A Generalized Approach to Verification Condition Generation

Cláudio Belo Lourenço*, Maria João Frade*, Shin Nakajima†, Jorge Sousa Pinto*
* HASLab/INESC TEC & Universidade do Minho, Portugal
† National Institute of Informatics, Japan

***Abstract*—In a world where many human lives depend on the correct behavior of software systems, program verification assumes a crucial role. Many verification tools rely on an algorithm that generates *verification conditions* (VCs) from code annotated with properties to be checked. In this paper, we revisit two major methods that are widely used to produce VCs: *predicate transformers* (used mostly by deductive verification tools) and the *conditional normal form transformation* (used in bounded model checking of software). We identify three different aspects in which the methods differ (logical encoding of control flow, use of contexts, and semantics of *asserts*), and show that, since they are orthogonal, they can be freely combined. This results in six new hybrid *verification condition generators* (VCGens), which together with the fundamental methods constitute what we call the *VCGen cube*. We consider two optimizations implemented in major program verification tools and show that each of them can in fact be applied to an entire face of the cube, resulting in optimized versions of the six hybrid VCGens. Finally, we compare all VCGens empirically using a number of benchmarks. Although the results do not indicate absolute superiority of any given method, they do allow us to identify interesting patterns.**

I. Introduction

In spite of recent advances in the area of program verification, tools such as those belonging to the two related families of deductive verification [3], [13] and bounded model checking [7], [9], are still not mature enough for software engineers to use daily in their software development activities: they are either not precise enough or lack efficiency. Even though the bases of these tools diverge, they share a common feature: they internally employ a *verification condition generator* (VCGen). This is a component that takes, as input, a program with a specification, and outputs a number of proof obligations, called *verification conditions* (VCs). The VCs are then sent to a backend proof tool for validity checking.

The VCGens may, however, differ on the format of the VCs they generate and optimizations they employ. The VCGen and the optimization algorithms are usually coupled tightly, and both are hidden in the internals of tools. Identifying the baseline features that influence efficacy and efficiency of the verification techniques is not clear, and thus a thorough study of these aspects is desirable to understand the intricacies of each technique, to contribute towards improving current tools, and to allow for new techniques to be investigated systematically.

Some published works have already contributed towards a uniform formulation for verification condition generation. A first step taken by Gordon et al. [16] cover some ground on proving the correctness of Hoare triples based on forward computation of postconditions. Cruz et al. [11] present in a systematic way VCGen algorithms for code in *static single-assignment* (SSA) form. Lourenço et al. [6] cover some empirical ground and shows that the selection of a VCGen is not irrelevant in terms of efficiency, but also not easy to differentiate. Although these works demonstrate the importance of such an approach, they fall short of both proposing a uniform framework at the theoretical level and presenting a suitable empirical evaluation of different VCGens. Moreover, further study is needed to clarify the interplay between baseline methods and optimization techniques.

In this paper, we present a systematic study and categorization of VCGens and compare them empirically. The contributions of the paper are as follows: taking as point of departure the logical encoding of programs used in two well-known VCGens, (i) we identify three dimensions in which they differ and, combining these dimensions, (ii) we introduce 6 new hybrid VCGens and (iii) organize them graphically in what we call the *VCGen cube*; (iv) we propose a set of novel *optimized* hybrid VCGens, by considering how two flagship tools optimize the two basic encodings of the control-flow of programs, and showing that these optimizations can be applied to entire faces of the cube; finally, (v) we provide a comparative evaluation of all the different VCGens considered.

The paper is structured as follows: the next section discusses background material and related work; Section III reviews the two fundamental methods on which our study is based, and identifies the characterizing dimensions in which they differ. Section IV then shows how the dimensions may be combined to produce six new hybrid methods. The resulting VCGens are organized visually as a cube, and defined as a pair of generic VCGens. The cube is extended in Section V by incorporating optimization methods that are used in flagship tools, and the resulting VCGens are then compared. In Section VI we integrate all the previous definitions into a sigle unified VCGen, and discuss how the existing verification tools fit in our cube. Section VII describes how the VCGens have been implemented and evaluated empirically, and discusses the results of this evaluation. Section VIII concludes the paper.

II. Background and Related Work

We recall how two families of verification tools represent programs and VCs. One aspect in which *deductive verifiers* and *bounded model checkers of software* (BMCs) differ, is the

way in which they reduce iterating programs with subroutines (procedures, functions, or methods), into simple *branching programs* with no iteration or subroutines. In the former, the treatment of iteration relies on the notion of *loop invariant*, and subroutine are handled by means of the assume/guarantee principle using the notion of a *contract*. Invariants and contracts are typically provided by the user as *annotations* in the code, and allow for the programs to be linearized: loops and function calls can be replaced by sequences of *assume* and *assert* statements (see for instance [14]) such that the resulting branching program, while not operationally equivalent to the original, produces an adequate set of VCs (the transformation is sound). On the other hand, BMCs eliminate user intervention, relying instead on *loop expansion* and *code inlining* up to a given bound to handle iteration and (recursive) calls to subroutine – for this reason bounded model checkers are sometimes described as *bug finders* rather than program verification tools. Note that the two families can be seen as *complementing* each other; for instance, loop expansion may be used in conjunction with a deductive verification tool, to provide evidence, up to some bound, that a given invariant is valid for the loop, or otherwise to find a counter-example.

The principles underlying VC generation in deductive verification tools are based on *program logics* [19], [25] and on *predicate transformers* [12]. A good discussion of the choices regarding the encoding of data is found in [9]; dealing with complex data-structures requires a specific mechanism, such as a *memory model* or the use of a dedicated logic [25]. The idea of predicate transformers is that instead of being interpreted as state transformers (as in other semantic approaches), programs are interpreted as transformers of logical formulas characterizing states. For instance, the *strongest postcondition* (SP) interpretation of a program maps a formula ϕ (a *precondition*) into another formula θ that characterizes the final states of the program after (non-blocking and non-error producing) executions, starting from initial states satisfying ϕ. The definition of predicate transformers may in general produce VCs of exponential size in the length of the program, since it is based on a path enumeration that duplicates formulas whenever a conditional is reached, and the number of execution paths is exponential in the worst-case. The problem can be avoided if the programs are in a (dynamic) *single-assignment* (SA) form, in which, in each execution path of the program, each variable is assigned at most once. Flanagan and Saxe [14] show that predicate transformers can be calculated from passive form (similar to SA) in a way that generates quadratic-size VCs in the worst-case. Predicate transformers can also be used to generate VCs for unstructured programs [4], [17].

Clarke et al. [8] have proposed an alternative VC generation method with the CBMC bounded model checker. It relies on the conversion of code to an intermediate SA form that is then transformed to *conditional normal form* (CondNF), in which programs are sequences of single-branch conditionals containing atomic commands. A VC of worst-case quadratic size can be obtained from the CondNF.

In addition to the related works mentioned in the introduction [16], [11], [6], Godefroid et al. [15] report on the techniques used by some tools for creating logical encodings of programs. The authors briefly mention the complexity of each method but no comparisons between the logical encodings are offered, neither empirically nor theoretically. In the present paper, on the other hand, we present and compare multiple algorithms to generate VCs in a uniform way, abstracting away from specific details of the tools, and then show how the tools fit in our formulation.

Verification condition generation based on Hoare logic has also been formalized in the past using Higher-Order Logic, see for instance [20], [26], but not in the SA setting.

Finally, it should be mentioned that tools based on other families of software analysis techniques, such as model checkers based on existential abstraction and symbolic execution tools, often also integrate a VCGen as an auxiliary component. For instance, in the TRACER tool [21], a VCGen is used to determine when a given execution path subsumes another (the problem of exponential explosion is solved by resorting to an *interpolation* technique). Therefore we believe that our work in this paper may also, indirectly, be of use to developers of verification tools outside the deductive and bounded model checking families.

III. Fundamental VCGens

We will consider branching programs over a set of variables $x \in \mathbf{Var}$, a language of program expressions $e \in \mathbf{Exp}$, and Boolean expressions $b \in \mathbf{Exp^{bool}}$:

$$\mathbf{Comm} \ni C ::= \mathbf{skip} \mid x := e \mid C\,;\,C \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{assume}\,\theta \mid \mathbf{assert}\,\theta$$

In addition, we require program assertions $\theta \in \mathbf{Assert}$, obtained as a first-order expansion of $\mathbf{Exp^{bool}}$, to express properties about states. The language includes the *assume* and *assert* commands, as normally found in the guarded commands language initially introduced by Dijkstra [12]. Semantically, the command $\mathbf{assume}\,\theta$ is seen as *blocking* whenever executed in a state in which θ is false, and $\mathbf{assert}\,\theta$ is seen as producing an *error* when θ is false. Note that blocked executions are dispensed from obligations that would be imposed by subsequent *assert* statements. A program is *correct* if no error may occur in non blocking executions. For example, the fragment $\mathbf{assume}\,x \geq 0\,;\,x := x+10\,;\,\mathbf{assert}\,x > 10$ admits blocking executions (for initial values of $x < 0$), normal executions (for $x > 0$) and an erroneous execution (for $x = 0$), thus *it is not correct*. The program can be made correct by replacing the first assume condition by $x > 0$.

A naive approach for generating VCs may be based on *path enumeration*: for each asserted property, every incoming execution path will generate a different VC, thus the number of VCs is potentially exponential in the size of the program [6]. Straightforward VC generation by symbolic execution or predicate transformers suffers from this problem, but it should be noted that path enumeration has advantages from the point of

$$\begin{array}{l}\textbf{if } x_0{>}0 \textbf{ then } y_1 := 1 \textbf{ else } y_1 := 0;\\ \textbf{assert } y_1 = 0 \vee y_1 = 1\,;\\ \textbf{if } x_0{>}0 \textbf{ then } y_2 := 1 \textbf{ else } y_2 := 0;\\ \textbf{assert } y_2 = y_1\,;\\ \textbf{if } x_0{>}0 \textbf{ then } y_3 := 1 \textbf{ else } y_3 := 0;\\ \textbf{assert } y_3 = y_1\,;\end{array}$$

1. $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \rightarrow (y_1 = 0 \vee y_1 = 1)$
2. $((((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0))) \rightarrow y_2 = y_1$
3. $((((((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0))) \wedge y_2 = y_1) \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0))) \rightarrow y_3 = y_1$

Fig. 1: Example of an SA Program and its SP VCs

view of *traceability*: an invalid VC immediately identifies the executions that may violate a property. But for big programs, like those resulting from loop expansion in bounded model checking, path enumeration is impractical and thus other methods must be considered.

VCs Based on Predicate Transformers: Let us illustrate this form of VC generation using the strongest postcondition (SP) predicate transformer. A set of VCs can be calculated by following recursively the structure of programs, carrying as accumulator a propagated precondition ϕ as follows:

$$\begin{array}{rcl}\mathsf{VC}^{\mathsf{SP}}(\phi, \textbf{skip}) & = & \emptyset\\ \mathsf{VC}^{\mathsf{SP}}(\phi, x := e) & = & \emptyset\\ \mathsf{VC}^{\mathsf{SP}}(\phi, \textbf{assume}\,\theta) & = & \emptyset\\ \mathsf{VC}^{\mathsf{SP}}(\phi, C_1\,;\, C_2) & = & \mathsf{VC}^{\mathsf{SP}}(\phi, C_1) \cup\\ & & \mathsf{VC}^{\mathsf{SP}}(\mathsf{sp}\,(\phi, C_1), C_2)\\ \mathsf{VC}^{\mathsf{SP}}(\phi, \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2) & = & \mathsf{VC}^{\mathsf{SP}}(\phi \wedge b, C_1) \cup\\ & & \mathsf{VC}^{\mathsf{SP}}(\phi \wedge \neg b, C_2)\\ \mathsf{VC}^{\mathsf{SP}}(\phi, \textbf{assert}\,\theta) & = & \{\phi \rightarrow \theta\}\end{array}$$

Each *assert* statement originates a new VC, and the sp function is used in the sequence case to propagate ϕ through the first subprogram – sp returns the SP of the program C w.r.t. the condition ϕ as defined in Gordon et al. [16]. Following Flanagan and Saxe [14], if programs are in SA form, predicate transformers can be calculated by combining the propagated condition (without duplication) with a formula that depends only on the program. For instance, $\mathsf{sp}\,(\phi, C) \equiv \phi \wedge \mathcal{F}(C)$, where:

$$\begin{array}{rcl}\mathcal{F}(\textbf{skip}) & = & \top\\ \mathcal{F}(x := e) & = & x = e\\ \mathcal{F}(\textbf{assume}\,\theta) & = & \theta\\ \mathcal{F}(C_1\,;\, C_2) & = & \mathcal{F}(C_1) \wedge \mathcal{F}(C_2)\\ \mathcal{F}(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2) & = & (b \wedge \mathcal{F}(C_1)) \vee (\neg b \wedge \mathcal{F}(C_2))\\ \mathcal{F}(\textbf{assert}\,\theta) & = & \theta\end{array}$$

This is in fact a compact logical encoding of all the non-blocking, non-error producing executions of C. We will refer to this method as the 'predicate transformers VCGen', since the same principle can be applied to calculate verification conditions based on the weakest (liberal) precondition predicate transformer.

VCs Based on Conditional Normal Form: Conversion to CondNF involves a number of transformation that are sound if the code is in SA form [2]. In particular, it involves the sequentialization of the two branches of each conditional as

$$\begin{array}{l}\textbf{if } x_0 > 0 \textbf{ then } y_1 := 1\,;\\ \textbf{if } \neg x_0 > 0 \textbf{ then } y_1 := 0\,;\\ \textbf{if } \top \textbf{ then assert } y_1 = 0 \vee y_1 = 1\,;\\ \textbf{if } x_0 > 0 \textbf{ then } y_2 := 1\,;\\ \textbf{if } \neg x_0 > 0 \textbf{ then } y_2 := 0\,;\\ \textbf{if } \top \textbf{ then assert } y_2 = y_1\,;\\ \textbf{if } x_0 > 0 \textbf{ then } y_3 := 1\,;\\ \textbf{if } \neg x_0 > 0 \textbf{ then } y_3 := 0\,;\\ \textbf{if } \top \textbf{ then assert } y_3 = y_1\,;\end{array}$$

1. $((x_0 > 0 \rightarrow y_1 = 1) \wedge (\neg x_0 > 0 \rightarrow y_1 = 0) \wedge (x_0 > 0 \rightarrow y_2 = 1) \wedge (\neg x_0 > 0 \rightarrow y_2 = 0) \wedge (x_0 > 0 \rightarrow y_3 = 1) \wedge (\neg x_0 > 0 \rightarrow y_3 = 0)) \rightarrow ((\top \rightarrow y_1 = 0 \vee y_1 = 1) \wedge (\top \rightarrow y_2 = y_1) \wedge (\top \rightarrow y_3 = y_1))$

Fig. 2: Conversion to CondNF and resulting VC

done by the function below: the accumulator π is a *path condition* enabling the execution of the program.

$$\begin{array}{rcl}\mathsf{condNF}(\pi, \textbf{skip}) & = & \textbf{if } \pi \textbf{ then skip}\\ \mathsf{condNF}(\pi, x := e) & = & \textbf{if } \pi \textbf{ then } x := e\\ \mathsf{condNF}(\pi, \textbf{assume}\,\theta) & = & \textbf{if } \pi \textbf{ then assume}\,\theta\\ \mathsf{condNF}(\pi, C_1\,;\, C_2) & = & \mathsf{condNF}(\pi, C_1)\,;\\ & & \mathsf{condNF}(\pi, C_2)\\ \mathsf{condNF}(\pi, \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2) & = & \mathsf{condNF}(\pi \wedge b, C_1)\,;\\ & & \mathsf{condNF}(\pi \wedge \neg b, C_2)\\ \mathsf{condNF}(\pi, \textbf{assert}\,\theta) & = & \textbf{if } \pi \textbf{ then assert}\,\theta\end{array}$$

For a program C, the encoding $\mathcal{C}$ of the operational behavior, and the properties $\mathcal{P}$ to be verified can now be extracted using $(\mathcal{C}, \mathcal{P}) = \mathsf{split}(\mathsf{condNF}(\top, C))$, with split defined below (the treatment of *assume* is ommitted for now). The verification condition is written simply as $\bigwedge \mathcal{C} \rightarrow \bigwedge \mathcal{P}$.

$$\begin{array}{rcl}\mathsf{split}(\textbf{if } b \textbf{ then skip}) & = & (\emptyset, \emptyset)\\ \mathsf{split}(\textbf{if } b \textbf{ then } x := e) & = & (\{b \rightarrow x = e\}, \emptyset)\\ \mathsf{split}(C_1\,;\, C_2) & = & (\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{P}_1 \cup \mathcal{P}_2), \text{ where}\\ & & (\mathcal{C}_1, \mathcal{P}_1) = \mathsf{split}(C_1), \text{ and}\\ & & (\mathcal{C}_2, \mathcal{P}_2) = \mathsf{split}(C_2)\\ \mathsf{split}(\textbf{if } b \textbf{ then assert}\,\theta) & = & (\emptyset, \{b \rightarrow \theta\})\end{array}$$

Example: Fig. 1 shows a simple program consisting of a sequence of three conditionals, each followed by an *assert*. On the bottom it shows the VCs obtained with the predicate transformers VCGen. This is in fact a worst-case example: the number of VCs is linear in the program size, and the size of each VC is linear in the length of the program prefix leading to it, which results in a set of VCs of *quadratic size*. Fig. 2 shows the same program converted to CondNF and the resulting VC. Both sets $\mathcal{C}$ and $\mathcal{P}$ have linear size, and so does the resulting VC, since the encoding is not partially duplicated for each assert formula. However, the CondNF method *also generates quadratic VCs in the worst case*, which occurs for programs containing a chain of nested conditionals. In this case each condition will be duplicated through all the internal branches, generating the quadratic pattern.

VCGen Dimensions: The example motivates the discussion of three dimensions in which the two methods differ.

- The first is the **logical encoding**: with predicate transformers, programs are encoded following their branching structure, with conditionals encoded as disjunctions of the form $(b \wedge \ldots) \vee (\neg b \wedge \ldots)$, whereas the CondNF method

encodes programs as conjunctions of implicative formulas, with path conditions guarding atomic statements.

- The second dimension is the use of **contexts** for each VC. The predicate transformers method produces one VC for each *assert*; each VC is an implicative formula whose antecedent is a *partial context* that encodes only the part of the program that is relevant for that *assert*. The CondNF method produces a *global context* that encodes the behavior of the entire program.
- Finally, the VCGens differ in the **semantics of the assert** command. The traditional interpretation of guarded commands implies that an execution stops whenever $\mathbf{assert}\,\theta$ produces an error [12]; in Fig. 1, assert conditions are introduced in the context to be used in subsequent VCs, which is consistent with this interpretation. The CondNF method treats *asserts* as commands that check the value of assertions, but do not fail; when an *assert* is reached, it is not known whether previous asserts have been passed successfully or not. In Fig. 2 the global context $\mathcal{C}$ does not contain assert conditions.

Let us now compare the options with respect to relevant aspects for both the efficacy and efficiency of the verification.

Traceability. With a global context, if a VC is invalid then the only way to locate the error is to *interpret the counterexample*, which will give us a concrete execution that violates a particular property. This may not be required with a partial context encoding, since there is a clear association between invalid VCs and violated properties.

Economy of Contexts. From the strict viewpoint of checking each *assert*, partial contexts are preferable to a global context, which may contain unnecessary information. Consider a big program with a single *assert* placed at the beginning: with partial contexts a small VC will be generated, but with a global context the encoding of the entire program will be part of the VC.

Redundancy. When verifying a whole program, the use of partial contexts has the disadvantage of replicating operational and axiomatic information in the different VCs, increasing the overall size.

Lemmas. Including assertions in the context, as in the predicate transformers VCGen, implies that intermediate assertions play the role of *lemmas*: once they are proved they can be used to prove subsequent assertions.

In summary, partial contexts seem to be preferable regarding traceability and the verification of each individual assert, whereas global contexts reduce the overall redundancy of generated VCs.

IV. A Cube of VCGens

The three VCGen dimensions identified in the previous section are orthogonal, and may be freely combined to produce hybrid VCGens for SA programs (ranging over $\mathbf{Comm}^{\mathrm{SA}}$). Let us now consider these combinations.

Some are more obvious than others. It is straightforward to modify the predicate transformers VCGen to exclude assertions from the context: it suffices to modify the definition of

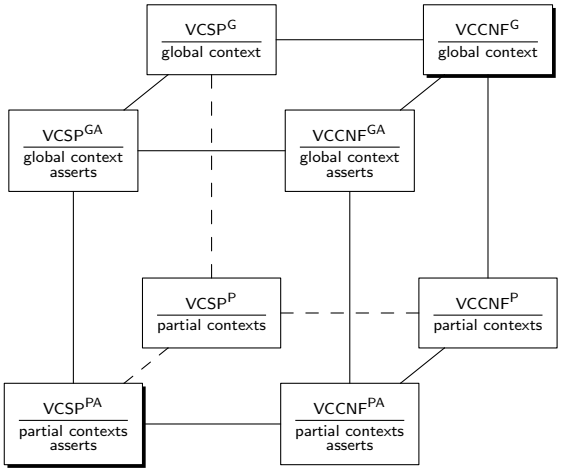


Fig. 3: A Cube of VCGens: **Left/right**: SP/CondNF encoding; **Top/bottom**: G/P contexts; **Front/back**: asserts included/not included in the context.

the program formula so that $\mathcal{F}(\mathbf{assert}\,\theta) = \top$. The logical encoding can be changed to CondNF in a straightforward way, while keeping the partial contexts that are typical of the predicate transformers VCGen. For instance, the first VC of Fig. 1 would become $((x_0 > 0 \to y_1 = 1) \wedge (\neg x_0 > 0 \to y_1 = 0)) \to (\top \to y_1 = 0 \vee y_1 = 1)$. It seems less trivial to include assertions in the context in the VCGen based on CondNF, while keeping a single global context. This can be achieved by combining the operational context (encoding the behavior of the entire program) with *axiomatic* partial contexts encoding the assertion information that is relevant for each *assert*. The VC of Fig. 2 would become $((x_0 > 0 \to y_1 = 1) \wedge (\neg x_0 > 0 \to y_1 = 0) \wedge (x_0 > 0 \to y_2 = 1) \wedge (\neg x_0 > 0 \to y_2 = 0) \wedge (x_0 > 0 \to y_3 = 1) \wedge (\neg x_0 > 0 \to y_3 = 0)) \to (\top \to y_1 = 0 \vee y_1 = 1) \wedge ((\top \to y_1 = 0 \vee y_1 = 1) \to (\top \to y_2 = y_1)) \wedge (((\top \to y_1 = 0 \vee y_1 = 1) \wedge (\top \to y_2 = y_1)) \to (\top \to y_3 = y_1))$. Note that the use of these axiomatic partial contexts reintroduces redundancy (in the sense discussed in the previous section) in the encoding of axiomatic information, but not of the operational behavior. Finally, using predicate transformers to create a VC with a single context would produce: $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \to (y_1 = 0 \vee y_1 = 1) \wedge y_2 = y_1 \wedge y_3 = y_1$.

We propose in Fig. 3 a graphical organization of the eight resulting VCGens as a *cube*. The VCGens will be defined as two generic algorithms, one based on the predicate transformers logical encoding, and the other based on the CondNF encoding. Each generic algorithm corresponds either to the left or right face of the cube, and admits four variants corresponding to each of its vertices.

The generic predicate transformers VCGen, defined in Fig. 4 (top), comprises the four concrete versions $\mathsf{VCSP}^{\mathsf{P}}$, $\mathsf{VCSP}^{\mathsf{PA}}$, $\mathsf{VCSP}^{\mathsf{G}}$, and $\mathsf{VCSP}^{\mathsf{GA}}$ that differ only in the case of the assert command. In order to standardize the VCGen definition while allowing for the use of either a partial or global context,

$$\mathsf{VCSP}^i : \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Comm}^{\mathrm{SA}} \to \mathbf{Assert} \times \mathbf{Assert} \times \mathcal{P}(\mathbf{Assert})$$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{skip}) = (\top, \top, \emptyset)$$

$$\mathsf{VCSP}^i(\phi, \rho, x := e) = (x = e, \top, \emptyset)$$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{assert}\,\theta) = \begin{cases} (\top, \top, \{\phi \wedge \rho \to \theta\}) & \text{if } i = \mathsf{P} \\ (\top, \theta, \{\phi \wedge \rho \to \theta\}) & \text{if } i = \mathsf{PA} \\ (\top, \top, \{\rho \to \theta\}) & \text{if } i = \mathsf{G} \\ (\top, \theta, \{\rho \to \theta\}) & \text{if } i = \mathsf{GA} \end{cases}$$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{assume}\,\theta) = (\top, \theta, \emptyset)$$

$$\mathsf{VCSP}^i(\phi, \rho, C_1\,;\, C_2) = (\psi_1 \wedge \psi_2,\ \gamma_1 \wedge \gamma_2,\ \Gamma_1 \cup \Gamma_2)$$
where $(\psi_1, \gamma_1, \Gamma_1) = \mathsf{VCSP}^i(\phi, C_1)$ and $(\psi_2, \gamma_2, \Gamma_2) = \mathsf{VCSP}^i(\phi \wedge \psi_1, \rho \wedge \gamma_1, C_2)$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2) = ((b \wedge \psi_1) \vee (\neg b \wedge \psi_2),\ (b \wedge \gamma_1) \vee (\neg b \wedge \gamma_2),\ \Gamma_1 \cup \Gamma_2)$$
where $(\psi_1, \gamma_1, \Gamma_1) = \mathsf{VCSP}^i(\phi \wedge b, \rho \wedge b, C_1)$ and $(\psi_2, \gamma_2, \Gamma_2) = \mathsf{VCSP}^i(\phi \wedge \neg b, \rho \wedge \neg b, C_2)$

$$\mathsf{VCCNF}^i : \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Comm}^{\mathrm{SA}} \to \mathbf{Assert} \times \mathbf{Assert} \times \mathcal{P}(\mathbf{Assert})$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{skip}) = (\top, \top, \emptyset)$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, x := e) = (\pi \to x = e, \top, \emptyset)$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{assert}\,\theta) = \begin{cases} (\top, \top, \{\phi \wedge \rho \to \pi \to \theta\}) & \text{if } i = \mathsf{P} \\ (\top, \pi \to \theta, \{\phi \wedge \rho \to \pi \to \theta\}) & \text{if } i = \mathsf{PA} \\ (\top, \top, \{\rho \to \pi \to \theta\}) & \text{if } i = \mathsf{G} \\ (\top, \pi \to \theta, \{\rho \to \pi \to \theta\}) & \text{if } i = \mathsf{GA} \end{cases}$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{assume}\,\theta) = (\top, \pi \to \theta, \emptyset)$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, C_1\,;\, C_2) = (\psi_1 \wedge \psi_2,\ \gamma_1 \wedge \gamma_2,\ \Gamma_1 \cup \Gamma_2)$$
where $(\psi_1, \gamma_1, \Gamma_1) = \mathsf{VCCNF}^i(\pi, \phi, \rho, C_1)$ and $(\psi_2, \gamma_2, \Gamma_2) = \mathsf{VCCNF}^i(\pi, \phi \wedge \psi_1, \rho \wedge \gamma_1, C_2)$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2) = (\psi_1 \wedge \psi_2,\ \gamma_1 \wedge \gamma_2,\ \Gamma_1 \cup \Gamma_2)$$
where $(\psi_1, \gamma_1, \Gamma_1) = \mathsf{VCCNF}^i(\pi \wedge b, \phi, \rho, C_1)$ and $(\psi_2, \gamma_2, \Gamma_2) = \mathsf{VCCNF}^i(\pi \wedge \neg b, \phi, \rho, C_2)$

Fig. 4: Generic VCGens, for $i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$, based on predicate transformers (top) and on CondNF (bottom)

incoming executions are encoded by a formula ϕ capturing the operational aspects of executions (state changing), and a formula ρ capturing the axiomatic aspects of executions (*assume* and *assert* statements). In practice this separation is not mandatory for the partial context variants. The result of $\mathsf{VCSP}^i(\phi, \rho, C)$ is a tuple (ψ, γ, Γ) where Γ is a set of VCs, and ψ, γ encode respectively the operational and axiomatic execution information. The verification conditions of C are obtained as follows. Let $(\psi, \gamma, \Gamma) = \mathsf{VCSP}^i(\top, \top, C)$. If $i \in \{\mathsf{P}, \mathsf{PA}\}$, the partial contexts VC are given by Γ. Otherwise, if $i \in \{\mathsf{G}, \mathsf{GA}\}$, the global context VC is $\psi \to \bigwedge \Gamma$. We will refer to this set of VCGens as VCSP.

The generic VCGen based on CondNF, defined in Fig. 4 (bottom), differs from the previous one in that the encoding of both the operational and axiomatic parts of programs is based on the conditional normal form method. This requires separating path conditions from the accumulator corresponding to the axiomatic component of executions. Thus in the call $\mathsf{VCCNF}^i(\pi, \phi, \rho, C)$ the formula π is the path condition enabling the execution of C, and ϕ and ρ encode as before the operational contents and information relative to *assume* and *assert* statements of incoming executions. Again the choice of a concrete VCGen depends on the assert clause used. Let $(\psi, \gamma, \Gamma) = \mathsf{VCCNF}^i(\top, \top, \top, C)$. If $i \in \{\mathsf{P}, \mathsf{PA}\}$, the partial contexts VC are given by Γ. Otherwise, if $i \in \{\mathsf{G}, \mathsf{GA}\}$, the global context VC is $\psi \to \bigwedge \Gamma$. We will refer to this set of VCGens as VCCNF.

V. Optimizations and Comparison of VCGens

The foremost deductive verification and bounded model checking tools are practically usable because of their advanced optimization techniques. These techniques are usually tied to specific VCGen algorithms, and thus it is not clear whether they can be applicable to others. This section clears that gap by presenting how two important practical optimization techniques [4], [7] are applicable to appropriate VCGens. The graphical notion of the VCGen cube helps clarifying the intricacies between the optimization technique and VCGens.

Guided by the work of Barnett et al. [4] and the VCGen implemented in Boogie [3], it is possible to modify Flanagan and Saxe's predicate transformers VCGen for producing a single VC that reduces redundancy. The idea is fairly simple: even though the size of each VC is linear, the overall quadratic size comes from the possible existence of a linear number

$$\mathsf{VCLean}^i : \mathbf{Comm}^{\mathrm{SA}} \to \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert}$$

$$\mathsf{VCLean}^i(\mathbf{skip}) = (\top, \top, \top)$$

$$\mathsf{VCLean}^i(x := e) = (x = e, \top, \top)$$

$$\mathsf{VCLean}^i(\mathbf{assert}\,\theta) = \begin{cases} (\top, \top, \theta) & \text{if } i \in \{\mathsf{P}, \mathsf{G}\} \\ (\top, \theta, \theta) & \text{if } i \in \{\mathsf{PA}, \mathsf{GA}\} \end{cases}$$

$$\mathsf{VCLean}^i(\mathbf{assume}\,\theta) = (\top, \theta, \top)$$

$$\mathsf{VCLean}^i(C_1\,;\,C_2) = \begin{cases} (\psi_1 \wedge \psi_2,\ \gamma_1 \wedge \gamma_2, \\ \quad \delta_1 \wedge (\psi_1 \wedge \gamma_1 \to \delta_2)) & \text{if } i \in \{\mathsf{P}, \mathsf{PA}\} \\ (\psi_1 \wedge \psi_2,\ \gamma_1 \wedge \gamma_2, \\ \quad \delta_1 \wedge (\gamma_1 \to \delta_2)) & \text{if } i \in \{\mathsf{G}, \mathsf{GA}\} \end{cases}$$

where $(\psi_1, \gamma_1, \delta_1) = \mathsf{VCLean}^i(C_1)$
and $(\psi_2, \gamma_2, \delta_2) = \mathsf{VCLean}^i(C_2)$

$$\mathsf{VCLean}^i(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2) = ((b \wedge \psi_1) \vee (\neg b \wedge \psi_2),\ (b \wedge \gamma_t) \vee (\neg b \wedge \gamma_2), (b \to \delta_1) \wedge (\neg b \to \delta_2))$$

where $(\psi_1, \gamma_1, \delta_1) = \mathsf{VCLean}^i(C_1)$,
and $(\psi_2, \gamma_2, \delta_2) = \mathsf{VCLean}^i(C_2)$

Fig. 5: Lean optimization VCGen, for $i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$

of such VCs, each replicating partially the encoding of the program. This set of VCs can be transformed into a single VC, applying the equivalence $(\phi \to \theta) \wedge (\phi \wedge \theta \wedge \gamma \to \psi) \equiv \phi \to \theta \wedge (\theta \wedge \gamma \to \psi)$ the required number of times (with θ and ψ assert conditions). Consider again the three VCs of Fig. 1; applying this equivalence twice we obtain the following VC, where asserts (the 'goals' of the VC) are in bold: $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \to \boldsymbol{(y_1 = 0 \vee y_1 = 1)} \wedge ((y_1 = 0 \vee y_1 = 1) \to ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \to \boldsymbol{y_2 = y_1} \wedge (y_2 = y_1 \to ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \to \boldsymbol{y_3 = y_1}))$. This optimization can actually be applied to the entire left-hand face of the VCGens cube. If asserts are not to be introduced in the context we simply note that $\phi \to \theta \wedge (\theta \wedge \gamma \to \psi) \equiv \phi \to \theta \wedge (\gamma \to \psi)$, resulting in $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \to \boldsymbol{(y_1 = 0 \vee y_1 = 1)} \wedge (((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \to \boldsymbol{y_2 = y_1} \wedge (((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \to \boldsymbol{y_3 = y_1}))$. This also applies to cases with a global context: $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \to \boldsymbol{(y_1 = 0 \vee y_1 = 1)} \wedge (y_1 = 0 \vee y_1 = 1 \to (\boldsymbol{y_2 = y_1} \wedge (y_2 = y_1 \to \boldsymbol{y_3 = y_1})))$. The lean optimization defined through the function VCLean^i is defined in Fig. 5. Note that, it does not use the accumulator parameters and the result of $\mathsf{VCLean}^i(C)$ is a tuple (ψ, γ, δ) where the first two components are the same as with VCSP^i, and δ is a single formula. Again, if $i \in \{\mathsf{P}, \mathsf{PA}\}$, the partial context VC is δ, otherwise, if $i \in \{\mathsf{G}, \mathsf{GA}\}$, the global context VC is $\psi \to \delta$.

SSA Optimization: An entirely different optimization, integrated in the CBMC tool [7], can be applied to the right-hand side face of the cube. This optimization requires converting programs to a *static* single-assignment (SSA) form, where variables may occur in the code (syntactically) at most once as an l-value. Different versions of the same variables assigned in different branches need to be merged using a Φ-function [10]. Following [7], we will use here *conditional expressions* as found in the C programming language. For instance, the program $\mathbf{if}\ b\ \mathbf{then}\ x := e_1\ \mathbf{else}\ x := e_2$ can be translated to the following SSA form: $\mathbf{if}\ b\ \mathbf{then}\ x_1 := e_1\ \mathbf{else}\ x_2 := e_2\,;\, x_3 := b\,?\,x_1 : x_2$. These conditional expressions can then be encoded logically as conjunctions of two implicative formulas with mutually exclusive conditions. For instance, $x_3 := b\,?\,x_1 : x_2$ can be encoded as $(b \to x_3 = x_1) \wedge (\neg b \to x_3 = x_2)$. The basis of the optimization is the observation that path conditions introduced in the CondNF for assignment statements are not required. Without the optimization the above example would be encoded as the formula $(b \to x_1 = e_1) \wedge (\neg b \to x_2 = e_2) \wedge (b \to x_3 = x_1) \wedge (\neg b \to x_3 = x_2)$, whereas with the optimization this becomes $x_1 = e_1 \wedge x_2 = e_2 \wedge (b \to x_3 = x_1) \wedge (\neg b \to x_3 = x_2)$. Note that the left hand side of the implications come from the conditional expression and not from the path condition. The generic VCGen based on this optimization will be denoted by VCSSA^i and differs from VCCNF^i in the assignment case, which becomes: $\mathsf{VCSSA}^i(\pi, \phi, \rho, x := e) = (x = e, \top, \emptyset)$. Note that path conditions are still required for *assert* and *assume* statements.

Comparison of VCGens: As a proof of concept, to show that studying different methods for the generation of VCs is useful and can lead to improvements in the state of the art verification tools, we give here an analytical comparison of the behavior of the different VCGens for a specific program.

Our case study is representative of programs with a dense presence of *assert* statements. This will allow us to investigate whether it will be advantageous (as in theory it seems to be) to use a global context VCGen for verifying programs with a substantial number of asserts. We will consider a program of size N containing a number of asserts that is linear in N, such that execution may go through all the asserts. An easy way to generate such a program is by expanding iterations of a loop containing asserts, as the one shown in Fig. 6a. The result of expanding the loop twice is in Fig. 6b and its CondNF representation in Fig. 6c.

The table below shows the asymptotic analysis of the size of the VCs generated from this program.

	VCCNF	VCSSA	VCSP	VCLean
G	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$	$\theta(N)$
GA	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^2)$	$\theta(N)$
P	$\theta(N^3)$	$\theta(N^2)$	$\theta(N^2)$	$\theta(N)$
PA	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^2)$	$\theta(N)$

For VCCNF, since there are N nested conditionals due to loop expansion, the loop condition is replicated $\theta(N^2)$ times with different variables. The size of the global context VC will be in $\theta(N^2)$, whereas the size of the partial contexts VCs will be in $\theta(1^2), \theta(2^2), \ldots, \theta(N^2)$, resulting in an overall size of $\theta(N^3)$ – $2N$ VCs will be generated because there are two asserts in the loop. If asserts are included in the context, the

```
assume x ≥ 0 ∧ x ≤ 50;
assume y < x;
while x < 100 do {
  assert y < 100;
  x := x + 1;
  y := y + 1;
  assert y ≤ 100}
```

(a) Initial program

```
assume x ≥ 0 ∧ x ≤ 50;
assume y < x;
if x < 100 then {
  assert y < 100;
  x := x + 1;
  y := y + 1;
  assert y ≤ 100;
  if x < 100 then {
    assert y < 100;
    x := x + 1;
    y := y + 1;
    assert y ≤ 100;
    if x < 100 then
      assume ⊥
}}
```

(b) Expanding loop

```
if ⊤ then assume x0 ≥ 0 ∧ x0 ≤ 50;
if ⊤ then assume y0 < x0;
if x0 < 100 then assert y0 < 100;
if x0 < 100 then x1 := x0 + 1;
if x0 < 100 then y1 := y0 + 1;
if x0 < 100 then assert y1 ≤ 100;
if x0 < 100 ∧ x1 < 100 then assert y1 < 100;
if x0 < 100 ∧ x1 < 100 then x2 := x1 + 1;
if x0 < 100 ∧ x1 < 100 then y2 := y1 + 1;
if x0 < 100 ∧ x1 < 100 then assert y2 ≤ 100;
if x0 < 100 ∧ x1 < 100 ∧ x2 < 100 then assume ⊥;
if x0 < 100 then x3 := (x1 < 100) ? x2 : x1;
if x0 < 100 then y3 := (x1 < 100) ? y2 : y1;
if ⊤ then x4 := (x0 < 100) ? x3 : x0;
if ⊤ then y4 := (x0 < 100) ? y3 : y0;
```

(c) After converting to CondNF

Fig. 6: Bounded model checking a program with asserts by unfolding loops twice

global context VC also becomes of size $\theta(N^3)$, because each lemma will be guarded by a formula of size $\theta(1^2), \ldots, \theta(N^2)$.

Let us consider now VCSSA with the program shown in Fig. 6c after removing the gray code. The left-hand side of the global context VC (operational encoding) becomes of size $\theta(N)$, since the optimization dispenses entirely the accumulated path conditions. However, note that path conditions can only be removed from assignments, and so the consequent of the VC is still of size $\theta(N^2)$ (resp. $\theta(N^3)$, with asserts in the context), since it contains $2N$ assert conditions, each guarded by a path condition of size in $\theta(N)$ (resp. $\theta(N^2)$). It is in the partial contexts version (with no asserts) that the optimization becomes more interesting, since it will now generate $2N$ VCs, of size $\theta(1)$ to $\theta(N)$ (each having a single assert condition as consequent), with overall size in $\theta(N^2)$.

Let us turn to predicate transformers. Our example leads to the $\theta(N^2)$ worst-case behavior of VCSP in terms of VC size. In the partial contexts version there will be $2N$ VCs of size in $\theta(1)$, $\ldots$, $\theta(N)$; for the global context version, the operational encoding will have size in $\theta(N)$, but the size of the VC consequent will be in $\theta(N^2)$ – each assert will be guarded by conditions originated by the loop expansion. With

$$
\begin{aligned}
&\mathsf{VCG} : \mathbf{Comm}^{\mathsf{SA}} \times \{\mathsf{SP},\mathsf{CNF},\mathsf{LEAN},\mathsf{SSA}\} \times \{\mathsf{P},\mathsf{PA},\mathsf{G},\mathsf{GA}\} \\
&\qquad\qquad \to \mathcal{P}(\mathbf{Assert}) \\
&\mathsf{VCG}(C,\mathsf{SP},i) \quad = \mathsf{let}\ (\psi,\_,\Gamma) = \mathsf{VCSP}^i(\top,\top,C) \\
&\qquad\qquad \mathsf{in\ if}\ i \in \{\mathsf{P},\mathsf{PA}\}\ \mathsf{then}\ \Gamma\ \mathsf{else}\ \{\psi \to \textstyle\bigwedge \Gamma\} \\
&\mathsf{VCG}(C,\mathsf{CNF},i) \quad = \mathsf{let}\ (\psi,\_,\Gamma) = \mathsf{VCCNF}^i(\top,\top,\top,C) \\
&\qquad\qquad \mathsf{in\ if}\ i \in \{\mathsf{P},\mathsf{PA}\}\ \mathsf{then}\ \Gamma\ \mathsf{else}\ \{\psi \to \textstyle\bigwedge \Gamma\} \\
&\mathsf{VCG}(C,\mathsf{LEAN},i) = \mathsf{let}\ (\psi,\_,\delta) = \mathsf{VCLean}^i(C) \\
&\qquad\qquad \mathsf{in\ if}\ i \in \{\mathsf{P},\mathsf{PA}\}\ \mathsf{then}\ \{\delta\}\ \mathsf{else}\ \{\psi \to \delta\} \\
&\mathsf{VCG}(C,\mathsf{SSA},i) \quad = \mathsf{let}\ (\psi,\_,\Gamma) = \mathsf{VCSSA}^i(\top,\top,\top,C) \\
&\qquad\qquad \mathsf{in\ if}\ i \in \{\mathsf{P},\mathsf{PA}\}\ \mathsf{then}\ \Gamma\ \mathsf{else}\ \{\psi \to \textstyle\bigwedge \Gamma\}
\end{aligned}
$$

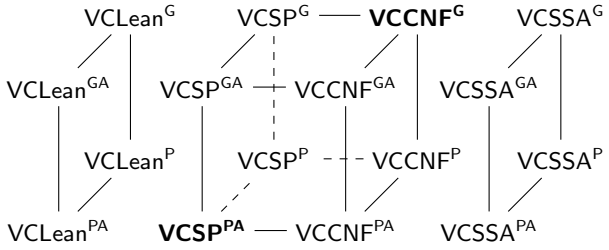
Fig. 7: Unified VCGen

VCLean, the VC size becomes linear in all variants.

It should be noted that in the worst case VCLean generates VCs of size $\theta(N^2)$. As an example consider the program $(\mathbf{assert}\,\phi_1\,;\,\mathbf{assert}\,\phi_2)\,;\,\mathbf{assert}\,\phi_3$ (note that the sequence of commands are associated to the left). The generated VC using $\mathsf{VCLean}^{\mathsf{PA}}$ is $\phi_1 \wedge (\phi_1 \to \phi_2) \wedge (\phi_1 \wedge \phi_2 \to \phi_3)$, which duplicates the asserted conditions. Even though in this example the duplication of assertions can be avoided by associating the sequences to the right (without interfering with the program semantics), the same growth of the VC size can be obtained with nested if statements.

VI. A unified VC Generator

It is clear from the above that each of the two original optimizations generates four new hybrid VCGens that can be added to our cube as follows:



All the VCGens can now be integrated into a single unified generator as shown in Fig. 7.

Let us now consider how the existing verification tools fit in our cube. First of all, it should be said that no tool, as far as we know, uses global contexts: all existing tools are located in the bottom face of the cube. Although the pure CondNF, using a global context, was originally introduced for CBMC [8], the current version of the tool seems to use the CondNF-encoding with partial contexts together with the SSA-based optimization. Therefore, CBMC is likely based on $\mathsf{VCSSA}^{\mathsf{P}}$.

The Boogie [3] and Why3 [13] deductive verifiers are located on the left-hand face. Both use partial contexts that include *assert* conditions. Boogie incorporates a linear optimization similar to the lean optimization, but it avoids du-

plication of assertions by translating structured programs into unstructured ones and resorting to SMT-LIB let binders [4]. While Boogie supports the use of loop invariants, typical of deductive verification, it also supports *loop unrolling* as found in bounded model checking tools, which has the potential to generate programs with a large cyclomatic complexity. This is probably additional motivation for the use of the linear-size VCGen. Why3 is entirely based on deductive reasoning with invariants and contracts, does not support loop unrolling, and has traceability as an important feature. The method used by default in Why3 is based on a potentially exponential path enumeration, which has advantages from the point of view of traceability: the Why3 graphical interface is able to highlight execution paths corresponding to selected VCs. Nevertheless, Why3 also implements and makes available (through a command-line switch) the $\mathsf{VCLean}^{\mathsf{PA}}$ algorithm, and a splitting operation can then be used explicitly to separate the VC into a set of 'single-goal' VCs.

VII. Experimental Results

The cube sets the basis for a thorough comparative evaluation of the VCGens w.r.t. different criteria, using a representative set of benchmark programs.

Setting and Toolchain: Since no existing tool implements all the algorithms, we developed a tool on top of SNIPER [22] to analyze the effect of solving VCs generated by different VCGens. This tool, baptized SNIPER-VCGen (available from http://alfa.di.uminho.pt/~belolourenco/sniper-vcgen.html), targets the verification of iteration-free LLVM intermediate representation [23]. The use of LLVM as intermediate language is convenient for our purposes, since loop expansion and optimizations involving constant propagation and simplification are readily implemented by the LLVM toolset prior to VC generation. An empirical comparison of VCGens requires the generation of VCs of substantial size, which we obtain by expanding loops. The resulting formulas are encoded in the SMT-LIB v2 language [5], and are then sent to different solvers for checking in the QF_AUFLIA logic, which supports quantifier-free formulas, (unbounded) integer arithmetic, and integer arrays. In our experiments we used the Z3 (v. 4.4.1, http://github.com/Z3Prover/z3), CVC4 (v. 1.4, http://cvc4.cs.stanford.edu), and MathSAT (v. 5.3.10, http://mathsat.fbk.eu) SMT solvers, to evaluate whether our VCGen comparison results hold consistently across a diverse set of solvers, or whether they are solver-dependent.

Evaluation of the Running Example: For the first part of the evaluation, we take the program from Fig. 6, translate it to LLVM, unwind loops N times, and generate a set of VCs using one of the VCGens mentioned before. The size of the corresponding SMT problem and the solving time are then measured experimentally. The detailed results for $N = 100, 200, 300$ are in the tool's webpage.

The file size data supports the above asymptotic reasoning. With respect to the VCGens based on CondNF, the SSA optimization produces a dramatic decrease on the file size when partial contexts are used, in particular if no asserts are included. For VCSP, the file size for both global and partial contexts without asserts are similar to those obtained for SSA, but now the inclusion of assertions as lemmas in the context has only marginal impact on the file size. With the lean optimization, VC size becomes linear in all cases; the data confirms that this is by far the most efficient of all the evaluated VCGens regarding file size.

As to the solver execution time, $\mathsf{VCCNF}^{\mathsf{G}}$ performs much better than the remaining VCGens based on CondNF (more than 10 times faster than the partial contexts $\mathsf{VCCNF}^{\mathsf{P}}$). The SSA optimization improves solving time only marginally with global contexts, but with partial contexts reduces it to roughly 1/3 (or to 1/2 if asserts are included in contexts). Nonetheless, $\mathsf{VCSSA}^{\mathsf{G}}$ beats $\mathsf{VCSSA}^{\mathsf{P}}$ (resulting in roughly 5 times faster solving). With the SP VCGens again the use of a global context results in several times faster solving than partial contexts: $\mathsf{VCSP}^{\mathsf{G}}$ stands roughly between $\mathsf{VCCNF}^{\mathsf{G}}$ and $\mathsf{VCSSA}^{\mathsf{G}}$, and $\mathsf{VCSP}^{\mathsf{P}}$ has similar performance to $\mathsf{VCSSA}^{\mathsf{P}}$. Finally, and as expected from the analysis of VC size, VCLean lead to significantly more efficient solving than any other VCGen. Although in all other cases it is preferable to use a global context, VCLean performs slightly better with partial contexts.

Although in theory it seems that the example program will expose the benefits of using asserts as lemmas, this kind of reasoning is misleading. In practice, when an automated solver is used, there is no way to influence the proof, and asserts are best left out of contexts, since they result in worse performance (with the exception of VCLean, for which adding asserts does not affect performance).

Benchmarks: In addition to the example program of Fig. 6, we evaluated the VCGens using a suite consisting of several case studies from the Eureka and InvGen benchmarks [1], [18], that have been used before to test, validate and evaluate other tools. These programs are algorithmically more complicated, and therefore allow us to compare the VCGens in a more realistic setting. In particular, the Eureka benchmark was created with the aim of assessing the scalability of software model checking tools, with programs of increasing complexity. Since assert statements are scarce in Eureka programs we also use case studies from the InvGen benchmarks, which are rich in asserts and allow us to evaluate the effect of including them in the context (one of the dimensions of the VCGen cube). The Eureka benchmark contains both correct and faulty annotated programs, while the InvGen benchmark contains only correct annotated programs. The properties found in both sets of programs rely on Boolean expressions of the C language, and therefore do not contain any quantifiers.

For the correct *bounded* Eureka programs, loops were expanded the required number of times, and unwinding assertions were introduced to ensure that the expansion was sufficient. For the other programs loops were expanded a reasonable number of times, with unwinding assumptions introduced to prevent executions with more iterations from being considered. Table I shows the results obtained with Z3

	Eureka						InvGen		Total		
VCGen	*Correct (s)*	#	*Faulty (s)*	#	*Total (s)*	#	*Total (s)*	#	*(s)*	#	
VCCNF^{GA}	464.74	2	146.29	3	611.03	5	150.24	1	761.27	6	
VCCNF^{G}	417.03	2	126.90	2	543.93	4	101.11	0	645.03	4	
VCCNF^{PA}	842.36	1	447.13	1	1289.49	2	564.00	0	1853.49	2	
VCCNF^{P}	850.52	1	520.40	2	1370.91	3	553.84	0	1924.75	3	
VCSSA^{GA}	183.80	5	149.07	4	332.87	9	117.88	2	450.75	11	
VCSSA^{G}	**92.75**	7	123.02	**27**	**215.77**	34	70.89	2	**286.66**	36	
VCSSA^{PA}	463.94	5	376.20	3	840.14	8	436.14	0	1276.28	8	
VCSSA^{P}	417.82	4	368.66	2	786.48	6	392.88	0	1179.36	6	
VCSP^{GA}	325.88	2	111.47	8	437.35	10	70.39	2	507.73	12	
VCSP^{G}	301.28	10	114.70	10	415.98	20	63.56	1	479.54	21	
VCSP^{PA}	437.28	1	129.54	8	566.83	9	247.08	1	813.90	10	
VCSP^{P}	559.95	1	244.85	3	804.80	4	291.71	0	1096.51	4	
VCLean^{GA}	328.43	18	109.82	5	438.25	23	46.27	12	484.52	35	
VCLean^{G}	291.30	**35**	109.34	9	400.63	**44**	43.68	**14**	444.31	**58**	
VCLean^{PA}	331.79	11	**76.24**	21	408.03	32	41.21	9	449.24	41	
VCLean^{P}	312.20	7	92.00	10	404.21	17	**39.34**	12	443.55	29	

TABLE I: Z3 solving time for benchmark programs (time in seconds)

for each VCGen separated by benchmark. The column labeled *Correct* (resp. *Faulty*) refers to the total solving time for all VCs generated from the correct Eureka programs (resp. faulty programs). The columns marked with *Total* refer to the total solving time for each benchmark set of programs; the sum of both is also shown in the final column. Columns marked with # show the number of times that each VCGen performed *better than the others*. Finally the bars on the right are just to simplify the comparison of the total results. The detailed results for each program can be found in the tool's webpage.

A considerable number of Eureka programs use *assumes* and *asserts* simply as *pre-* and *postconditions*, as opposed to InvGen programs which are densely populated with asserts. This is reflected in the table: the solving time difference between VCGens based on partial and global contexts is greater in the InvGen benchmark than it is in the Eureka benchmark. The results also confirm the trends identified previously (global contexts lead to faster solving; placing asserts in contexts increases solving time), but allow for an exception to be identified: in all three data sets (Eureka Correct, Eureka Faulty, and InvGen), VCSP^{PA} performs better than VCSP^{P} (nonetheless, VCSP^{G} behaves much better).

As before, the solving times for VCSP^{G} and VCCNF^{G} are close, with VCSP^{G} performing consistently better in all three benchmark datasets. The optimizations improve performance: VCSSA^{G} performs better than VCCNF^{G} in all three datasets, and VCLean outperform all the others. Note that this observation is not immediately visible in the Eureka Correct data, because the dataset contains an outlier program (bubblesort_safe-9.c) that biases the data heavily in favor of VCSSA. If this program is removed, the supremacy of VCLean is restored. Finally, it remains to discuss which variety of the VCLean performs better. The aggregate solving time for the benchmarks is inconclusive (with G, PA and P resulting in similar times), but if we take the number of programs in which each VCGen outperforms the others, the clear winner is VCLean^{G} (58 programs against 41 for VCLean^{PA}). This is in accordance with the general trend that using a global context without asserts seems to be the best choice.

The analysis of the data obtained with CVC4 and MathSAT confirms the general trends described above, but reveals some points that are solver-specific. In particular it reinforces the fact that VCSP^{PA} performs better than VCSP^{P} *with all solvers*; and with MathSAT several programs in the benchmarks time-out with all VCGens except VCSSA, which causes the latter to have the best aggregate solving time, supplanting VCLean. It is also interesting to note that VCSSA^{G} performs better with CVC4 than all other VCGens in most case studies.

VIII. Conclusions

Based on two well-known fundamental VCGen algorithms, we identified three orthogonal design dimensions, and proposed a conceptual framework (the VCGen Cube) that allowed us to define in a uniform way 6 hybrid VCGens, which had not (to our knowledge) been studied or assessed before. We then extended our framework by showing how optimizations implemented in popular tools could be carried over to the hybrid VCGens. We believe that this categorization is helpful for both users of existing program verification tools, since it helps in the interpretation of the output results, and designers of new tools comprising a VCGen. Finally, we presented an evaluation of the design dimensions in our framework. We remark that the equivalence of all the VCGens described in this paper are formally proved. The proofs are non-trivial and can be found in [24].

The results of our evaluation are compactly summarized as follows: we identified the general trends that it is almost always preferable to use a global context; asserts should not be included in the context; and the VCSSA$^{\mathsf{G}}$ and VCLean$^{\mathsf{G}}$ are the best VCGens overall. With Z3, those based on VCLean seem to be, for the example of Fig. 6, the only ones for which solving time grows slower than $\theta(N^3)$ (but recall that these VCGens may be a bad choice if traceability has high priority). However, we remark that based on our data it is not possible to elect a single 'best VCGen', since different VCGens beat all the others for a significant number of cases.

We are currently in the process of testing the VCGens discussed in this paper in Why3. This requires extending them to the much richer features of WhyML (the underlying programming/specification language of Why3), and will allow us to evaluate the VCGens with programs whose specifications make extensive use of quantifiers, which is not the case for the benchmarks used in this paper.

There are also specific reasons for integrating and testing alternative VCGens in other verification tools. We first remark that although our experimental results do not consider a SAT encoding, there is a good chance that using VCLean or the global context VCSSA$^{\mathsf{G}}$ could improve the performance of CBMC. Our results indicate that VCLean$^{\mathsf{G}}$ performs slightly better than VCLean$^{\mathsf{PA}}$ with Z3: since Boogie is designed to work with this solver it might be interesting to consider testing an alternative global context algorithm for Boogie.

We remark that the two VCGens from which we departed have their origins in two different traditions and families of tools. Our work here will hopefully help bridging a gap between the deductive verification and software model checking communities and thus contribute towards a uniform framework that makes program verification more accessible to developers.

IX. Acknowledgments

This work is partially financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project 'POCI-01-0145-FEDER-006961', and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project 'UID/EEA/50014/2013'. The first author is also sponsored by FCT grant SFRH/BD/52236/2013.

References

[1] Alessandro Armando, Massimo Benerecetti, and Jacopo Mantovani. Counterexample-guided abstraction refinement for linear programs with arrays. *Automated Software Engineering*, 21(2):225–285, 2013.

[2] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[3] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Symposium Formal Methods for Components and Objects*, pages 364–387, 2005.

[4] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 82–87, 2005.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[6] Cláudio Belo Lourenço, Si-Mohamed Lamraoui, Shin Nakajima, and Jorge Sousa Pinto. Studying verification conditions for imperative programs. In *Proceedings of the 15th International Workshop on Automated Verification of Critical Systems*, 2015.

[7] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[8] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Annual Design Automation Conference*, pages 368–371. ACM, 2003.

[9] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, July 2012.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[11] Daniela da Cruz, Maria João Frade, and Jorge Sousa Pinto. Verification conditions for single-assignment programs. In *Proceedings of the 27th ACM Symposium On Applied Computing*, pages 1264–1270. ACM, 2012.

[12] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *22nd European Symposium on Programming*, pages 125–128, 2013.

[14] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of the 28th Symposium on Principles of Programming Languages*, pages 193–205. ACM, 2001.

[15] Patrice Godefroid and Shuvendu K. Lahiri. From program to logic: An introduction. In *Tools for Practical Software Verification, LASER, International Summer School 2011*, pages 31–44, 2011.

[16] Mike Gordon and Hélène Collavizza. Forward with hoare. In *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 101–121. Springer, 2010.

[17] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, pages 6:1–6:7. ACM, 2009.

[18] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 634–640. Springer-Verlag, 2009.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[20] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *Comput. J.*, 38(2):131–141, 1995.

[21] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A symbolic execution tool for verification. In *Proceedings of the 24th International Conference Computer Aided Verification*, pages 758–766, 2012.

[22] Si-Mohamed Lamraoui and Shin Nakajima. A formula-based approach for automatic fault localization of imperative programs. In *Proceedings of the 16th International Conference on Formal Engineering Methods*, pages 251–266. Springer, 2014.

[23] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 75–88, 2004.

[24] Cláudio Belo Lourenço. *Single-assignment programs and VCGens*. PhD thesis, University of Minho, 2018 (to appear).

[25] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[26] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2004.