jacobs, C.J.H.: Parsing Techniques - a practical guide. Springer, 2nd edn. (2008)
26. Hahn, E., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric markov models. STTT 13(1), 3–19 (2011)
27. Immonen, A., Niemela, E.: Survey of reliability and availability prediction methods from the viewpoint of software architecture. Software and Systems Modeling 7(1), 49–65 (2008)
28. Johnson, K., Calinescu, R., Kikuchi, S.: An incremental verification framework for component-based software systems. In: Proc. of CBSE '13. pp. 33–42. ACM (2013)
29. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
30. Josuttis, N.: SOA in Practice: The Art of Distributed System Design. O'Reilly (2007)
31. Knuth, D.E.: Semantics of context-free languages. Theory of Computing Systems 2, 127–145 (1968)
32. Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for markov decision processes. In: Proc. of DSN 2011. pp. 359–370. IEEE (2011)
33. Meedeniya, I., Grunske, L.: An efficient method for architecture-based reliability evaluation for evolving systems with changing parameters. In: Proc. of ISSRE 2010. pp. 229–238. IEEE (2010)
34. Pham, H.: System software reliability. Springer (2006)
35. Sistla, P.: Hybrid and incremental model-checking techniques. ACM Comput. Surv. 28(4es) (1996)
36. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases 14(1), 5–51 (Jul 2003)