A Syntactic-Semantic Approach to Incremental Verification

Domenico Bianculli
SnT Centre, University of Luxembourg, Luxembourg, Luxembourg
domenico.bianculli@uni.lu

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

Carlo Ghezzi
DEEPSE group - DEI, Politecnico di Milano, Milano, Italy
ghezzi@elet.polimi.it

Dino Mandrioli
DEEPSE group - DEI, Politecnico di Milano, Milano, Italy
mandrioli@elet.polimi.it

April 30, 2013

Abstract

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.

This paper introduces SiDECAR, a general framework for the definition of verification procedures, which are made incremental by the framework itself. Verification procedures are driven by the syntactic structure (defined by a grammar) of the system and encoded as semantic attributes associated with the grammar. Incrementality is achieved by coupling the evaluation of semantic attributes with an incremental parsing technique.

We show the application of SiDECAR to the definition of two verification procedures: probabilistic verification of reliability requirements and verification of safety properties.

Keywords: incremental verification; syntax-driven algorithms; attribute grammars; operator precedence grammars.

1 Introduction

Software evolution is a well-known phenomenon in software engineering. Software may evolve because of a change in the requirements or in the domain assumptions, leading to the development and deployment of many new versions of the software. This phenomenon is taken to extremes by new kinds of software, called *open-world software* [4], built by composing heterogeneous, third-party components, whose behavior and interactions cannot be fully controlled or predicted. This software is required to react to changes in its environment, by bringing verification to run time [7] and (self-) adapting its behavior while it is executing.

Incremental verification has been suggested as a possible approach to dealing with evolving software [34]. 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. Moreover, incremental verification may speed up change management, which may be subject to severe time constraints, especially if it needs to be performed at run time, to support dynamic self-adaptation.

This paper proposes SiDECAR (Syntax-DrivEn inCrementAl veRification), a general framework to define verification procedures, which are automatically enhanced with incrementality by the framework itself. The framework follows a syntactic-semantic approach, since it assumes that the software artifact to be verified has a syntactic structure described by a formal grammar, and that the verification procedure is encoded as synthesis of semantic attributes [28], associated with the grammar and evaluated by traversing the syntax tree of the artifact. We based the framework on operator precedence grammars [19], 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. 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.

The main contributions of the paper are: i) the definition of a methodological approach for incremental syntactic-semantic verification procedures (SiDECAR); ii) the application of SiDECAR to the definition of two verification procedures: probabilistic verification of reliability requirements and verification of safety properties. Indeed, the goal of the paper is to present the general framework, which can be used to define incremental verification procedures. The two examples are provided to show the generality and versatility of the 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 SiDECAR exploits operator precedence grammars to support syntactic-semantic incremental verification. In section 4 we show SiDECAR at work, by presenting the two examples. In section 5 we discuss the application of the methodology supported by SiDECAR. Section 6 presents related work. Section 7 provides some concluding remarks.

$$
\begin{aligned}
\langle S\rangle &::= \langle A\rangle \mid \langle B\rangle \\
\langle A\rangle &::= \langle A\rangle \text{ `+' } \langle B\rangle \mid \langle B\rangle \text{ `+' } \langle B\rangle \\
\langle B\rangle &::= \langle B\rangle \text{ `*' `n'} \mid \text{`n'}
\end{aligned}
$$

(a)

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

(b)

Figure 1: Example of an operator grammar ('n' stands for any natural number) and its operator precedence matrix

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 [21] and [12].

2.1 Operator precedence Grammars

We start by recalling the definition of a *context-free (CF)* grammar G as 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 $\langle A\rangle$; 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., [21].

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) [19] 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 in 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 Grammars) *An OG G is an OPG grammar if and only if its OPM is a conflict-free matrix, i.e., for each $a, b \in V_T$, $|m_{a,b}| \leq 1$.*

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

Figure 2: Example of attribute grammar

Definition 2 (Fischer Normal Form, from [12]) *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 (rhs) 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. 3 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 [28]. 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 [28].

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 Fig. 2, where semantic functions are enclosed in braces next to each rule: The $+$ and $*$ operators

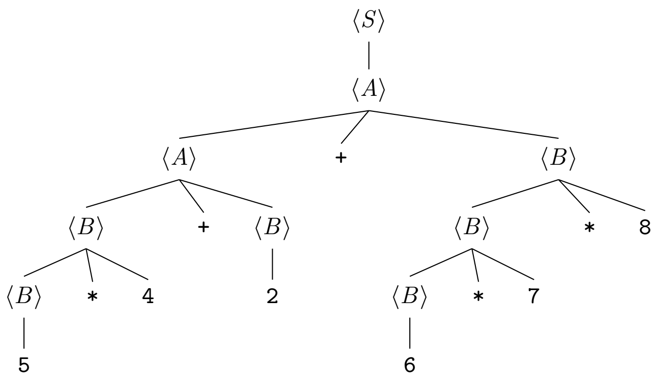

Figure 3: Abstract syntax tree of the expression '`5*4+2+6*7*8`'

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 SiDECAR and Syntactic-semantic Incrementality

SiDECAR 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 enjoy 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—by means of 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. 3. 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 $[\![$'`+`', '`#`'$]\!]$. The

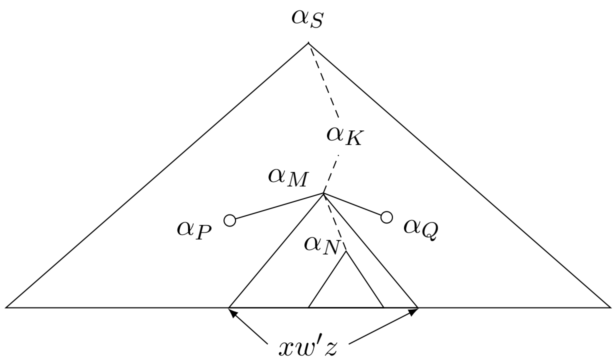

Figure 4: Incremental evaluation of semantic attributes

matching condition is satisfied by '+'$\langle B\rangle$'#' $\overset{*}{\Rightarrow}$ '+''6''*''7''*''8''#' and '+'$\langle B\rangle$'#' $\overset{*}{\Rightarrow}$ '+''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[1].

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 $\langle A\rangle$, $a\langle A\rangle b \overset{*}{\Rightarrow} at_1wu_1b$ and $a\langle A\rangle b \overset{*}{\Rightarrow} at_1w'u_1b$.

The locality property[2] 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 [21, p. 271]; a complete OPG is available for Prolog [14]. Moreover, in many practical cases one can obtain an OPG by minor adjustments to a non operator-precedence grammar [19].

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 the re-parsing of the modified substring. Suppose that, after replacing substring w with w', incremental re-parsing builds a derivation $\langle N\rangle \overset{*}{\Rightarrow} xw'z$, with the same non-terminal $\langle N\rangle$ as in $\langle N\rangle \overset{*}{\Rightarrow} xwz$, so that the matching condition is

[1] Some further optimization could be applied by integrating the matching condition with techniques adopted in [20] (not reported here for brevity).

[2] The locality property has also been shown to support an efficient parallel parsing technique [3], which is not further exploited here.

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. 4) 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 SiDECAR at work

Using SiDECAR requires to define 1) an OPG for the programming language one wants to support and 2) the associated attribute grammar schema corresponding to the verification procedures that one wants to implement. In this section we use programs written in the *Mini* language, whose OPG is shown in Fig. 5. It is a minimalistic language that includes the major constructs of structured programming. For the sake of readability and to reduce the complexity of attribute schemas, *Mini* programs support only (global) boolean variables and boolean functions (with no input parameters). These assumptions can be relaxed, with no impact on the applicability of our approach.

In the rest of this section we demonstrate the generality of the SiDECAR framework by means of two examples of incremental verification. The former one (Section 4.1) reports on probabilistic verification of reliability properties of programs that compose possibly faulty functions. The latter (Section 4.2) reports on verification of safety properties of programs. We chose two simple, but rather diverse examples to demonstrate SiDECAR's versatility as a general framework. For space reasons and for the sake of readability, we adopt a straightforward encoding of these verification procedures and make several simplifying assumptions. We deliberately omit all optimizations and heuristics that would improve the verification, which are adopted by state-of-the-art tools. Nevertheless these could be accommodated in SiDECAR through richer (and more complex) attributes.

To show the benefits of incrementality, for each of the verification procedures defined in the next subsections, we analyze two versions of the same example program (shown in Fig. 6), which 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 7 depicts the syntax tree of version 1 of the program, as well as the subtree that is different in version 2; nodes of the tree have been numbered for quick reference.

The next two subsections describe in detail the two analyses and their corresponding attribute schemas. Before presenting them, here we introduce some useful notations. Given a *Mini* program P, F_P is the set of functions and V_P the set of variables defined within P; E_P is the set of boolean expressions that can appear as the condition of an *if* or a *while* statement in P. An expression $e \in E_P$ is either a combination of boolean predicates on program variables or a placeholder predicate

$\langle S \rangle$ ::= ‘`begin`’ $\langle$*stmtlist*$\rangle$ ‘`end`’
$\langle$*stmtlist*$\rangle$ ::= $\langle$*stmt*$\rangle$ ‘`;`’ $\langle$*stmtlist*$\rangle$
  | $\langle$*stmt*$\rangle$ ‘`;`’
$\langle$*stmt*$\rangle$ ::= $\langle$*function-id*$\rangle$ ‘`(`’ ‘`)`’
  | $\langle$*var-id*$\rangle$ ‘`:=`’ ‘`true`’
  | $\langle$*var-id*$\rangle$ ‘`:=`’ ‘`false`’
  | $\langle$*var-id*$\rangle$ ‘`:=`’ $\langle$*function-id*$\rangle$ ‘`(`’ ‘`)`’
  | ‘`if`’ $\langle$*cond*$\rangle$ ‘`then`’ $\langle$*stmtlist*$\rangle$ ‘`else`’ $\langle$*stmtlist*$\rangle$ ‘`endif`’
  | ‘`while`’ $\langle$*cond*$\rangle$ ‘`do`’ $\langle$*stmtlist*$\rangle$ ‘`endwhile`’
$\langle$*var-id*$\rangle$ ::= ...
$\langle$*function-id*$\rangle$ ::= ...
$\langle$*cond*$\rangle$ ::= ...

Figure 5: The grammar of the *Mini* language

labeled $*$. Hereafter, we drop the subscript P in F_P, V_P, and E_P whenever the program is clear from the context.

4.1 Probabilistic Verification of Reliability Requirements

In this section we show how to apply SiDECAR to perform probabilistic verification of reliability requirements of *Mini* programs. Reliability is a “user-oriented” property [8]; in other words, 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 [2]; 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.

We observe that the verification problem presented here for *Mini* can be viewed as a high-level abstraction of a similar verification problem for service compositions in the context of service-oriented architectures, since the call to possibly faulty functions mimics the call to third-party services.

Most of the current approaches for verification of reliability requirements use probabilistic model checking [24,33]. Software systems are translated into stochastic models, such as Discrete Time Markov Chains (DTMCs), which are suitable to

```
begin
 opA();
 x := true;
 if (x==true)
  then opB();
  else opA();
 endif;
end
```

(a) Version 1

```
begin
 opA();
 x := false;
 if (x==true)
  then opB();
  else opA();
 endif;
end
```

(b) Version 2

Figure 6: The two versions of the example program

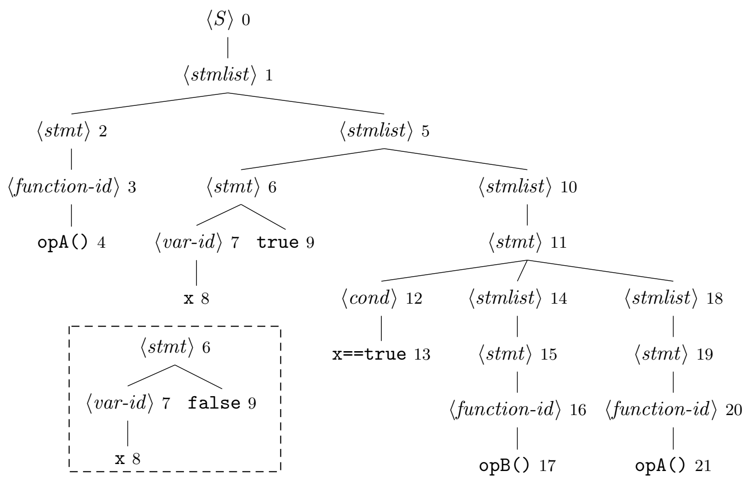

Figure 7: The syntax tree of version 1 of the example program; the subtree in the box shows the difference (node 9) in the syntax tree of version 2

represent usage profiles and failure probabilities. A DTMC is essentially a finite state automaton where states abstract the program execution state, such as the execution of a task or the occurrence of a failure, and the transitions among states are defined through a probabilistic distribution. DTMCs can be analyzed with probabilistic model checkers such as PRISM [30] and MRMC [27].

To model the probabilistic verification problem in SiDECAR, 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 at the rhs 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 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 program 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 easily derived by applying well-known properties of probability theory:

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

A different construction of this result can be found in [15].

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 is defined as follows:

1. $\langle S\rangle$ ::= '`begin`' $\langle stmtlist\rangle$ '`end`'
   $\gamma(\langle S\rangle) \coloneqq \gamma(\langle stmtlist\rangle)$
2. (a) $\langle stmtlist_0\rangle$ ::= $\langle stmt\rangle$ '`;`' $\langle stmtlist_1\rangle$
   $\gamma(\langle stmtlist_0\rangle) \coloneqq (\gamma(\langle stmt\rangle) \cdot \gamma(\langle stmtlist_1\rangle)) \mid \vartheta(\langle stmt\rangle)$
   (b) $\langle stmtlist\rangle$ ::= $\langle stmt\rangle$ '`;`'
   $\gamma(\langle stmtlist\rangle) \coloneqq \gamma(\langle stmt\rangle)$
3. (a) $\langle stmt\rangle$ ::= $\langle function\text{-}id\rangle$ '`(`' '`)`'
   $\gamma(\langle stmt\rangle) \coloneqq Pr_S(f)$
   with $f \in F$ and $\eta(\langle function\text{-}id\rangle) = f$
   (b) $\langle stmt\rangle$ ::= $\langle var\text{-}id\rangle$ '`:=`' '`true`'
   $\gamma(\langle stmt\rangle) \coloneqq 1$,
   $\vartheta(\langle stmt\rangle) \coloneqq \{\langle \eta(\langle var\text{-}id\rangle), 1\rangle\}$
   (c) $\langle stmt\rangle$ ::= $\langle var\text{-}id\rangle$ '`:=`' '`false`'
   $\gamma(\langle stmt\rangle) \coloneqq 1$,
   $\vartheta(\langle stmt\rangle) \coloneqq \{\langle \eta(\langle var\text{-}id\rangle), 0\rangle\}$
   (d) $\langle stmt\rangle$ ::= $\langle var\text{-}id\rangle$ '`=`' $\langle function\text{-}id\rangle$ '`(`' '`)`'
   $\gamma(\langle stmt\rangle) \coloneqq 1$,
   $\vartheta(\langle stmt\rangle) \coloneqq \{\langle \eta(\langle var\text{-}id\rangle), Pr_T(\eta(\langle function\text{-}id\rangle))\rangle\}$
   with $f \in F$ and $\eta(\langle function\text{-}id\rangle) = f$
   (e) $\langle stmt\rangle$ ::= '`if`' $\langle cond\rangle$ '`then`' $\langle stmlist_0\rangle$ '`else`' $\langle stmlist_1\rangle$ '`endif`'
   $$\gamma(\langle stmt\rangle) \coloneqq \gamma(\langle stmtlist_0\rangle) \cdot \delta(\langle cond\rangle) + \gamma(\langle stmtlist_1\rangle) \cdot (1 - \delta(\langle cond\rangle))$$
   (f) $\langle stmt\rangle$ ::= '`while`' $\langle cond\rangle$ '`do`' $\langle stmtlist\rangle$ '`endwhile`'
   $$\gamma(\langle stmt\rangle) \coloneqq \frac{1 - \delta(\langle cond\rangle)}{1 - \delta(\langle cond\rangle) \cdot \gamma(\langle stmtlist\rangle)}$$
4. $\langle cond\rangle$ ::= ...
   $\delta(\langle cond\rangle) \coloneqq Pr_T(e)$, 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 program of Fig. 6. In the steps of attribute synthesis, for brevity, we use numbers to refer to corresponding nodes in the syntax tree of Fig. 7. As for the reliability of the two functions used in the program, we assume $Pr_S(opA) = .97$, $Pr_S(opB) = .99$.

Example Program - Version 1

Given the abstract syntax tree in Fig. 7, evaluation of attributes leads to the following values: (η attributes omitted):

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

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

Example Program - Version 2

Version 2 of the example program 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. 7. 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:

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

In conclusion, this example shows that SiDECAR re-analyzes only a limited part of the program and re-computes only a small subset of the attributes.

4.2 Verification of Safety Properties

This section shows how to use SiDECAR to define a basic software model checking procedure, which solves the safety verification problem: given a program and a safety property, we want to decide whether there is an execution of the program that leads to a violation of the property.

In software model checking, it is common to use a transition-relation representation of programs [25], in which a program is characterized by a set of (typed) variables, a set of control locations (including an initial one), and a set of transitions, from a control location to another one, labeled with constraints on variables and/or with program operations. Examples of this kind of representation are control-flow graphs [1] and control-flow automata [5]. A state of the program is characterized by a location and by the valuation of the variables at that location. A computation of the program is a (finite or infinite) sequence of states, where the sequence is induced by the transition relation over locations. Checking for a safety property can be reduced to the problem of checking for the reachability of a particular location, the *error* location, for example, by properly instrumenting the program code according to the safety specification.

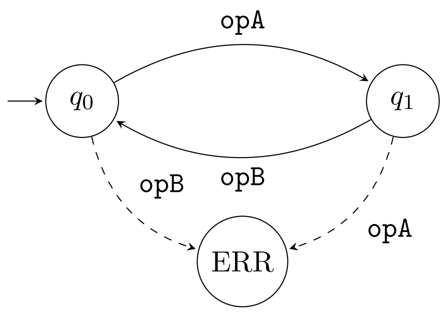

Figure 8: A property automaton; dashed lines belong to the corresponding image automaton

In the implementation of safety verification with SiDECAR we assume that the property is defined as a *property automaton* [9], whose transitions correspond either to a procedure call or to a function call that assigns a value to a variable. From this automaton we then derive the corresponding *image automaton*, which traps violation of the property in an error location (called *ERR*).

Formally, let VA be the set of variable assignments from functions, i.e., $\mathit{VA} = \{x := f \mid x \in V \text{ and } f \in F\}$. A property automaton A is a quadruple $A = \langle S, T, \delta, s_0 \rangle$ where S is a set of locations, T is the alphabet $T = F \cup \mathit{VA}$, δ is the transition function $\delta \colon S \times T \to S$, and s_0 is the initial location. Given a property automaton A, the corresponding image automaton A' is defined as $A' = \langle S \cup \{\mathit{ERR}\}, T, \delta', s_0 \rangle$, where $\delta' = \delta \cup \{(s, t, \mathit{ERR}) \mid (s, t) \in S \times T \wedge \neg \exists s' \in S \mid (s, t, s') \in \delta\}$. An example of a property automaton specifying the alternation of operations `opA` and `opB` on sequences starting with `opA` is depicted in Fig. 8; transitions drawn with a dashed line are added to the property automaton to obtain its image automaton.

Instead of analyzing the program code instrumented with the safety specification, we check for the reachability of the error location in an execution trace of the image automaton, as induced by the syntactic structure of the program.

More specifically, each location of the automaton is paired with a configuration of the program, which consists of a mapping of the program variables and of the traversal conditions for the paths taken so far. A configuration is invalid if the set of predicate conditions holding at a certain location of the program are not compatible with the current variables mapping for that location. Formally, let $\mathit{VM} : V \mapsto \{\mathit{true}, \mathit{false}\}$ be a mapping from program variables to their value (if defined). The set of possibile configurations that can be reached during the execution of a program is denoted by $C = (\mathit{VM} \times E) \cup \{\bot\}$, where E is the set of boolean expressions that can appear as the condition of an *if* or a *while* statement and $\bot$ stands for an invalid configuration.

Configurations of the program may change when variables are assigned a new value, e.g., by a direct assignment of a literal or by assigning the return value of a function. We use a function *upd* that updates a configuration and checks whether it is valid or not. The function *upd* is defined as $\mathit{upd} \colon (C \times V \cup \{\varepsilon\} \times \{\mathit{true}, \mathit{false}\} \cup \{\varepsilon\} \times E \cup \{\varepsilon\} \times \{\mathit{true}, \mathit{false}\} \cup \{\varepsilon\}) \to C$. The function takes a configuration, a variable, its new value, a combination of boolean expressions (corresponding to a certain path condition), its new value, and returns the new configuration; the ε symbol accounts for empty parameters.

We call the pair ⟨location of the image automaton, configuration of the program⟩ an *extended state*. A safety property represented as an image automaton is violated if it is possible to reach from the initial extended state another extended state whose location component is the *ERR* location. Each statement in the program defines a transition from one extended state to another.

For example, a procedure call determines the location component in an extended state by following the transition function of the image automaton corresponding to the call. An assignment to a variable updates the program configuration component of an extended state. In case a variable is assigned the return value of a function invocation, both components of an extended state are updated.

Conditions in selection and loop statements are evaluated and the program configuration of the corresponding extended state is updated accordingly, to keep track of which path conditions have been taken. For an *if* statement, we keep track of which extended states could be reachable by executing the statement, considering both the *then* branch and the *else* branch. For a *while* statement, we make the common assumption that a certain constant K is provided to indicate the number of unrolling passes of the loop. We then keep track of which extended states could be reachable, both in case the loop is not executed and in case the loop is executed K times.

The set of attributes is defined as:

- $SYN(\langle S\rangle) = SYN(\langle \mathit{stmlist}\rangle) = SYN(\langle \mathit{stmt}\rangle) = \{\gamma\}$;
- $SYN(\langle \mathit{cond}\rangle) = \{\gamma, \nu\}$;
- $SYN(\langle \mathit{var\text{-}id}\rangle) = SYN(\langle \mathit{function\text{-}id}\rangle) = \{\eta\}$;

where:

- $\gamma \subseteq S \times C \times S \times C$ is the relation that defines a transition from one extended state to another one;
- ν is a string corresponding to the literal value of an expression $e \in E$;
- η is a string corresponding to the literal value of an identifier.

For the γ attribute of $\langle \mathit{cond}\rangle$ we use the symbol γ^T (respectively γ^F) to denote the attribute γ evaluated when the condition $\langle \mathit{cond}\rangle$ is *true* (respectively, *false*). We also define the operation of composing γ relations (the $\circ$ operator) as follows: $\gamma_1 \circ \gamma_2 = \langle s_1, c_1, s_2, c_2\rangle$ such that there exist $\langle s_1, c_1, s_i, c_i\rangle \in \gamma_1$ and $\langle s_i, c_i, s_2, c_2\rangle \in \gamma_2$.

The attribute schema is defined as follows, where we use the symbols s, s_1, s_2 and c, c_1, c_2 to denote generic elements in S and C, respectively.

1. $\langle S\rangle$ ::= '`begin`' $\langle \mathit{stmtlist}\rangle$ '`end`'
   $\gamma(\langle S\rangle) \coloneqq \gamma(\langle \mathit{stmtlist}\rangle)$
2. (a) $\langle \mathit{stmtlist_0}\rangle$ ::= $\langle \mathit{stmt}\rangle$ '`;`' $\langle \mathit{stmtlist_1}\rangle$
   $\gamma(\langle \mathit{stmtlist_0}\rangle) \coloneqq \gamma(\langle \mathit{stmt}\rangle) \circ \gamma(\langle \mathit{stmtlist_1}\rangle)$
   (b) $\langle \mathit{stmtlist}\rangle$ ::= $\langle \mathit{stmt}\rangle$ '`;`'
   $\gamma(\langle \mathit{stmtlist}\rangle) \coloneqq \gamma(\langle \mathit{stmt}\rangle)$
3. (a) $\langle \mathit{stmt}\rangle$ ::= $\langle \mathit{function\text{-}id}\rangle$ '`(`' '`)`'
   $\gamma(\langle \mathit{stmt}\rangle) \coloneqq \langle s_1, c, s_2, c\rangle$ such that there is $f \in F$ with $\delta(s_1, f) = s_2$ and $\eta(\langle \mathit{function\text{-}id}\rangle) = f$
   (b) $\langle \mathit{stmt}\rangle$ ::= $\langle \mathit{var\text{-}id}\rangle$ '`:=`' '`true`'
   $\gamma(\langle \mathit{stmt}\rangle) \coloneqq \langle s, c_1, s, c_2\rangle$ with $c_2 = \mathit{upd}(c_1, \eta(\langle \mathit{var\text{-}id}\rangle), \mathit{true}, \varepsilon, \varepsilon)$

(c) $\langle stmt \rangle$ ::= $\langle var\text{-}id \rangle$ ‘`:=`’ ‘`false`’
$\gamma(\langle stmt \rangle) \coloneqq \langle s, c_1, s, c_2 \rangle$ with $c_2 = upd(c_1, \eta(\langle var\text{-}id \rangle), false, \varepsilon, \varepsilon)$

(d) $\langle stmt \rangle$ ::= $\langle var\text{-}id \rangle$ ‘`=`’ $\langle function\text{-}id \rangle$ ‘`(`’ ‘`)`’
$\gamma(\langle stmt \rangle) \coloneqq \langle s_1, c_1, s_2, c_2 \rangle \cup \langle s_1, c_1, s_2, c_3 \rangle$ such that there is $f \in F$ with $\delta(s_1, f) = s_2$, $\eta(\langle function\text{-}id \rangle) = f$, $c_2 = upd(c_1, \eta(\langle var\text{-}id \rangle), true, \varepsilon, \varepsilon)$, and $c_3 = upd(c_1, \eta(\langle var\text{-}id \rangle), false, \varepsilon, \varepsilon)$

(e) $\langle stmt \rangle$ ::= ‘`if`’ $\langle cond \rangle$ ‘`then`’ $\langle stmlist_0 \rangle$ ‘`else`’ $\langle stmlist_1 \rangle$ ‘`endif`’
$\gamma(\langle stmt \rangle) \coloneqq \gamma^T(\langle cond \rangle) \circ \gamma(\langle stmtlist_0 \rangle) \cup \gamma^F(\langle cond \rangle) \circ \gamma(\langle stmtlist_1 \rangle)$

(f) $\langle stmt \rangle$ ::= ‘`while`’ $\langle cond \rangle$ ‘`do`’ $\langle stmtlist \rangle$ ‘`endwhile`’
$\gamma(\langle stmt \rangle) \coloneqq \gamma^{body} \circ \gamma^F(\langle cond \rangle)$ where $\gamma^{body} = \left(\gamma^T(\langle cond \rangle) \circ \gamma(\langle stmtlist \rangle)\right)^K$

4. $\langle cond \rangle$ ::= ...
$\gamma(\langle cond \rangle) \coloneqq \gamma^T(\langle cond \rangle) \cup \gamma^F(\langle cond \rangle) =$
$\langle s, c_1, s, c_2 \rangle \cup \langle s, c_1, s, c_3 \rangle$ where $c_2 = upd(c_1, \varepsilon, \varepsilon, \nu(\langle cond \rangle), true)$ and $c_3 = upd(c_1, \varepsilon, \varepsilon, \nu(\langle cond \rangle), false)$

We now show how to perform safety verification with SiDECAR on the two versions of the example program. For both examples, we consider the safety property specified with the automaton in Fig. 8.

Example Program - Version 1

Given the abstract syntax tree depicted in Fig. 7, attributes are synthesized as follows:

$\gamma(2) \coloneqq \{\langle q_0, c, q_1, c \rangle, \langle q_1, c, ERR, c \rangle\}$;
$\gamma(6) \coloneqq \langle s, c_1, s, upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon) \rangle$;
$\gamma(12) \coloneqq \gamma^T(12) \cup \gamma^F(12) \coloneqq \langle s, c_1, s, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle \cup$
$\langle s, c_1, s, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle$;
$\gamma(15) \coloneqq \{\langle q_1, c, q_0, c \rangle, \langle q_0, c, ERR, c \rangle\}$;
$\gamma(14) \coloneqq \gamma(15)$;
$\gamma(19) \coloneqq \{\langle q_0, c, q_1, c \rangle, \langle q_1, c, ERR, c \rangle\}$;
$\gamma(18) \coloneqq \gamma(19)$;
$\gamma(11) \coloneqq \gamma^T(12) \circ \gamma(14) \cup \gamma^F(12) \circ \gamma(18) \coloneqq$
$\langle s, c_1, s, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle \circ \{\langle q_1, c, q_0, c \rangle, \langle q_0, c, ERR, c \rangle\} \cup$
$\langle s, c_1, s, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle \circ \{\langle q_0, c, q_1, c \rangle, \langle q_1, c, ERR, c \rangle\} \coloneqq$
$\{\langle q_1, c_1, q_0, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle, \langle q_0, c_1, ERR, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle,$
$\langle q_0, c_1, q_1, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle, \langle q_1, c_1, ERR, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle\}$;
$\gamma(10) \coloneqq \gamma(11)$;
$\gamma(5) \coloneqq \gamma(6) \circ \gamma(10) \coloneqq \{\langle q_1, c_1, q_0, upd(upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle,$
$\langle q_0, c_1, ERR, upd(upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle,$
$\langle q_0, c_1, q_1, \perp \rangle, \langle q_1, c_1, ERR, \perp \rangle\}$.

The last two tuples of $\gamma(5)$ are discarded because they contain a $\perp$ configuration. $\perp$ is returned by *upd*; according to its semantics, the evaluation of the condition `"x==true"` to *false* is not compatible with the previous configuration, where `x` is assigned the value *true*. Hence, we have:

$\gamma(5) \coloneqq \{\langle q_1, c_1, q_0, upd(upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle,$
$\langle q_0, c_1, ERR, upd(upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle\}$;
$\gamma(1) \coloneqq \gamma(2) \circ \gamma(5) \coloneqq \langle q_0, c, q_0, upd(upd(c, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle$;

$\gamma(0) := \gamma(1) := \langle q_0, c, q_0, upd(upd(c, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true) \rangle$.
The resulting $\gamma(0)$ shows that the error location is not reachable from the initial extended state. Therefore we can conclude that the property will not be violated by any execution of the program.

Example Program - Version 2

The change in version 2 of the example program affects node 9 of the subtree shown in the box of Fig. 7. Attribute evaluation proceeds from node 6 up to the root, as shown below:
$\gamma(6) := \langle s, c_1, s, upd(c_1, \texttt{"x"}, false, \varepsilon, \varepsilon) \rangle$;
$\gamma(5) := \gamma(6) \circ \gamma(10) := \{ \langle q_1, c_1, q_0, \perp \rangle,\ \langle q_0, c_1, ERR, \perp \rangle,$
$\langle q_0, c_1, q_1, upd(upd(c_1, \texttt{"x"}, false, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle$
$\langle q_1, c_1, ERR, upd(upd(c_1, \texttt{"x"}, false, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle\ \}$.
The first two tuples of $\gamma(5)$ are discarded because they contain a $\perp$ configuration. $\perp$ is returned by *upd*; according to its semantics, the evaluation of the condition `"x==true"` to *true* is not compatible with the previous configuration, where `x` is assigned the value *false*. Hence, we have:
$\gamma(5) := \{ \langle q_0, c_1, q_1, upd(upd(c_1, \texttt{"x"}, false, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle,$
$\langle q_1, c_1, ERR, upd(upd(c_1, \texttt{"x"}, false, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle \}$;
$\gamma(1) := \gamma(2) \circ \gamma(5) := \langle q_0, c, ERR, upd(upd(c, \texttt{"x"}, false, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle$;
$\gamma(0) := \gamma(1) := \langle q_0, c, ERR, upd(upd(c, \texttt{"x"}, false, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, false) \rangle$.
By looking at $\gamma(0)$, we notice that the error location is now reachable, which means that version 2 of the program violates the safety property.

Note that we reuse results from the analysis of version 1, since $\gamma(10)$ and $\gamma(2)$ have not changed. In the analysis of version 2 we processed only 7 tuples of the state space, compared with the 26 ones processed for version 1.

5 Discussion

SiDECAR introduces a general methodology for the definition of incremental verification procedures. It has only two usage requirements: R1) the artifact to be verified should have a syntactic structure derivable from an OPG; R2) the verification procedure has to be formalized as synthesis of semantic attributes.

The parsing algorithm used within SiDECAR has a temporal complexity (on average) linear in the size of the modified portion of the syntax tree. Hence any change in the program 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 program. Thus the use of SiDECAR may result in a significant reduction of the re-analysis and semantic re-evaluation steps. The saving could be very relevant in the case of large programs and rich and complex attribute schemas.

We emphasize that the two examples showed in the previous section were not designed to be directly applied to real-world software verification, but to give an intuitive glimpse of the generality of the approach. The generality and flexibility of

OPGs allow for using much richer languages than the *Mini* example used in this paper. Moreover, attribute grammars—being Turing complete—enable formalizing in this framework any algorithmic schema at any sophistication and complexity level, posing no theoretical limitation to using SiDECAR. For example, more expressive language constructs and features (like procedure calls, procedures with reference parameters and side effects, pointers, shared-variable concurrency, non-determinism) could be accommodated with attribute schemas more complex both in terms of the attributes definition and in terms of the type (e.g., AGs with references [22] could be useful when the "semantics" of a program element is not confined within its "syntactic context"). More generally, richer attribute schemas could support both new language features and different verification algorithms (e.g., abstraction-based techniques for the case of verification of safety properties, or more realistic assumptions on the probabilistic system behavior for the case of verification of reliability requirements). In all these scenarios incrementality would be automatically provided by the framework, without any further effort for the developer.

We acknowledge that some technical issues should be faced when using SiDECAR in non-trivial practical cases. First, existing grammars could need to be transformed to satisfy requirement R1: the transformation (especially when automated) might reduce the readability of the grammar and could impact on the definition of attribute schemas. Expressing verification procedures as AGs (to satisfy requirement R2) could be a non-trivial task too: for instance, developers might simply be not familiar with the programming paradigm required by AGs; the reuse of known verification algorithms might be more or less straightforward and/or effective in the context of AG. We emphasize, however, that such a non-trivial effort is typically done once for all at design time, possibly in cooperation with domain experts. When the system is in operation, developers should only care about applying the changes and automatically (and incrementally) verifying their effects.

The generality of the methodology advocated by SiDECAR widens the scope of application to a number of scenarios. For example, at design time, SiDECAR (possibly integrated within IDE tools) can effectively support designers in evaluating the impact of changes in their products, in activities such as what-if analysis and regression verification, very common in agile development processes. Existing techniques for automated verification based either on model checking or on deductive approaches, as well as their optimizations, could be adapted to use SiDECAR, exploiting the benefits of incrementality. At run time, the incrementality provided by SiDECAR could be the key factor for efficient online verification of continuously changing situations, which could then trigger and drive the adaptation of self-adaptive systems [7]. As another instance of the approach's generality, similarly to the probabilistic verification described in section 4.1, other quantitative properties, such as execution time and energy consumption, could be verified with SiDECAR. Furthermore, SiDECAR could also bring at run time the same analysis techniques so far limited to design time because of efficiency reasons.

6 Related Work

In this section we present related work in two parts. First, we discuss work that addresses incrementality in verification[3] in general; next, we discuss other incremental approaches in the fields of the two examples presented here, namely probabilistic verification and safety program verification.

Different methodologies have been proposed in the literature as the basis for incremental verification techniques. They are mainly grounded in the assume-guarantee [26] 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 boundary 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 [10], which proposes a framework for performing assume-guarantee reasoning in an incremental and fully automatic fashion. Assume-guarantee based verification has been exploited also for probabilistic reasoning (e.g., in [31]), even though we are not aware of approaches using it in an incremental fashion.

Focusing now, more specifically, on incremental probabilistic verification, a known technique to achieve incremental verification is *parametric analysis* [13]. With this technique, the probability values 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 depending on the symbolic parameters. 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 reliability. Whenever there is a change of the values of the parameters, the results of the preprocessing phase can be reused, with significant improvements of the verification time [17]. The main limitation of this approach is that a structural change in the software invalidates the results of the preprocessing phase, requiring the verification to start from scratch, with consequent degradation of the analysis performance.

Parametric analysis is reminiscent of the notion of *partial evaluation*, originally introduced in [16]. 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.

Concerning related work on incremental safety verification, other approaches based on (regression) model checking reason in terms of the representation (e.g., a state-transition system) explored during the verification, by assessing how it is affected by changes in the program. The main idea is to maximize the reuse of the state space already explored for previous versions of the program, isolating the

[3]Incidentally, the use of the term *incremental model checking* in the context of bounded model checking [6] has a different meaning, since it refers to the possibility of changing the bound of the checking.

parts of the state space that have changed in the new version. The first work in this line of research addressed modal mu-calculus [35]. Henzinger et al. [23] analyze a new version of the program by checking for the conformance of its (abstract) state space representation with respect to the one of the previous version. When a discrepancy is found, the algorithm that recomputes the abstraction is restarted from that location. Depending on where the change is localized in the program text, the algorithm could invalidate—and thus recompute—a possibly large portion of the program state space. Similarly, incremental approaches for explicit-state model checking of object-oriented programs, such as [32] and [36], analyze the state space checked for a previous version and assess, respectively, either the transitions that do not need to be re-executed in a certain exploration of the state space, or the states that can be pruned, because not affected by the code change. These approaches tie incrementality to the low-level details of the verification procedure, while SiDECAR supports incrementality at a higher level, independently on the algorithm and data structures defined in the attributes. Conway et al. [11] define incremental algorithms for automaton-based safety program analyses. Their granularity for the identification of reusable parts of the state space is coarse-grained, since they take a function as the unit of change, while SiDECAR has a finer granularity, at the statement level. A combination of a modular verification technique that also reuse cached information from the checks of previous versions is presented in [29] for aspect-oriented software.

In conclusion, the syntactic-semantic approach embedded in SiDECAR does not constrain incrementality depending on on the modular structure of the artifacts, as instead required by assume-guarantee approaches. Furthermore, it provides a general and unifying methodology for defining verification procedures for functional and non-functional requirements.

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 verification activities by introducing SiDECAR, a framework for the definition of verification procedures, which are automatically enhanced with incrementality by the framework itself. SiDECAR supports a verification procedure encoded as synthesis of semantic attributes associated with a grammar. The attributes are evaluated by traversing the syntax tree that reflects the structure of the software system. By exploiting incremental parsing and attributes evaluation techniques, SiDECAR reduces the complexity of the verification procedure in presence of changes. We have shown SiDECAR in use to define two kinds of verification, namely probabilistic verification of reliability properties and safety verification of programs.

Future work will address several directions. We want to support run-time changes of the language (and thus the grammar) in which the artifact to be verified is described, motivated by advanced adaptiveness capability scenarios. We also want to support changes in the properties to be verified, and still exploit the benefit of incremental verification. We will continue our work to develop an incremental verification environment—by incorporating improvements to exploit parallelism [3] and to apply finer incremental parsing techniques—and will conduct experimental studies on real-world applications to quantify the effectiveness of SiDECAR in the definition and the execution of state-of-the-art verification procedures, identifying the kind of verification procedures for which the SiDECAR approach is more cost-effective. We will also investigate the pragmatic issues discussed in section 5, i.e., what can be reasonably encoded using OPGs and AGs. Finally, we plan to exploit SiDECAR to introduce verification-driven development in iterative and/or agile development processes.

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).

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

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

[3] A. Barenghi, E. Viviani, S. Crespi Reghizzi, D. Mandrioli, and M. Pradella. PAPAGENO: a parallel parser generator for operator precedence grammars. In *Proc. of SLE 2012*, volume 7745 of *LNCS*, pages 264–274. Springer, 2012.

[4] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issues and challenges. *IEEE Computer*, 39(10):36–43, 2006.

[5] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9:505–525, 2007.

[6] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[7] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self- adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, 2012.

[8] R. Cheung. A user-oriented software reliability model. *IEEE Trans. Soft. Eng.*, SE-6(2):118–125, March 1980.

[9] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999.

[10] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. of TACAS 2003*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.

[11] C. Conway, K. Namjoshi, D. Dams, and S. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *Proc. of CAV 2005*, volume 3576 of *LNCS*, pages 387–400. Springer, 2005.

[12] S. Crespi Reghizzi and D. Mandrioli. Operator precedence and the visibly pushdown property. *J. Comput. Syst. Sci.*, 78(6):1837–1867, 2012.

[13] C. Daws. Symbolic and parametric model checking of discrete-time markov chains. In *Proc. of ICTAC 2004*, volume 3407 of *LNCS*, pages 280–294. Springer, 2005.

[14] K. de Bosschere. An operator precedence parser for standard Prolog text. *Softw. Pract. Exper.*, 26(7):763–779, 1996.

[15] S. Distefano, A. Filieri, C. Ghezzi, and R. Mirandola. A compositional method for reliability analysis of workflows affected by multiple failure modes. In *Proc. of CBSE 2011*, pages 149–158. ACM, 2011.

[16] A. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977.

[17] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *Proc. of ICSE 2011*, pages 341–350. ACM, 2011.

[18] A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Asp. Comput.*, 24(2):163–186, March 2012.

[19] R. W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10:316–333, July 1963.

[20] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Trans. Program. Lang. Syst.*, 1(1):58–70, Jan. 1979.

[21] D. Grune and C. J. H. Jacobs. *Parsing Techniques - a practical guide.* Springer, 2nd edition, 2008.

[22] G. Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[23] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 180–181. Springer, 2004.

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

[25] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, October 2009.

[26] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, Oct 1983.

[27] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *Proc. of QEST 2002*, pages 243–244. IEEE Computer Society, 2005.

[28] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2:127–145, 1968.

[29] S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):Article 7, April 2007.

[30] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *Proc. of TOOLS 2002*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[31] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Assume-guarantee verification for probabilistic systems. In *Proc. of TACAS 2010*, volume 6015 of *LNCS*, pages 23–37. Springer, 2010.

[32] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *Proc. ICSE 2008*, pages 291–300. ACM, 2008.

[33] H. Pham. *System software reliability.* Springer, 2006.

[34] P. Sistla. Hybrid and incremental model-checking techniques. *ACM Comput. Surv.*, 28(4es), 1996.

[35] O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Proc. of CAV 1994*, volume 818 of *LNCS*, pages 351–363. Springer, 1994.

[36] G. Yang, M. Dwyer, and G. Rothermel. Regression model checking. In *Proc. of ICSM 2009*, pages 115–124. IEEE Computer Society, sept. 2009.