A Comparative Study of Verification Condition Generators

Position Paper

Diogo Fialho[1] and Jorge Sousa Pinto[2]

[1] Departamento de Informática
Universidade da Beira Interior
Covilhã, Portugal
a17538@ubi.pt
[2] Departamento de Informática / CCTC
Universidade do Minho
Braga, Portugal
jsp@di.uminho.pt

Abstract. We propose an empirical comparison of two VCGen algorithms for imperative languages.

1 Introduction

The context of this work is the mechanical verification of imperative programs using program logics in the style introduced by Hoare [4]. The programs are annotated with specifications (in the form of preconditions and post-conditions) and other annotations such as loop invariants, whose purpose is to reduce user intervention during the verification process.

The most usual organisation of a tool for Program Verification consists of two components:

1. A *Verification Condition Generator* (VCGen for short) – a program that reads in the program to be verified and generates a set of proof obligations (called verification conditions). These are simply first-order formulas that do not contain any occurrences of program constructs.
2. A generic proof tool (a theorem prover or proof assistant) used to discharge the verification conditions. The tool should contain theories that allow for reasoning about the data types that are present in the source language.

The rationale for this organisation is that, on the assumption that the VCGen is sound with respect to the underlying operational semantics of the programming language, if all the verification conditions generated from a specification are valid then the specification is valid. Thus once the VCGen has been applied to the program, verification becomes a matter of establishing the validity of a set of first-order proof obligations. Many existing VCGens generate proof obligations that can be given as inputs to a choice of different theorem provers.

VCGens are a core component of the emerging *Design By Contract* (DBC) paradigm, in particular in its recent and exciting evolution: DBC with contract verification at compile/development time. A tight integration in a classic compilation process gives rise to the development of another emerging concept, namely *certifying compilers*.

Several approaches to the design of VCGens exist. Nevertheless, to the best of our knowledge, there is no clear comparison between different VCGens. The present work aims to establish an empirical comparative study between two different algorithms for the generation of verification conditions.

2 Setting

Figure 1 contains the syntax definition for the programming language (with categories B, E, C for boolean and integer expressions and commands), logical assertions A and specifications S.

We assume a standard semantics for programs, assertions and specifications. Program expressions are interpreted in program *states*, which are partial mappings from variables to (the interpretation of) integers. The same is true of assertions, which are interpreted as *true* or *false* in the standard model for a first-order theory of integers (with states seen as valuations). Specifications, on the other hand, are simply interpreted in $\{true, false\}$.

Informally, a specification $\{P\}\,\pi\,\{Q\}$ is valid (i.e., interpreted as *true*) if when π is executed in an initial state in which the precondition P is *true*, then either execution does not terminate or if it does, then the post-condition Q will be *true* in the final state. This notion is called a *partial correctness* specification since termination is not guaranteed.

In the following, we will also need the following alternative syntactic definition of programs, as (possibly empty) sequences of commands.

$$P ::= C\,;\,P \mid \varepsilon$$
$$C ::= \mathbf{skip} \mid V := E \mid \mathbf{if}\ E\ \mathbf{then}\ P\ \mathbf{else}\ P \mid \mathbf{while}\,(E)\,\mathbf{do}\ P$$

Note that it is straightforward to define translations between the two notions of program; the second notion simply imposes a left-associative view of the sequencing construct.

3 Comparing VCGens

In this project two different VCGens will be implemented. The basic algorithms are given in Figures 2 and 3.

The first is VCGen based on weakest-preconditions, as calculated by the auxiliary function $\mathrm{wp}\,(\cdot,\cdot)$. This is a classic algorithm (see for instance [5]) that takes as inputs a program and a post-condition; the precondition in the given specification merely generates an additional verification condition. In Figure 2 the notation $Q[x \mapsto e]$ denotes the substitution of e for x in Q, and $[\ a\ ]$ with

$$
\begin{array}{l}
B ::= \mathbf{true} \mid \mathbf{false} \\
\quad \mid \; B \;\&\&\; B \mid B \parallel B \mid !B \\
\quad \mid \; E == E \mid E < E \mid E <= E \mid E > E \mid E >= E \mid E != E \\
\\
E ::= \cdots -2, -1, 0, 1, 2 \ldots \mid \mathcal{V} \\
\quad \mid \; -E \mid E + E \mid E - E \mid E * E \mid E \;\mathbf{div}\; E \mid E \;\mathbf{mod}\; E \\
\\
C ::= \mathbf{skip} \mid C \,;\, C \mid V := E \mid \mathbf{if}\; E \;\mathbf{then}\; C \;\mathbf{else}\; C \mid \mathbf{while}\,(E)\,\mathbf{do}\; C \\
\\
A ::= \mathbf{true} \mid \mathbf{false} \\
\quad \mid \; A \;\&\&\; A \mid A \parallel A \mid !A \mid \forall\, \mathcal{V}.\, A \mid \exists\, \mathcal{V}.\, A \mid A \rightarrow A \\
\quad \mid \; E == E \mid E < E \mid E <= E \mid E > E \mid E >= E \mid E != E \\
\\
S ::= \{A\}\, C\, \{A\}
\end{array}
$$

Fig. 1. Language syntax

a an assertion denotes the *universal closure* of a, i.e., the formula obtained by universally quantifying over all the free variables in a.

The second VCGen is based on Hoare Logic with updates [3], which borrows ideas from JavaCardDL [1], a Dynamic Logic for *Java Card* programs. Specifications are extended with an update U (a partial map from variables to expressions); updates may be applied to expressions, assertions, and states.

Informally, the meaning of a specification $\{P\}[U]\,\pi\,\{Q\}$ is that if s is an initial state in which the precondition P is *true* and π is executed in the state $U(s)$, then either execution does not terminate or if it does, then the post-condition Q will be *true* in the final state.

Both VCGens are sound with respect to the standard transition semantics of the programming language, which means that programs that do not satisfy their specifications cannot be verified with success. Our aim is to implement both VCGens using as output language the standard SMT library concrete syntax for first-order assertions (thus we generate a format that can be given as input to a whole family of theorem provers).

Our main goal is to compare in useful ways the two algorithms: on one hand, comparing the sets of verification conditions generated, notably with respect to their number/size and their growth with the size of the program. On the other hand, the execution time of the algorithms will also be measured and compared.

4 Extensions

A second part of the work will consist in adding important extensions to the basic programming language, as follows.

Exception Mechanism. Adding exceptions to the language (through the introduction of **throw**$\cdot$ and **try** $\cdot$ **catch** $\cdot$ constructs) will be useful because it

$$
\begin{aligned}
\mathrm{wp}\,(\mathbf{skip}, Q) &= Q \\
\mathrm{wp}\,(x := e, Q) &= Q[x \mapsto e] \\
\mathrm{wp}\,(C_1; C_2, R) &= \mathrm{wp}\,(C_1, \mathrm{wp}\,(C_2, R)) \\
\mathrm{wp}\,(\mathbf{if}\ B\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) &= B\ \rightarrow\ \mathrm{wp}\,(C_t, Q) \\
&\quad \&\& \\
&\quad !B\ \rightarrow\ \mathrm{wp}\,(C_f, Q) \\
\mathrm{wp}\,(\mathbf{while}\,\{I\}\,(B)\,\mathbf{do}\ C, Q) &= I
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{vc}\,(\mathbf{skip}, Q) &= \emptyset \\
\mathrm{vc}\,(x := e, Q) &= \emptyset \\
\mathrm{vc}\,(C_1; C_2, Q) &= \mathrm{vc}\,(C_1, (\mathrm{wp}\,(C_2, Q))) \cup\ \mathrm{vc}\,(C_2, Q) \\
\mathrm{vc}\,(\mathbf{if}\ B\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) &= \mathrm{vc}\,(C_t, Q) \cup\ \mathrm{vc}\,(C_f, Q) \\
\mathrm{vc}\,(\mathbf{while}\,\{I\}\,(B)\,\mathbf{do}\ C, Q) &= \{[\ (I\ \&\&\ B)\ \rightarrow\ \mathrm{wp}\,(C, I)\ ]\} \\
&\quad \cup\ \mathrm{vc}\,(C, I)\} \\
&\quad \cup\ \{[\ (I\ \&\&\ !B)\ \rightarrow\ Q\ ]\}
\end{aligned}
$$

$$
\mathrm{vcg}\,(\{P\}\,C\,\{Q\}) = \{[\ P\ \rightarrow\ \mathrm{wp}\,(C, Q)\ ]\} \cup\ \mathrm{vc}\,(C, Q)
$$

Fig. 2. A VCGen based on weakest preconditions

$$
\begin{aligned}
\mathrm{vcg}\,(\{P\}[U]\,\mathbf{skip}\,;\ s\,\{Q\}) &= \mathrm{vcg}\,(\{P\}[U]\,l\,\{Q\}) \\
\mathrm{vcg}\,(\{P\}[U]\,x := e\,;\ s\,\{Q\}) &= \mathrm{vcg}\,(\{P\}[U\,; x := e]\,l\,\{Q\}) \\
\mathrm{vcg}\,(\{P\}[U]\,\varepsilon\,\{Q\}) &= [\ P\ \rightarrow\ U\,(Q)\ ] \\
\mathrm{vcg}\,(\{P\}[U]\,\mathbf{while}\,\{I\}\,(B)\,\mathbf{do}\ C\,;\ s\,\{Q\}) &= \{[\ P\ \rightarrow\ U\,(I)\ ]\} \\
&\quad \cup\ \mathrm{vcg}\,(\{I\ \&\&\ \lfloor B\rfloor\}[\emptyset]\,C\,\{I\}) \\
&\quad \cup\ \mathrm{vcg}\,(\{I\ \&\&\ !\lfloor B\rfloor\}[\emptyset]\,l\,\{Q\}) \\
\mathrm{vcg}\,(\{P\}[U]\,\mathbf{if}\ B\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\,;\ s\,\{Q\}) &= \mathrm{vcg}\,(\{P\ \&\&\ U\,(\lfloor B\rfloor)\}[U]\,C_t\,;\ s\,\{Q\}) \\
&\quad \cup\ \mathrm{vcg}\,(\{P\ \&\&\ !U\,(\lfloor B\rfloor)\}[U]\,C_f\,;\ s\,\{Q\})
\end{aligned}
$$

Fig. 3. A VCGen based on Updates

captures abrupt control transfer situations (like those introduced by **break** commands).

Arrays. Arrays present an obstacle to Hoare Logic-based reasoning, since they introduce an opportunity for index *aliasing*. A more ambitious goal would be to cope with pointers data-structures constructed in heap memory.

Procedure and Function Calls. Handling subroutines adequately has always been a challenge for Hoare Logic, and initial solutions were later proved to be incorrect. We are interested in extending the language with procedures for which specifications are given, to allow for modular reasoning.

All of these mechanisms have been addressed in the literature and are part of common program verification systems. However, the solutions adopted have typically not been given as simple extensions to a basic VCGen. Solutions include

either intricate translations into simpler intermediate languages (say, a guarded command language in ESC/Java [6] or an ML-like language in Why [2]), or complex extensions to Hoare Logic that are difficult to read as VCGen algorithms.

We have already investigated extending the VCGen of Figure 2 to cope with these constructs, and the VCGen of Figure 3 can be extended using ideas that have been adopted in JavaCardDL. Again, we will produce an empirical comparison of the sets of verification conditions generated by extensions to both algorithms.

5 Conclusion and Motivations

Nowadays there are several ways to apply/implement the concept of VCGen. However there is no comparative study which provides clear answers to the following questions. Which kind of VCGen

- Is more adequate for the automatization of the contract verification process?
- Has a simpler and more compact implementation?
- Is more appropriate for a smooth integration in the classical software engineering process, in particular for embedded systems?

This work aims to answer theses questions in a pragmatic way, by the use of a experimental approach.

An important outcome of this ongoing study is the setting up of a base of knowledge that can be used in related projects . In this context this work is one of several preliminary studies for the implementation of a source code level PCC architecture for embeded systems done in the context of a funded portuguese research project[3].

References

1. Bernhard Beckert. A dynamic logic for the formal verification of java card programs. In Isabelle Attali and Thomas P. Jensen, editors, *Java Card Workshop*, volume 2041 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2000.
2. Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
3. Reiner Hähnle and Richard Bubel. A hoare-style caulculus with explicit state updates. Department of Computer Science, Chalmers University of Technology.
4. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
5. Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *Comput. J.*, 38(2):131–141, 1995.
6. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking java programs via guarded commands. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 110–111, London, UK, 1999. Springer-Verlag.

[3] The RESCUE project, funded by the Portuguese Research Foundation (FCT).