Semantics-based generation of verification conditions by program specialization

E. De Angelis F. Fioravanti
DEC, University "G. d'Annunzio" of Chieti-Pescara, Italy
emanuele.deangelis@unich.it
fabio.fioravanti@unich.it

A. Pettorossi
DICII, University of Rome Tor Vergata, Roma, Italy
pettorossi@info.uniroma2.it

M. Proietti
CNR-IASI, Roma, Italy
maurizio.proietti@iasi.cnr.it

Abstract

We present a method for automatically generating verification conditions for a class of imperative programs and safety properties. Our method is parametric with respect to the semantics of the imperative programming language, as it specializes, by using unfold/fold transformation rules, a Horn clause interpreter that encodes that semantics.

We define a multi-step operational semantics for a fragment of the C language and compare the verification conditions obtained by using this semantics with those obtained by using a more traditional small-step semantics. The flexibility of the approach is further demonstrated by showing that it is possible to easily take into account alternative operational semantics definitions for modeling new language features. Finally, we provide an experimental evaluation of the method by generating verification conditions using the multi-step and the small-step semantics for a few hundreds of programs taken from various publicly available benchmarks, and by checking the satisfiability of these verification conditions by using state-of-the-art Horn clause solvers. These experiments show that automated verification of programs from a formal definition of the operational semantics is indeed feasible in practice.

Categories and Subject Descriptors I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation, Program verification; F.3.1 [*Logic and Meaning of Programs*]: Semantics of Programming Languages—Partial evaluation, Program analysis; F.3.2 [*Logic and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Logic and constraint programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods, Model checking

General Terms Languages, Theory, Verification

Keywords Software model checking, program verification, program specialization, semantics of programming languages, verification conditions, constraint logic programming, Horn clauses.

PPDP '15, July 14–16, 2015, Siena, Italy.

http://dx.doi.org/10.1145/2790449.2790529

1. Introduction

A well-established trend in the verification of program correctness relies on the generation of *verification conditions* (VCs, for short) from the program code [2, 9, 24]. Verification conditions are logical formulas whose satisfiability implies program correctness and can be verified, if at all possible,[1] by using specialized theorem provers or *Satisfiabilty Modulo Theories* (SMT) solvers [4, 15].

Recently, *constrained Horn clauses* have been proposed as a common encoding format for software verification problems, thus facilitating the interoperability of different software model checkers, and efficient solvers are available for checking the satisfiability of Horn-based verification conditions [4, 12, 15, 23]. The notion of a constrained Horn clause is equivalent to the notion of a clause in *Constraint Logic Programming* (CLP) [26], where constraints can refer to any first order theory. (The choice of the terminology very much depends on the context of use.)

Typically, verification conditions are obtained by using a *verification condition generator* which is a special-purpose software component that implements algorithms that are specifically tailored to handle the (syntax and semantics of the) programming language under consideration and the class of properties of interest.

A VC generator takes as input a program written in a given programming language, and a property of that program to be verified, and by applying axiomatic rules à la Floyd-Hoare, it produces as output a set of verification conditions.

Building a VC generator for programs written in a different language, or in an extension of the considered language, or even for programs in the same language but with a different semantics (for instance, small-step, or multi-step, or big-step semantics) requires the design and the implementation of a new, ad hoc algorithm.

In this paper we present a method for generating verification conditions that is based on the Horn clause encoding of the operational semantics of the programming language and uses program specialization based on unfold/fold rules.

The use of CLP program specialization for analyzing programs is not novel. In [36] it has been used for analyzing imperative programs and in [1] for analyzing Java bytecode. In [13] VCs are generated from a small-step semantics, for verifying imperative programs using iterated specialization. Here we extend and further develop the VC generation method of [13], and we demonstrate its generality, flexibility, and the performance that we have achieved in our implementation.

Given an imperative program P and a safety property, we introduce a CLP program I, which defines a predicate `unsafe` such that P is safe if and only if the atom `unsafe` is not derivable from I. The CLP program I is parametric with respect to the imperative

[1] Recall that the problem of verifying program correctness is undecidable.

program P, the operational semantics of its imperative language, the property to be proved, and the logic used for specifying the property of interest (in this case, the reachability of an unsafe state).

The verification conditions are obtained by specializing program I with respect to its parameters. The specialization process is performed by applying semantics-preserving unfold/fold transformation rules [16] guided by a strategy designed for verification condition generation (VCG strategy, for short). Thus, the correctness of the verification conditions follows directly from the correctness of the transformation rules used during program specialization.

In our approach, similarly to [6, 31, 35], we use a formal representation of the operational semantics as an explicit parameter of the verification problem. One of the most significant advantages of this approach is that it enables the design of widely applicable VC generators for programs written in different programming languages, and for different operational semantics of the same language, by making small modifications only. Additionally, having a formal model of the operational semantics specified as a CLP program facilitates the design of interpreters which are actually executable by a Prolog system.

The contributions of the paper can be summarized as follows.
- We have defined a multi-step operational semantics for a fragment of the C language.
- We have designed a VCG strategy which is parametric with respect to the operational semantics of the imperative language under consideration and the logic used for specifying the property of interest.
- We have compared the verification conditions obtained by applying our VCG strategy on the multi-step semantics, with those obtained by using the same VCG strategy on a more traditional small-step semantics. Indeed, although these two semantics are equivalent with respect to the specification of the input-output behavior of the programs, they show differences in the structure of the verification conditions that are generated and the subsequent ability of an automatic system to prove the program properties of interest.
- We have further demonstrated the flexibility of the approach by showing that it is possible, with a very low effort, to take into account alternative operational semantics definitions for modeling new language features.
- Finally, we have empirically proved the feasibility of the approach by performing an experimental evaluation. We have generated verification conditions using the multi-step semantics and small-step semantics for a few hundreds of programs taken from various publicly available benchmarks. We have also checked the satisfiability of these verifications conditions by using state-of-the-art Horn clause solvers such as QARMC [23], Z3 [15], and MathSAT [4]. Our experiments show that, when compared with the HSF(C) software model checker [23], which makes use of an *ad hoc* technique for generating VCs, our semantics-based approach to VC generation incurs in a relatively small increase of verification time and, interestingly enough, determines a significant improvement of accuracy over HSF(C) itself.

In conclusion, we have demonstrated that the use of program specialization for generating VCs provides great flexibility with little performance overhead, and thus it is viable also in the practice of automatic program verification.

2. An Imperative Language and its Operational Semantics

We consider programs written in an imperative language, subset of the C intermediate Language (CIL) [34], manipulating objects of elementary types, such as integers or characters. The language presented in this paper, is an extension of that in [13]. In particular, (i) functions can be recursively defined, (ii) there is an `abort` command which causes the abrupt termination of the execution of the program. The syntax of our language is shown in Table 1.

Language assumptions. We assume that in our language: (i) labels are totally ordered and every label occurs in every program no more than once, (ii) there are no blocks, nor structures, nor pointers, (iii) expressions have no side effects, but functions can have side effects. Without loss of generality, we also assume that the last command of every program is ℓ_h : `halt` and no other `halt` command occurs in the program.

In order to present the *multi-step* (*MS*) operational semantics of our imperative language (see, for instance, [37]), we introduce the following functions and data structures. Let the *global variables* of a program be those introduced in the declarations of the program, and the *local variables* of a function be those introduced in the declarations of the function definition. We assume that the value of every variable can be denoted by an integer.

A *global environment* $\delta\colon \mathit{Vars} \to \mathbb{Z}$, is a function that maps global variables to their integer values. A *local environment* $\sigma\colon \mathit{Vars} \to \mathbb{Z}$, is a function that maps function parameters and local variables to their integer values.

A *configuration* is a pair $\langle\!\langle c, \gamma \rangle\!\rangle$, where: (i) c is a labelled command, and (ii) γ is either a pair $\langle \delta, \sigma \rangle$ in case of regular execution (the configuration is said to be *regular*) or a triple $\langle \bot, \delta, \sigma \rangle$ in case of an aborted execution (the configuration is said to be *aborted*), where δ is a global environment, σ is a local environment, and $\bot$ is a symbol denoting aborted execution.

Given a (local or global) environment φ, a variable x, and an integer v, the term $\mathit{update}(\varphi, x, v)$ denotes the environment φ' that is equal to φ, except that $\varphi'(x) = v$.

For any program P, for any label ℓ, (i) $at(\ell)$ denotes the command in P with label ℓ, and (ii) *nextlab*(ℓ) denotes the label of the command that is written in P *immediately after* the command with label ℓ. Given a function f, the first command of f is called the *entry point* of f and its label is denoted by $\mathit{firstlab}(f)$. For any expression e, any global environment δ, and any local environment σ, $[\![e]\!]\,\delta\,\sigma$ is the integer value of e. For instance, if x is a global variable and $\delta(x) = 2$, then $[\![x+1]\!]\,\delta\,\sigma = 3$.

2.1 Multi-step semantics

The *MS* semantics is represented as a binary transition relation between configurations, denoted $\Longrightarrow$, which is defined by the following rules $R1$–$R5$. As usual, by $\Longrightarrow^*$ we denote the reflexive, transitive closure of $\Longrightarrow$. If $C_1 \Longrightarrow C_2$ or $C_1 \Longrightarrow^* C_2$ we say that C_1 is the *source configuration* and C_2 is the *target configuration*.

$(R1)$ *Assignment*. Let v be the integer $[\![e]\!]\,\delta\,\sigma$.

$$\langle\!\langle \ell\colon x = e,\ \langle \delta, \sigma \rangle \rangle\!\rangle \Longrightarrow \langle\!\langle at(\mathit{nextlab}(\ell)),\ \mathit{update}(\langle \delta, \sigma \rangle, x, v) \rangle\!\rangle$$

where update has been extended to pairs of functions with non overlapping domains as follows: $\mathit{update}(\langle \delta, \sigma \rangle, x, v)$ is
- $\langle \mathit{update}(\delta, x, v), \sigma \rangle$, if x is a global variable,
- $\langle \delta, \mathit{update}(\sigma, x, v) \rangle$, if x is a local variable or a function parameter.

Informally, an assignment updates either the global environment δ or the local environment σ.

$(R2)$ *Function call*. During the execution a function definition one of the following situations may occur: either execution aborts (see rule $(R2a)$), or it proceeds regularly and the value of a given expression is returned (see rule $(R2r)$).

Let $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_h\}$ be the set of the formal parameters and the set of the local variables, respectively, of the function f.

$$(R2a)\ \langle\!\langle \ell\colon x = f(e_1, \ldots, e_k), \langle \delta, \sigma \rangle \rangle\!\rangle \Longrightarrow \langle\!\langle \ell_a\colon \texttt{abort}, \langle \bot, \delta', \sigma \rangle \rangle\!\rangle$$
$$\text{if } \langle\!\langle at(\mathit{firstlab}(f)), \langle \delta, \overline{\sigma} \rangle \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_a\colon \texttt{abort}, \langle \bot, \delta', \sigma' \rangle \rangle\!\rangle$$

where $\overline{\sigma}$ is a local environment used in the definition of f, the expressions $e_1, \ldots, e_k$ are evaluated in the caller environment and

$x, y, \ldots \in$ *Vars*	(variable identifiers)		$f, g, \ldots \in$ *Functs*	(function identifiers)	
$\ell, \ell_1, \ldots \in$ *Labs*	(labels)		*const* $\in \mathbb{Z}$	(integer constants, character constants, ...)	
type $\in$ *Types*	(`int`, `char`, ...)		*uop*, *bop* $\in$ *Ops*	(unary and binary operators: $+, -, \leq, \ldots$)	
prog ::= decl* fundef* lab_cmd+	(programs)		decl ::= *type* x	(declarations)	
fundef ::= *type* f (decl*) { decl* lab_cmd+ }	(function definitions)		lab_cmd ::= ℓ : cmd	(labelled commands)	
expr ::= *const* $\mid x \mid$ *uop* expr $\mid$ expr *bop* expr	(expressions)				
cmd ::= x = expr $\mid x = f($expr*$) \mid$ `return` expr $\mid$ `goto` $\ell \mid$ `if (`expr`)` ℓ_1 `else` $\ell_2 \mid$ `abort` $\mid$ `halt`	(commands)				

Table 1. Syntax of the imperative language under consideration. Superscripts $^+$ and * denote non-empty and possibly empty finite sequences, respectively. Commands occurring in sequences are possibly separated by semicolons.

their values are bound to the function parameters; $\overline{\sigma}$ is of the form: $\{\langle x_1, [\![e_1]\!]\,\delta\,\sigma\rangle, \ldots, \langle x_k, [\![e_k]\!]\,\delta\,\sigma\rangle, \langle y_1, n_1\rangle, \ldots, \langle y_h, n_h\rangle\}$, for some values $n_1, \ldots, n_h$ in $\mathbb{Z}$ (indeed, when the local variables $y_1, \ldots, y_h$ are declared, they are not initialized).[2]

$(R2r)$ $\langle\!\langle \ell : x = f(e_1, \ldots, e_k),\ \langle\delta, \sigma\rangle\rangle\!\rangle \implies$
$\langle\!\langle at(nextlab(\ell)), update(\langle\delta', \sigma\rangle, x, [\![e]\!]\,\delta'\,\sigma')\rangle\!\rangle$
if $\langle\!\langle at(firstlab(f)),\ \langle\delta, \overline{\sigma}\rangle\rangle\!\rangle \implies^* \langle\!\langle \ell_r : \texttt{return}\ e,\ \langle\delta', \sigma'\rangle\rangle\!\rangle$

Informally, a function call either (i) aborts, if the execution of the function definition eventually leads to an aborted configuration, or (ii) updates the environment using the value returned by the function definition and continues executing the command that occurs after the function call.

$(R3)$ *Abort*. $\langle\!\langle \ell_a : \texttt{abort},\ \langle\delta, \sigma\rangle\rangle\!\rangle \implies \langle\!\langle \ell_a : \texttt{abort},\ \langle\bot, \delta, \sigma\rangle\rangle\!\rangle$

The `abort` command determines the move from a regular configuration to an aborted configuration.

$(R4)$ *Conditional*. Let v be the integer $[\![e]\!]\,\delta\,\sigma$.

$\langle\!\langle \ell : \texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2,\ \langle\delta, \sigma\rangle\rangle\!\rangle \implies \langle\!\langle at(\ell_1),\ \langle\delta, \sigma\rangle\rangle\!\rangle$ if $v \neq 0$

$\langle\!\langle \ell : \texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2,\ \langle\delta, \sigma\rangle\rangle\!\rangle \implies \langle\!\langle at(\ell_2),\ \langle\delta, \sigma\rangle\rangle\!\rangle$ if $v = 0$

Depending on the evaluation of the expression used in the condition, an if-then-else command follows either the 'then' branch or the 'else' branch.

$(R5)$ *Jump*. $\langle\!\langle \ell : \texttt{goto}\ \ell',\ \langle\delta, \sigma\rangle\rangle\!\rangle \implies \langle\!\langle at(\ell'),\ \langle\delta, \sigma\rangle\rangle\!\rangle$

The `goto` ℓ' command simply makes the program execution to continue from the command with label ℓ'.

Note that rules are given neither for the `halt` command, nor the `return` commands, nor for aborted configurations (indeed, rule $(R3)$ for the `abort` command is applied only when the `abort` command occurs in a regular configuration).

3. Encoding Program Safety using Constraint Logic Programs

In this section we define program safety and we show how to encode it as a CLP program.

Given a program P acting on the global variables $z_1, \ldots, z_r$, we define an *initial configuration* to be a triple: $\langle\!\langle \ell_0 : c_0,\ \delta_{Init},\ \sigma_{Init}\rangle\!\rangle$, where: (i) $\ell_0 : c_0$ is the first command of P, (ii) δ_{Init} is the initial global environment of the form: $\{\langle z_1, n_1\rangle, \ldots, \langle z_r, n_r\rangle\}$, where $n_1, \ldots, n_r$ are some given integers in $\mathbb{Z}$, and (iii) the initial local environment σ_{Init} is an empty function, that is, a function whose domain is the empty set (indeed, there are no local variables in the initial configuration). A *final configuration* is either an aborted configuration or a configuration whose command is `halt`. An *error configuration* is a final configuration that satisfies a given undesirable property as we now specify.

Safety. The safety of a program P is defined as the non-reachability of an error configuration from an initial configuration through an execution of P.

[2] Since the values of the n_i's are left unspecified, this transition is nondeterministic.

We may formalize safety by means of an *unsafety triple* of the form $\{\!\{Init\}\!\}\ P\ \{\!\{Err\}\!\}$, where *Init* denotes a set of initial configurations, and *Err* denotes a set of error configurations. We say that a program P is *unsafe* with respect to *Init* and *Err*, if there exist $C_i \in Init$ and $C_e \in Err$ such that $C_i \implies^* C_e$. A program is said to be *safe* if it is not unsafe.

Now we will show how to encode as a CLP program the multi-step semantics and an unsafety triple.

3.1 CLP encoding of the interpreter for multi-step semantics

We assume that the reader is familiar with the basic notions of constraint logic programming [26]. We will consider constraint logic programs with linear constraints over the set $\mathbb{Z}$ of the integer numbers. The semantics of a CLP program I is defined to be the *least* $\mathbb{Z}$*-model* of I, denoted $M(I)$, that is, the least $\mathbb{Z}$-interpretation in which every clause of I is true [26].

The transition relation $\implies$ between configurations and its reflexive, transitive closure $\implies^*$ are encoded by the binary predicates `tr` and `reach`, respectively. These predicates, shown in Table 2, constitute the CLP interpreter for the multi-step semantics of the imperative language under consideration. We have the clauses relative to: (i) assignments (clause `1`), (ii) function calls (clauses `2a` and `2r`), (ii) aborts (clause `3`), (iv) conditionals (clauses `4t` and `4f`), (v) jumps (clause `5`), (vi) environment updates (clauses `6` and `7`) and (vii) reachability of configurations (clauses `8` and `9`).

Configurations are represented by using terms of the form `cf(cmd(L,C),Env)`, where: (i) `L` and `C` encode the label and the command, respectively, (ii) `Env` is either a pair `(D,S)` or a triple `(bot,D,S)`, where `bot` represents the symbol $\bot$, and `D` and `S` encode the global and the local environment, respectively.

The term `asgn(X,expr(E))` encodes the assignment of the value of the expression `E` to the variable `X`. The predicate `eval(E,(D,S),V)` evaluates the expression `E` to the value `V` in the global environment `D` and the local environment `S`. The predicate `eval_list` extends the applicability of the predicate `eval` to a lists of expressions. The predicate `beval(E,(D,S))` holds if the value of the expression `E` is not 0 in the environment `(D,S)`.

The predicate `at(L,C)` returns the command `C` with label `L`. The predicate `nextlab(L,L1)` returns the label `L1` of the command that is written immediately after the command with label `L`. The predicate `firstlab(F,L1)` returns the label `L1` of the first command of the definition of the function `F`. The predicate `build_funenv(F,Vs,FEnv)` takes as input the function identifier `F` and the list `Vs` of the values of the actual parameters, retrieves the definition of function `F` using the predicate `fun`, and builds the local environment `FEnv` to be used for starting the execution of the body of `F`.

The term `ite(E,L1,L2)` encodes the conditional command (`ite` stands for if-then-else), and labels `L1` and `L2` specify where to jump to, depending on the value of the expression `E`. The term `goto(L)` encodes the jump to label `L`. The predicates `global(X)` and `local(X)` hold if `X` denotes a global or a local variable, respectively. The predicate `update_glocal(E,X,V,E1)` updates the (global or local) environment `E` and produces the new (global or local) environment `E1` by binding the variable `X` to the value `V`.

```
1. tr(cf(cmd(L,asgn(X,expr(E))), (D,S)), cf(cmd(L1,C), (D1,S1))) :-
       eval(E,(D,S),V), update((D,S),X,V,(D1,S1)), nextlab(L,L1), at(L1,C).
2a.tr(cf(cmd(L,asgn(X,call(F,Es))),(D,S)), cf(cmd(LA,abort),(bot,D1,S))) :-
       eval_list(Es,D,S,Vs), build_funenv(F,Vs,FEnv), firstlab(F,FL), at(FL,C),
       reach(cf(cmd(FL,C),(D,FEnv)), cf(cmd(LA,abort),(bot,D1,S1))).
2r.tr(cf(cmd(L,asgn(X,call(F,Es))),(D,S)), cf(cmd(L2,C2),(D2,S2))) :-
       eval_list(Es,D,S,Vs), build_funenv(F,Vs,FEnv), firstlab(F,FL), at(FL,C),
       reach(cf(cmd(FL,C),(D,FEnv)), cf(cmd(LR,return(E)),(D1,S1))),
       eval(E,(D1,S1),V), update((D1,S),X,V,(D2,S2)), nextlab(L,L2), at(L2,C2).
3. tr(cf(cmd(L,abort),(D,S)), cf(cmd(L,abort),(bot,D,S))).
4t.tr(cf(cmd(L,ite(E,L1,L2)),(D,S)), cf(cmd(L1,C),(D,S))) :- beval(E,(D,S)), at(L1,C).
4f.tr(cf(cmd(L,ite(E,L1,L2)),(D,S)), cf(cmd(L2,C),(D,S))) :- beval(not(E),(D,S)), at(L2,C).
5. tr(cf(cmd(L,goto(L1)),(D,S)), cf(cmd(L1,C),(D,S))) :- at(L1,C).
6. update((D,S),X,V,(D1,S)) :- global(X), update_glocal(D,X,V,D1).
7. update((D,S),X,V,(D,S1)) :- local(X), update_glocal(S,X,V,S1).
8. reach(C,C).
9. reach(C,C2) :- tr(C,C1), reach(C1,C2).
```

Table 2. The CLP interpreter for the multi-step operational semantics *MS*: the clauses for `tr` and `reach`.

3.2 CLP encoding of an unsafety triple

We encode any given unsafety triple $\{\!\{\mathit{Init}\}\!\}\ P\ \{\!\{\mathit{Err}\}\!\}$ by the CLP program I containing the following clause:

10. `unsafe :- initConf(C), reach(C,C1), errorConf(C1).`

together with the clauses defining: (i) the predicates `tr` and `reach` that encode the interpreter, (ii) the predicates `initConf` and `errorConf` that encode *Init* and *Err*, respectively, and (iii) the predicates that encode the commands of the imperative program P (among these last clauses we have those defining the predicate `at`).

Program I encodes the given unsafety triple in the sense that the unsafety triple holds (and thus the program P is unsafe) iff the atom `unsafe` belongs to the least $\mathbb{Z}$-model of I. We can state the following correctness result.

THEOREM 1. (Correctness of CLP Encoding) *Let I be the CLP encoding of any given unsafety triple $\{\!\{\mathit{Init}\}\!\}\ P\ \{\!\{\mathit{Err}\}\!\}$. The program P is safe with respect to Init and Err iff* $\texttt{unsafe} \notin M(I)$.

The proof of this theorem is similar to the proof of Theorem 1 in [13]. However, in this paper: (i) we use a slightly different representation of configurations (we do not use an execution stack), and (ii) the predicate `reach` has two arguments, instead of one only (this change is needed by the multi-step semantics *MS* for encoding the reachability relation within function calls).

4. Automatic generation of Verification Conditions by Program Specialization

In this section we present the *Verification Condition Generation strategy* (*VCG strategy*, for short) that we use for automatically generating verification conditions.

The VCG strategy (see Figure 1) takes as input the CLP program I encoding an unsafety triple as shown is Section 3, and produces by program specialization a set VCs of verification conditions, here encoded as the CLP program I_{sp}, such that I_{sp} is equivalent to I with respect to the atom `unsafe`, that is, $\texttt{unsafe} \notin M(I)$ iff $\texttt{unsafe} \notin M(I_{sp})$.

The VCG strategy works by performing the so-called *removal of the interpreter*, that is, by removing the overhead caused by the level of interpretation which is present in the initial program I, thereby generating the desired set I_{sp} of CLP clauses, whose call graph may be viewed as an abstraction of the control flow graph of the imperative program P.

Now, due to undecidability limitations, there is no algorithm for checking whether or not $\texttt{unsafe} \notin M(I)$. Moreover, even if one restricts himself to decidable subclasses of programs and properties, it is very hard to construct efficient algorithms for showing that $\texttt{unsafe} \notin M(I)$, thereby proving program safety, because in general for that proof, one has to make deductions within multiple constraint theories. We are not aware of any available tool which can perform in an automatic and effective way, the check that $\texttt{unsafe} \notin M(I)$, starting directly from the program I encoding of the unsafety triple.

In contrast, by relying on the fact that $\texttt{unsafe} \notin M(I)$ iff $\texttt{unsafe} \notin M(I_{sp})$ iff $I_{sp} \cup \{\texttt{:- unsafe.}\}$ is satisfiable, we can prove program safety by showing the satisfiability of $I_{sp} \cup \{\texttt{:- unsafe.}\}$. It turns out that this satisfiability check is often much easier than the check that $\texttt{unsafe} \notin M(I)$. This is due to the fact that the VCG strategy, when it specializes the CLP program I and produces the verification conditions I_{sp}, compiles away both the references to the commands of the program P and the references to the operational semantics of the imperative programming language. In practice, it is frequently the case that the satisfiability check required for showing program safety can successfully be performed by automatic tools which deal with Horn clauses with linear integer arithmetic constraints (see, for instance, [4, 12, 15, 23]).

4.1 The VCG strategy

During the application of the VCG strategy we use the following transformation rules: *unfolding*, *definition introduction*, and *folding* [16, 18]. In particular, VCG strategy unfolds the `initConf` and `errorConf` predicates which characterize the initial and the error configurations, the predicate `tr` encoding the operational semantics, thereby exploring the control flow graph of program P. New predicate definitions are introduced for calls to the recursive predicate `reach` and are used for applying the folding rule.

Unfolding. Given a clause C of the form `H :- c,L,A,R`, where `H` and `A` are atoms, `c` is a constraint, and `L` and `R` are (possibly empty) conjunctions of atoms, let $\{\texttt{K}_i \texttt{ :- c}_i\texttt{,B}_i \mid i = 1, \ldots, m\}$ be the set of the (renamed apart) clauses in program I such that, for $i = 1, \ldots, m$, `A` is unifiable with $\texttt{K}_i$ via the most general unifier ϑ_i and $(\texttt{c,c}_i)\,\vartheta_i$ is satisfiable. We define the following function *Unf*:

$$\mathit{Unf}(C, \texttt{A}, I) = \{(\texttt{H :- c,c}_i\texttt{,L,B}_i\texttt{,R})\,\vartheta_i \mid i = 1, \ldots, m\}$$

Each clause in $\mathit{Unf}(C, \texttt{A}, I)$ is said to be derived by *unfolding* C *w.r.t.* `A` using I.

The application of the unfolding rule during specialization is guided by annotations which tell whether or not the atoms that occur in bodies of clauses should be unfolded. These annotations ensure that any sequence of (sets of) clauses constructed by unfolding is finite. The reader may refer to [29] for a survey on related techniques.

In particular, every atom is marked by an *unfolding annotation*, which is: either (i) *unfoldable once*, or (ii) *fully unfoldable*, or (iii) *non-unfoldable*. An atom is said to be *unfoldable* if it is annotated as unfoldable once or fully unfoldable.

For a clause C, an atom `A` which is unfoldable once, and a program I, the unfolding rule derives the set $\mathit{Unf}(C,\mathtt{A},I)$ of clauses. For an atom `A` which is fully unfoldable, the unfolding rule derives the set $\mathit{FullUnf}(C,\mathtt{A},I)$ of clauses D defined as follows: either (i) $D \in \mathit{Unf}(C,\mathtt{A},I)$ and D contains no unfoldable atom in its body, or (ii) $D \in \mathit{FullUnf}(D',\mathtt{B},I)$ for some $D' \in \mathit{Unf}(C,\mathtt{A},I)$ and some unfoldable atom `B` occurring in the body of D'. Informally, for an atom `A` which is fully unfoldable, the unfolding rule is repeatedly applied to all clauses which are directly or indirectly derived by unfolding C w.r.t. `A` using I, until all unfoldable atoms have been unfolded.[3]

Definition introduction. The VCG strategy makes use of the function *Gen*, called *generalization operator*, for introducing new predicate definitions. Given a clause E: `H :- e, reach(cf1,cf2)`, $\mathit{Gen}(E)$ returns a clause G: `newr(X) :- reach(cf1,cf2)` such that: (i) `newr` is a new predicate symbol, and (ii) `X` is the set of variables occurring in `reach(cf1,cf2)`.

Note that, since the clauses produced by *Gen* keep no information about the constraints on the variables in `X`, the number of such clauses is finite (and depends on the number of the labeled commands of program P). This property will be used when proving the termination of the VCG strategy.

For the purpose of generating VCs considered in this paper, it suffices to use the simple generalization operator presented above, which generalizes the constraint `e` occurring in the body of clause E to the constraint `true` (which is left implicit in clause G) and keeps the configurations unaltered.

For reasons of space, we omit to define alternative generalization operators that keep some constraints (for instance, constraints belonging to a finite abstract domain) or that partially abstract subterms in the configurations (for instance, command labels) and may derive definitions which are more general, and thus, obtain more compact verification conditions.

More sophisticated generalization operators [19], which use widening and convex-hull [7, 8] and take into account the set of definitions already introduced, are needed when applying program specialization for verifying program correctness [11, 13]. However, in this paper we do not deal with specialization-based *verification*, and we only address the problem of generating the verification conditions.

Folding. Clause E: `H :- e, reach(cf1,cf2)` is *folded* by using clause G: `newr(X) :- reach(cf1,cf2)`, thereby deriving the new clause `H :- e, newr(X)`.

The correctness of the folding rule guarantees that the clause obtained by folding is equivalent to clause E with respect to the least $\mathbb{Z}$-model semantics [16, 18] (in our case, because the atom `newr(X)` is equivalent, by definition, to the atom `reach(cf1,cf2)`).

Termination and Correctness of the VCG strategy

The VCG strategy has two potential sources of non-termination: the loop within the UNFOLDING phase, and the *UDF-cycle*, that is the outer loop containing the UNFOLDING and the DEFINITION-INTRODUCTION & FOLDING phases. However, as already mentioned, the unfolding annotations of the atoms occurring in the body of the clauses guarantee the termination of the UNFOLDING phase. Moreover, we have shown above that the set of definitions which can be introduced by using the *Gen* function, is finite, and thus the set *InCls* will eventually become empty. Hence the whole VCG strategy terminates.

[3] The order in which clauses and atoms in their bodies are selected for unfolding is not relevant.

Input: Program I encoding a given unsafety triple.
Output: Program I_{sp} encoding the verification conditions VCs, such that `unsafe` $\in M(I)$ iff `unsafe` $\in M(I_{sp})$.

INITIALIZATION:
$I_{sp} := \emptyset$;
InCls $:= \{$`unsafe :- initConf(C),reach(C,C1),errorConf(C1)`$\}$;
Defs $:= \emptyset$;

while in *InCls* there is a clause C with an atom in its body *do*
  UNFOLDING: *SpC* $:= \mathit{Unf}(C, \mathtt{A}, I)$,
    where `A` is the leftmost atom in the body of C;
  while in *SpC* there is a clause D whose body contains an occurrence of an unfoldable atom `A` *do*
    SpC $:= (\mathit{SpC} - \{D\}) \cup U$,
    where $U = \mathit{Unf}(D, \mathtt{A}, I)$, if `A` is unfoldable once, and
      $U = \mathit{FullUnf}(D, \mathtt{A}, I)$, if `A` is fully unfoldable.
  end-while;
  DEFINITION-INTRODUCTION & FOLDING:
  while in *SpC* there is a clause E of the form:
    `H :- e, L, reach(cf1,cf2), R`
  where `H` is either `unsafe` or an atom of the form `newp(X)`, `e` is a constraint, and `L` and `R` are possibly empty conjunctions of atoms *do*
    if in *Defs* there is a (renamed apart) clause D of the form:
      `newq(V) :- B`
      where `V` is the set of variables occurring in `B` and,
      for some substitution ϑ, $\mathtt{B}\vartheta = $ `reach(cf1,cf2)`
    then *SpC* $:= (\mathit{SpC} - \{E\}) \cup \{$`H :- e, L,newq(V)`$\vartheta$`,R`$\}$;
    else let $\mathit{Gen}(E)$ be `newr(V) :- reach(cf1,cf2)`
      where: (i) `newr` is a predicate symbol not occurring in $I \cup$ *Defs*, and (ii) `V` is the set of variables occurring in `reach(cf1,cf2)`;
      Defs $:= \mathit{Defs} \cup \{\mathit{Gen}(E)\}$;
      InCls $:= \mathit{InCls} \cup \{\mathit{Gen}(E)\}$;
      SpC $:= (\mathit{SpC} - \{E\}) \cup \{$`H :- e, L,newr(V),R`$\}$
  end-while;
  InCls $:= \mathit{InCls} - \{C\}$;
  $I_{sp} := I_{sp} \cup \mathit{SpC}$;
end-while;

Figure 1. The Verification Condition Generation (VCG) strategy.

THEOREM 2. (Termination and Correctness of the VCG strategy) (i) *The VCG strategy terminates.* (ii) *Let program I_{sp} be the output obtained by applying the VCG strategy on the input program I. Then* `unsafe` $\in M(I)$ *iff* `unsafe` $\in M(I_{sp})$.

4.2 Applying the VCG Strategy

Let us consider the interpreter for the multi-step semantics presented in Section 3. In this example we will see in action the Verification Condition Generation strategy of Figure 1, when given in input the encoding of the unsafety triple $\{\!\{\mathit{Init}\}\!\}$ *gcd* $\{\!\{\mathit{Err}\}\!\}$, where *gcd* is the following C program for computing the greatest common divisor between two integers x and y, Init is the set of configurations such that $x \geq 1 \wedge y \geq 1$, and Err is the set of configurations such that $x < 0$.

Program *gcd*

```
int x, y;                              n1
int sub(int a, int b) {                n2
  int r = a-b;                         n3
  return r;                            n4
}                                      n5
void main() {                          n6
  while (x!=y)                         n7
    if (x>y) x = sub(x,y);             n8
    else     y = sub(y,x);             n9
}                                      n10
```

First, the source C program is translated into a program in the CIL subset complying with the syntax of Table 1. In particular, this translation replaces while loops by using conditionals and jumps. Then, the resulting CIL program is encoded into a set of clauses. In the *gcd* example, we derive the following clauses:

```
11. fun(sub,[a,b],[r],1).
12. at(1,asgn(r,minus(a,b))).
13. at(2,return(r)).
14. fun(main,[],[],3).
15. at(3,ite(neq(x,y),4,h)).
16. at(4,ite(gt(x,y),5,7)).
17. at(5,asgn(x,call(sub,[x,y]))).
18. at(6,goto(3)).
19. at(7,asgn(y,call(sub,[y,x]))).
20. at(8,goto(3)).
21. at(h,halt).
22. globals([x,y]).
```

where: (i) the predicate `fun` encodes function definitions, (ii) the predicate `at` encodes labeled commands, and (iii) the predicate `globals` encodes the list of identifiers for global variables. In particular, clause 11 encodes the `sub` function, having two formal parameters `[a,b]`, one local variable `[r]`, and whose definition starts with the command with label `1` (encoded by clause 12). Note that, by effect of the translation of C into CIL, the while loop at line n7 has been encoded into the conditional (`ite`) of clause 15 along with the jumps (`goto`) of clauses 18 and 20. The first argument of the `ite` command is the condition of the while loop. The second (third) argument is the label of the first command occurring in the 'then' ('else', respectively) branch of the conditional.

We also have the clauses for the initial and error configurations: `initConf(C,[X,Y]):- at(3,C), X>=1, Y>=1.` and `errorConf(C,[X,Y]):- at(h,C), X=< -1`. Note that `initConf` refers to the entry point of the `main` function (encoded by clause 15), and `errorConf` refers to the `halt` command (encoded by clause 21). Since the focus of this paper is the generation of the verification conditions, the properties characterizing the initial and error configurations have intentionally been left simple, and restricted to predicates definable by sets of non-recursive clauses. The interested reader may refer to [14] for more complex properties defined by a set of possibly recursive Horn clauses.

In order to generate a set of VCs for the *gcd* program, we have first to provide suitable unfolding annotations to the VCG strategy. In particular, (i) atoms whose predicate symbol belongs to the set {`initConf`, `errorConf`} are (annotated as) fully unfoldable; (ii) atoms of the form `tr(_,_)` are fully unfoldable if they are not unifiable with the heads of clauses `2a` and `2r`, otherwise they are unfoldable once; (iii) atoms of the form `reach(cf(Cmd,_),_)` are non-unfoldable when `Cmd` is an if-then-else command, or the entry point of a function definition, or a jump to a command different from if-then-else, otherwise the `reach` atoms are unfoldable once.

In the following, for reasons of readability, we will omit the round parentheses around the pair of lists denoting the global and local environments. The VCG strategy starts off by performing the UNFOLDING phase for the set $InCls = \{10\}$.

By unfolding clause 10 w.r.t. the atom `initConf(X)`, which is fully unfoldable, we get:

```
23. unsafe:- X>=1, Y>=1,
      reach(cf(cmd(3,ite(neq(x,y)),4,h),
                [(x,X),(y,Y)],[]),C), errorConf(C).
```

Then, the UNFOLDING phase selects the fully unfoldable atom `errorConf(C)`, as it is the only unfoldable atom in clause 23 (note that the `reach(_,_)` atom is not unfoldable because its command is an if-then-else). By unfolding `errorConf(C)` we get:

```
24. unsafe:- X>=1, Y>=1, X1=< -1,
      reach(cf(cmd(3,ite(neq(x,y)),4,h),[(x,X),(y,Y)],[]),
            cf(cmd(h,halt),[(x,X1),(y,Y1)],[])).
```

We have that no unfoldable atom occurs in the body of clause 24. Thus, we continue by executing the DEFINITION-INTRODUCTION & FOLDING phase. In order to fold clause 24 the following clause is introduced in *Defs* and added to *InCls*:

```
25. new3(X,Y,X1,Y1):-
      reach(cf(cmd(3,ite(neq(x,y)),4,h),[(x,X),(y,Y)],[]),
            cf(cmd(h,halt),[(x,X1),(y,Y1)],[])).
```

where `new3` is a new predicate symbol. By folding clause 24 w.r.t. the atom `reach` using clause 25 we get:

```
26. unsafe:- X>=1, Y>=1, X1=< -1, new3(X,Y,X1,Y1).
```

Since $InCls \neq \emptyset$, we perform one more iteration of the UDF-cycle.

By unfolding clause 25 w.r.t. the atom in its body we get:

```
27. new3(X,Y,X1,Y1):- X=Y,
      reach(cf(cmd(h,halt),[(x,X),(y,Y)],[]),
            cf(cmd(h,halt),[(x,X1),(y,Y1)],[])).
28. new3(X,Y,X1,Y1):- X>=Y+1,
      reach(cf(cmd(4,ite(gt(x,y)),5,7),[(x,X),(y,Y)],[]),
            cf(cmd(h,halt),[(x,X1),(y,Y1)],[])).
29. new3(X,Y,X1,Y1):- X+1=<Y,
      reach(cf(cmd(4,ite(gt(x,y)),5,7),[(x,X),(y,Y)],[]),
            cf(cmd(h,halt),[(x,X1),(y,Y1)],[])).
```

Note that the symbolic evaluation of the while loop condition `neq(x,y)` in clause 25 generates the three constraints `X=Y` (the exit condition), `X>=Y+1` and `X+1=<Y` in clauses 27–29.

We have that the atom in clause 27 is unfoldable, and hence, by reflexivity of the `reach` predicate (clause 8), we get the following constrained fact:

```
30. new3(X,Y,X,Y):- X=Y.
```

No unfoldable atom occurs in the body of clauses 28 and 29. In order to fold clause 28 the following clause is introduced in *Defs* and *InCls*:

```
31. new4(X,Y,X1,Y1):-
      reach(cf(cmd(4,ite(gt(x,y)),5,7),[(x,X),(y,Y)],[]),
            cf(cmd(h,halt),[(x,X1),(y,Y1)],[])).
```

By folding clauses 28 and 29 using clause 25 we get:

```
32. new3(X,Y,X1,Y1):- X>=Y+1, new4(X,Y,X1,Y1).
33. new3(X,Y,X1,Y1):- X+1=<Y, new4(X,Y,X1,Y1).
```

Since we have introduced a new definition, namely clause 31, $InCls \neq \emptyset$ and we need to perform one more iteration of the UDF-cycle. By unfolding clause 31 w.r.t. the atom `reach` we get:

```
34. new4(X,Y,X1,Y1):- X>=Y+1,
      reach(cf(cmd(5,asgn(x,call(sub,[x,y]))),
                [(x,X),(y,Y)],[]),
            cf(C,[(x,X1),(y,Y1)],[])).
35. new4(X,Y,X1,Y1):- X=<Y,
      reach(cf(cmd(7,asgn(y,call(sub,[y,x]))),
                [(x,X),(y,Y)],[]),
            cf(C,[(x,X1),(y,Y1)],[])).
```

Clauses 34 and 35 correspond to the 'then' and 'else' branches of the conditional at lines n8 and n9, respectively, of the *gcd* program.

The atom occurring in clause 34 is unfoldable; after the execution of some unfolding steps we get:

```
36. new4(X,Y,X3,Y3):- X>=Y+1, A=X, B=Y, X2=R1,
      reach(cf(cmd(1,asgn(r,minus(a,b))),
          [(x,X),(y,Y)],[(a,A),(b,B),(r,R)]),
            cf(cmd(2,return(r)),
          [(x,X1),(y,Y1)],[(a,A1),(b,B1),(r,R1)])),
```

```
    reach(cf(cmd(3,ite(neq(x,y)),4,h),
                [(x,X2),(y,Y1)],[]),
          cf(cmd(h,halt),[(x,X3),(y,Y3)],[]))).
```

We observe that: (i) the command occurring in the first argument of the first `reach` atom corresponds to the entry point of the `sub` function, and (ii) the command occurring in the first argument of the second `reach` atom is an `ite` command. Thus, none of the atoms of clause 36 are unfoldable.

Note also that the local environments in the first argument of the first `reach` atom is a list where new logical variables, namely `A`, `B`, and `R`, are associated with the parameters and local variable identifiers used by `sum`, that is, `a`, `b`, and `r`.

In order to fold the first atom occurring in the body of clause 36 the following clause is introduced:

```
37. new6(X,Y,A,B,R,X1,Y1,A1,B1,R1):-
    reach(cf(cmd(1,asgn(r,minus(a,b))),
        [(x,X),(y,Y)],[(a,A),(b,B),(r,R)]),
          cf(cmd(2,return(r)),
        [(x,X1),(y,Y1)],[(a,A1),(b,B1),(r,R1)]))).
```

By folding clause 36 using definition 37 we get:

```
38. new4(X,Y,X3,Y3):- X>=Y+1, A=X, B=Y, X2=R1,
    new6(X,Y,A,B,R,X1,Y1,A1,B1,R1),
    reach(cf(cmd(3,ite(neq(x,y)),4,h),
                [(x,X2),(y,Y1)],[]),
          cf(cmd(h,halt),[(x,X3),(y,Y3)],[]))).
```

In order to fold the second atom occurring in the body of clause 38 the VCG strategy does not require to introduce any new definition. Indeed, it is possible to fold clause 38 using clause 25 in *Defs* and we get:

```
39. new4(X,Y,X3,Y3):- X>=Y+1, A=X, B=Y, X2=R1,
      new6(X,Y,A,B,R,X1,Y1,A1,B1,R1), new3(X2,Y1,X3,Y3).
```

Clause 35, corresponding to the 'else' branch, is processed in a similar way, and we get:

```
40. new4(X,Y,X3,Y3):- X=<Y, A=Y, B=X, Y2=R1,
      new6(X,Y,A,B,R,X1,Y1,A1,B1,R1), new3(X1,Y2,X3,Y3).
```

Since we introduced the new predicate `new6` defined by clause 37, we start a new iteration of the UDF-cycle. By unfolding clause 37 w.r.t. the atom in its body we get:

```
41. new6(X,Y,A,B,R,X1,Y1,A1,B1,R1):- R1=A-B,
    reach(cf(cmd(2,return(r)),
        [(x,X),(y,Y)],[(a,A),(b,B),(r,R)]),
          cf(cmd(2,return(r)),
        [(x,X1),(y,Y1)],[(a,A1),(b,B1),(r,R1)]))).
```

The atom `reach` in the above clause is unfoldable. After one more unfolding step, by using the reflexivity of the `reach` predicate, we get:

```
42. new6(X,Y,A,B,R,X,Y,A,B,R1):- R1=A-B.
```

Since $InCls = \emptyset$, the VCG strategy terminates. The final, specialized program consists of the following set VC_{MS} of verification conditions:

```
43. new6(X,Y,A,B,R,X,Y,A,B,R1):- R1=A-B.
44. new4(X,Y,X3,Y3):- X>=Y+1, A=X, B=Y, X2=R1,
      new6(X,Y,A,B,R,X1,Y1,A1,B1,R1), new3(X2,Y1,X3,Y3).
45. new4(X,Y,X3,Y3):- X=<Y, A=Y, B=X, Y2=R1,
      new6(X,Y,A,B,R,X1,Y1,A1,B1,R1), new3(X1,Y2,X3,Y3).
46. new3(X,Y,X,Y):- X=Y.
47. new3(X,Y,X1,Y1):- X>=Y+1, new4(X,Y,X1,Y1).
48. new3(X,Y,X1,Y1):- X+1=<Y, new4(X,Y,X1,Y1).
49. unsafe:- X>=1, Y>=1, X1=< -1, new3(X,Y,X1,Y1).
```

5. Multi-step and small-step semantics compared

Now we will compare the multi-step operational semantics *MS* presented in Section 2 with a *small-step* operational semantics, denoted *SS*, that extends and improves the semantics presented in [13]. We will also discuss the main differences between the verification conditions we derive by applying the VCG strategy for these two different semantics.

The small-step semantics *SS* is similar to the multi-step semantics *MS* in the case of expressions, assignments, conditionals, and jumps.[4] These two semantics differ in the way they deal with function calls and function `return`'s.

Before making the comparison between the *MS* and the *SS* semantics, let us briefly recall the *SS* semantics for function calls (see [13], for details). The *SS* semantics keeps an execution stack (which is empty in the initial configurations), whose elements are called *activation frames*. Each activation frame contains information about a single function call. In particular, it includes (i) the label where to jump after returning from the function call, (ii) the variable used for storing the value returned by the call, and (iii) the local environment to be used during the execution of the function.

Configurations are encoded as terms of the form `cf(Cmd,D,T)` where `Cmd` is a labeled command, `D` is a global environment and `T` is a stack of activation frames.

When a function call of the form $\ell : x = f(e_1, \ldots, e_k)$ is encountered, the *SS* semantics 'dives into' the function definition and makes a transition from the configuration containing the function call to the configuration containing the entry point of f (that is, the command *at(firstlab*(f)*)*). When making this transition the *SS* semantics pushes a new activation frame on top of the execution stack as shown in the following clause:

```
tr(cf(cmd(L,asgn(X,call(F,Es))),D,T),
   cf(cmd(FL,C),D,[frame(L1,X,FEnv)|T])) :-
     nextlab(L,L1), loc_env(T,S), eval_list(Es,D,S,Vs),
     build_funenv(F,Vs,FEnv), firstlab(F,FL), at(FL,C).
```

When exiting from a function call, that is, when a command of the form ℓ : `return` e is encountered, the topmost activation frame in the execution stack is retrieved, and the caller environment is updated using the value returned by the function call. Then, program execution proceeds by popping the activation frame from the execution stack and jumping to the command which is written immediately after the function call.

```
tr(cf(cmd(L,return(E)),D,[frame(L1,X,S)|T]),
   cf(cmd(L1,C),D1,T1)) :-
      eval(E,D,S,V), update(D,T,X,V,D1,T1), at(L1,C).
```

Unlike *SS*, the *MS* semantics does not need to keep an execution stack for dealing with function calls. Indeed, when a function call is encountered, *MS* 'steps over' the function definition and makes a transition from the configuration containing the function call to the configuration containing the command which is written immediately after the function call. Since such transition can only be performed if the function call terminates, *MS* checks that there exists a sequence of multiple transitions[5] from the configuration containing the entry point of the function definition to a configuration containing either a `return` or an `abort` command occurring in the function definition. To make that check possible, the *MS* semantics requires the introduction of a `reach` predicate with two arguments that encode the source and target configurations (see clauses 8 and 9 of Table 2), while for the *SS* semantics it suffices to use a `reach` predicate with one argument only that stores the current configuration.

Indeed, for the *SS* semantics, program unsafety is specified by using the following clauses, where the predicate `reach` is unary and encodes the reachability of an error configuration, not that of a generic configuration as in the *MS* semantics.

[4] Thus, we will not show the rules for these commands and the interested reader may refer to [13].

[5] Hence, the semantics has been called *multi-step*.

```
unsafe :- initConf(C), reach(C).
reach(C) :- tr(C,C1), reach(C1).
reach(C) :- errorConf(C).
```

As a consequence of these differences, the VCs generated by our VCG strategy for the *MS* semantics are different from those generated for the *SS* semantics. In particular, in the case of the *gcd* program, the VCG strategy for the *SS* semantics (and some suitable unfolding annotations) generates the following set VC_{SS} of verification conditions:

```
50. new12(X,Y,A,B,R) :- X1=R, new3(X1,Y).
51. new11(X,Y,A,B,R) :- R1=A-B, new12(X,Y,A,B,R1).
52. new9(X,Y,A,B,R) :- Y1=R, new3(X,Y1).
53. new8(X,Y,A,B,R) :- R1=A-B, new9(X,Y,A,B,R1).
54. new7(X,Y) :- A=Y, B=X, new8(X,Y,A,B,R).
55. new6(X,Y) :- A=X, B=Y, new11(X,Y,A,B,R).
56. new4(X,Y) :- X>=Y+1, new6(X,Y).
57. new4(X,Y) :- X=<Y, new7(X,Y).
58. new3(X,Y) :- X=< -1, Y=X.
59. new3(X,Y) :- X+1=<Y, new4(X,Y).
60. new3(X,Y) :- X>=1+Y, new4(X,Y).
61. unsafe :- X>=1, Y>=1, new3(X,Y).
```

Linearity of VCs. The most evident difference between VC_{MS} and VC_{SS} concerns the form of clauses they contain. Indeed, when using *SS*, the generated VCs consist of linear Horn clauses (that is, clauses having at most one atom in their body), while those generated by using the multi-step semantics *MS* might contain nonlinear clauses (see clauses 44 and 45). Again, this is due to the fact that the predicate `tr` encoding the transition relation $\Longrightarrow$ for *MS* is defined in terms of the predicate `reach` encoding its reflexive, transitive closure $\Longrightarrow^*$. Thus, the clauses obtained at the end of the UNFOLDING phase of the VCG strategy may contain multiple `reach` atoms in their body. We will see in Section 7 that linear clauses are typically easier to analyze than nonlinear ones. Moreover, some Horn clause solvers are unable to deal with nonlinear clauses [4, 12].

Size of VCs. Another visible difference is that VC_{SS} is larger than VC_{MS}, both in the number of clauses and in the number of predicates. This is due to the fact that VC_{SS} contains distinct predicates for different calls to the `sub` function (`new8` and `new11`), thereby causing an increase of the size of the VCs, whereas in VC_{MS} a single predicate is used (`new6`).

For the small-step semantics it would be possible to derive VCs where a single predicate per function definition is introduced. However, in order to derive such VCs, during the application of the VCG strategy it is necessary to delay the introduction of the new predicate relative to the `return` commands occurring in the function definition until all the labels where to jump after executing such commands are known.

Recursive functions. If functions can be recursively defined, *MS* is probably a better choice than *SS*. Indeed, in this case the VCs for the *SS* semantics would have to use a dynamic data structure for keeping track of the execution stack, thus making the task of verifying satisfiability much harder for Horn solvers. In contrast, the multi-step semantics can easily deal with recursively defined functions and produces nonlinear VCs whose satisfiability can be checked by using SMT solvers supporting the UFLIA theory.[6]

Number of variables. The VCs occurring in VC_{MS} and VC_{SS} also differ with respect to the number of variables occurring in atoms. In particular, the number of variables in VC_{MS} is twice the number of variables in VC_{SS}. This is a consequence of the different arity of the `reach` predicate used in *MS* and *SS*. Indeed, in VC_{MS} the variables encode the identifiers occurring both in the source and target configurations of `reach` atoms, while in VC_{SS} they encode the identifiers occurring in the current configuration only.

We will see in Section 7 that the differences between the VCs automatically generated using *MS* and *SS* have an impact on the effectiveness of the Horn clause solvers we use for verifying program correctness.

6. Encoding variations of the semantics

One of the biggest advantages of a semantics-based approach to VC generation via program specialization lies in its agility, that is, its ability to rapidly adapt to changes in the semantics of the imperative programming language under consideration. For example, it might be desirable for a software verification engineer to start modeling a core fragment of the language semantics. That fragment of the semantics will be incrementally extended and refined by adding support for language features which were initially ignored.

In this section, we will see how to extend the *MS* semantics for supporting additional features and how easy it is to encode such extensions in our VC generation framework, without having to modify the VCG strategy.

Side-effect free functions. In general, functions may have side effects, that is, the value of the global variables may be altered by a function call. However, if we know that a given function is side-effect free, then we can use custom semantics rules that leave the global environment unchanged, thus generating verification conditions that are hopefully easier to verify.

Here is the rule for a function f that is side-effect free.

$(R2r_{sef})$ $\langle\!\langle \ell : x = f(e_1, \ldots, e_k),\ \langle \delta, \sigma \rangle \rangle\!\rangle \implies$
$\langle\!\langle at(nextlab(\ell)),\ update(\langle \delta, \sigma \rangle, x, [\![e]\!]\, \delta\, \sigma') \rangle\!\rangle$
if $\langle\!\langle at(firstlab(f)),\ \langle \delta, \overline{\sigma} \rangle \rangle\!\rangle \implies^* \langle\!\langle \ell_r : \texttt{return}\ e,\ \langle \delta, \sigma' \rangle \rangle\!\rangle$

If we use of this rule, instead of rule $R2r$, the number of logical variables in the VCs decreases because there is no need to encode the values of the global variables occurring in the target configuration. Let us show an example of this fact. Consider again the *gcd* program of Section 4.2. If we annotate the `sub` function as side-effect free, either manually or by using an automated analysis, the VCG strategy generates the following verification conditions:

```
62. new6(X,Y,A,B,R,A,B,R1):- R1=A-B.
63. new4(X,Y,X3,Y3):- X>=Y+1, A=X, B=Y, X2=R1,
      new6(X,Y,A,B,R,A1,B1,R1), new3(X2,Y,X3,Y3).
64. new4(X,Y,X3,Y3):- X=<Y,   A=Y, B=X, Y2=R1,
      new6(X,Y,A,B,R,A1,B1,R1), new3(X,Y2,X3,Y3).
65. new3(X,Y,X,Y):- X=Y.
66. new3(X,Y,X1,Y1):- X>=Y+1, new4(X,Y,X1,Y1).
67. new3(X,Y,X1,Y1):- X+1=<Y, new4(X,Y,X1,Y1).
68. unsafe:- X>=1, Y>=1, X1=< -1, new3(X,Y,X1,Y1).
```

Note that in clause 62 the predicate `new6`, encoding the call to the `sub` function, has two arguments less than the corresponding predicate `new6` in clause 43 above, which was obtained using rule $R2r$, instead of $R2r_{sef}$.

Undefined functions and assertions. When presenting the multi-step semantics of our language we have assumed that there exists a definition for every function that is called. Now we remove this assumption and we allow programs to call functions whose definition is unknown at verification time. In order to extend our semantics with this new feature, we have (i) to restrict the applicability of the rules $(R2a)$ and $(R2r)$ for function calls to defined functions only, and (ii) to introduce two new rules for dealing with an undefined function f_u.

$(R2a_u)$ $\langle\!\langle \ell : x = f_u(e_1, \ldots, e_k), \langle \delta, \sigma \rangle \rangle\!\rangle \implies \langle\!\langle \ell_a : \texttt{abort}, \langle \bot, \delta', \sigma' \rangle \rangle\!\rangle$

This rule considers the case where the call to f_u aborts. In this case there is a transition to an aborted configuration. Note that the environments δ' and σ' are unknown.

[6] Uninterpreted Function Symbols and Linear Integer Arithmetic.

We also assume that, for each undefined function f_u, it is given an assertion $assn(f_u)$, which denotes an over-approximation of the set of values which may be returned by f_u.[7]

$(R2r_u)$ $\langle\!\langle \ell : x = f_u(e_1, \ldots, e_k),\ \langle \delta, \sigma \rangle \rangle\!\rangle \implies$
$\langle\!\langle at(nextlab(\ell)),\ update(\langle \delta', \sigma \rangle, x, v) \rangle\!\rangle$
where $v \in assn(f_u)$.

This rule considers the case where the call to f_u returns an unknown value v satisfying the assertion on f_u. In this case the caller environment is updated by using v as the new value of variable x.

Let us now assume that the definition of the `sub` function of our *gcd* program above is unknown. We only know that `sub` returns a value x such that $x \geq 0$. If we annotate the program with this assertion, then we get the following VCs, where the atoms with predicate `new6`, encoding the calls to the `sub` function, have been replaced by the underlined constraints:

```
69. new4(X,Y,X3,Y3):- X>=Y+1, X2>=0, new3(X2,Y,X3,Y3).
70. new4(X,Y,X3,Y3):- X=<Y, Y2>=0, new3(X,Y2,X3,Y3).
71. new3(X,Y,X,Y):- X=Y.
72. new3(X,Y,X1,Y1):- X>=Y+1, new4(X,Y,X1,Y1).
73. new3(X,Y,X1,Y1):- X+1=<Y, new4(X,Y,X1,Y1).
74. unsafe:- X>=1, Y>=1, X1=< -1, new3(X,Y,X1,Y1).
```

Aborted stack traces. In case of an aborted execution, it might be desirable, for debugging purposes, to record the call stack trace containing the command labels and local environments which led to the execution of the `abort` command.

This can be accomplished by adding to the configuration an extra third component that stores the stack trace. We should also make the following changes to the rules for the `abort` command and the function call:

$(R3_{st})$ $\langle\!\langle \ell_a\!:\!\texttt{abort}, \langle \delta, \sigma \rangle, [\,] \rangle\!\rangle \Longrightarrow \langle\!\langle \ell_a\!:\!\texttt{abort}, \langle \bot, \delta, \sigma \rangle, [(\ell_a, \sigma)] \rangle\!\rangle$

$(R2a_{st})$ $\langle\!\langle \ell : x = f(e_1, \ldots, e_k), \langle \delta, \sigma \rangle, [\,] \rangle\!\rangle \implies$
$\langle\!\langle \ell_a : \texttt{abort},\ \langle \bot, \delta', \sigma \rangle, [(\ell, \sigma) | s] \rangle\!\rangle$
if $\langle\!\langle at(\mathit{firstlab}(f)),\ \langle \delta, \overline{\sigma} \rangle, [\,] \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_a : \texttt{abort},\ \langle \bot, \delta', \sigma' \rangle, s \rangle\!\rangle$
where stack traces are represented by using the familiar list notation.

Output commands. A new '`print` expr' command could be introduced for writing output. This requires adding a new output field to the data structure used for representing configurations. That field will contain a list of values (initially empty) which is left unaltered by all commands except the `print` command, which appends to it the result of the evaluation of the expression expr.

Mapping the verification conditions VCs to source code. We observe that the new definition clauses introduced by the VCG strategy contain terms that encode the labeled commands of the imperative program. Therefore, if we extend this encoding to include the line numbers where the commands occur in the source code (such as numbers n1–n10 in the *gcd* example), then we can make the VCG strategy to generate VCs where line numbers, labels, and commands explicitly occur. Thus, we have an effective way of mapping VCs to lines and commands in the source code, and vice versa.

Tuning the VCG strategy. An added value of the rule-based transformational approach to VC generation is that it gives the verification engineer fine-grained control over the shape of the VCs which can be generated. For example, as shown in the *gcd* example above, simply by choosing to annotate as non-unfoldable the atoms corresponding to if-then-else commands and function entry points, we can avoid a well-known risk of potential exponential explosion of the number of VCs, and automatically obtain an effect similar to that described in [22].

[7] Library functions usually provide some information about their specifications. For instance, the `abs` function of the GNU C Library is side-effect free and returns a value greater than zero.

In some cases, however, if the number of consecutive if-then-else commands is limited, we may allow the VCG strategy to unfold these atoms and get sets of VCs which are considerably smaller. For example, below is the set of VCs we would obtain for *gcd*:

```
75. new3(X,Y,X,Y):- X=Y.
76. new3(X,Y,X2,Y2):- X1=X-Y,X>=Y+1, new3(X1,Y,X2,Y2).
77. new3(X,Y,X2,Y2):- Y1=Y-X,X+1=<Y, new3(X,Y1,X2,Y2).
78. unsafe:- X>=1, Y>=1, X1=< -1, new3(X,Y,X1,Y1).
```

Moreover, by simply changing the unfolding annotations, we may reduce the set of unfoldable atoms to only those corresponding to 'forward' jumps (that is jumps to labels greater than the current label). The effect we obtain by doing so, is that the VCG strategy essentially introduces a definition for each program point. For instance, for the *gcd* example, we obtain the following set of VCs:

```
79. new17(X,Y,A,B,R,A,B,R).                                  %return
80. new14(X,Y,A,B,R,A1,B1,R1) :- A-B=R,                      %asgn
      new17(X,Y,A,B,R,A1,B1,R1).
81. new12(X,Y,X2,Y2) :- A=Y, B=X, R1=Y1,                     %sub
      new14(X,Y,A,B,R,A1,B1,R1),new7(X,Y1,X2,Y2).
82. new11(X,Y,X2,Y2) :- A=X, B=Y, R1=X1,                     %sub
      new14(X,Y,A,B,R,A1,B1,R1),new7(X1,Y,X2,Y2).
83. new8(X,Y,X1,Y1) :- X>=Y+1, new11(X,Y,X1,Y1).             %then
84. new8(X,Y,X1,Y1) :- X=<Y, new12(X,Y,X1,Y1).               %else
85. new7(X,Y,X1,Y1) :- X>=Y+1, new8(X,Y,X1,Y1).              %loop
86. new7(X,Y,X1,Y1) :- X+1=<Y, new8(X,Y,X1,Y1).              %loop
87. new4(X,Y,X1,Y1) :- new7(X,Y,X1,Y1).                      %main
88. unsafe :- A=< -1, B>=1, C>=1, new4(C,B,A,D).
```

7. Experimental evaluation

In this section we present the results of the experimental evaluation we have performed for assessing the viability of our semantics-based method for generating VCs. This experimental evaluation is important because the form of the VCs may have a significant impact on the efficiency and, more importantly, on the effectiveness of the tools which are then used for checking their satisfiability,

We have applied our VCG strategy for generating the VCs for several verification problems taken from the literature, using both the *SS* and the *MS* semantics. Then, we have evaluated the quality of the generated VCs by giving them as input to the following state-of-the-art Horn solvers: (i) QARMC (the Horn solver of the HSF(C) software model checking tool [23]), (ii) Z3 [15] using the PDR engine, and (iii) MSATIC3 (a version of MathSAT [4] optimized for Horn solving). In order to evaluate the efficiency of our implementation we have also run the HSF(C) tool alone on the same benchmark set.

The results of the experiments demonstrate that our method improves the overall accuracy of HSF(C) with a little increase of verification time, and, thus, it is viable in practice.

We also show the performance improvements that we have obtained by improving the implementation of our VCG strategy.

Verification problems. We have considered a benchmark set of 320 verification problems written in the C language (227 of which are safe and the remaining 93 are unsafe), taken from the benchmark sets of various software model checking tools,[8] whose size ranges from a dozen to about three thousand lines of code. The C programs of the problems we have considered and the VCs we have generated are available at `http://map.uniroma2.it/vcgen`.

Implementation. We have implemented our approach as a part of VeriMAP [12], a software model checking tool written in SICStus Prolog and based on program transformation of CLP programs. Our

[8] DAGGER (21 problems) TRACER (66 problems) and InvGen (68 problems), WHALE (7 problems) and from the TACAS Software Verification Competition (149 problems). The remaining 9 problems are taken from the literature.

		n	t_{VCG}	t'_{VCG}
Small-step	1. SS^p_o	216	159.45	159.45
	2. SS^s_o	320	1235.96	35.58
	3. SS^p_f	317	4254.71	34.71
	4. SS^s_f	320	218.57	11.25
Multi-step	5. *MS*	318	364.43	7.70

Table 3. Times (in seconds) taken for the VC generation using different language semantics and settings. The time limit is five minutes. n is the number of programs out of 320, for which the VCs were generated.

prototype implementation of the VC generator consists of three modules. (1) A front-end module, based on the C Intermediate Language (CIL) [34], that compiles the given verification problem into a set of Horn clauses (such as the clauses for the `at`, `initConf`, and `errorConf` predicates) using a custom implementation of the CIL visitor pattern. (2) A back-end module, based on VeriMAP, realizing the VCG strategy described in Section 4.1. (3) A module that translates the generated VCs to the specific input format of the solvers we have considered, that is, the constrained Horn clauses dialect of QARMC and the SMT-LIBv2 format for the Z3 and MSATIC3 solvers.

Technical resources. All experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system Ubuntu 12.10 (64 bit). A time limit of five minutes has been set for all problems.

Generating the VCs. Now we discuss the performance and the scalability of the VC generation process. In [13] we have shown that our verification framework can be effectively used both for generating the VCs and for checking their satisfiability. In the present work, besides experimenting with different formalizations of the operational semantics, we have also implemented several improvements for increasing the scalability of our method. In particular, we have replaced the use of an inefficient implementation of the constraint satisfiability test based on a projecting operator (*psat*) [26] with a more efficient one (*sat*), and we have introduced the use of the `findall` Prolog predicate for computing *FullUnf* when unfolding atoms annotated as fully unfoldable.

We report the results we have obtained in Table 3. Columns (n) and (t_{VCG}) report the total number of verification tasks for which our tool was able to generate the VCs within the time limit of five minutes, and the time taken for the generation, respectively.

Line 1 (SS^p_o) reports the results obtained when the VCG strategy uses the small-step *SS* semantics [13] with the *psat* operator (denoted by superscript p) and unfoldable atoms can only be annotated as unfoldable once (denoted by subscript o), not as fully unfoldable. Line 2 reports the results obtained by replacing the *psat* operator with the more efficient *sat* operator (denoted by superscript s). Line 3 and 4 show the results obtained by enabling the use of fully unfoldable annotations for all atoms with non-recursive predicates (denoted by subscript f) and by using *psat* and *sat*, respectively.

The best performance is obtained by the *SS* semantics by using the efficient satisfiability test and the fully unfoldable annotations (SS^s_f) allowing us to produce the VCs for the whole benchmark set in less than 4 minutes (see line 4). Note that if we use *psat* there are always timed out problems.

With regard to the multi-step *MS* semantics, in line 5 we report the results we obtained by using *sat* and fully unfoldable annotations for all `tr` atoms, except those which can be unified with the head of clauses `2a` and `2r` (see Table 2), whose body contains a `reach` atom. This restriction is needed for guaranteeing the termination of the calls to the *FullUnf* procedure, but leads to an increase of the VC generation time.[9]

Despite the increase of the VC generation time, the *MS* semantics generates more compact VCs than those obtained by the *SS* semantics. Indeed, by considering the 318 verification problems for which both SS^s_f and *MS* are able to generate VCs, the number of clauses of the VCs generated using *MS* is 34% lower than that of the VCs generated using SS^s_f.

In order to evaluate the performance improvement on the VC generation time we achieved by improving our implementation, in Column (t'_{VCG}) we report the total time required to generate the VCs on a subset of the benchmark set consisting of the 216 verification problems for which *SS* and *MS* are able to generate the VCs whichever setting is used. We note that for this subset the VC generation speedup with respect to SS^p_o reaches $14\times$ for SS^s_f and $20\times$ for *MS*.

In our experiments with different semantics we have also considered a subset of the benchmark set consisting of the SV-COMP verification tasks `systemc-transmitter`$*$ and `systemc-token_ring`$*$ (43 problems) whose size ranges from 450 LOC to 2 KLOC. On this subset the VC generation time using *MS* is always lower than the one required to generate the VCs using SS^s_f. Moreover, if we consider the hardest verification tasks in this set, namely `systemc-transmitter.16_unsafeil.c` and `systemc-token_ring.15_unsafeil.c`, the VC generation time using SS^p_o is about 38 and 43 minutes, respectively. This time drops dramatically if we generate the VCs using SS^s_f (about 8.5s, for each problem) and *MS* (about 2.5s, for each problem).

Solving the VCs (*that is, proving satisfiability of the VCs*). The results we have obtained by running the Horn solvers QARMC, Z3, and MSATIC3 on the VCs generated by our tool are reported in Table 4. Line (c) reports the total number of correct answers, which is the sum of the number of correct answers for safe and unsafe problems reported at lines (s) and (u), respectively. Line (i) reports the total number of incorrect answers, which is the sum of the number of false alarms (safe problems that have been proved unsafe) and missed bugs (unsafe problems that have been proved safe) reported at lines (f) and (m), respectively. Line (to) reports the number of problems for which the tool did not provide any conclusive answer within the time limit of five minutes. Line (n) reports the total number of problems on which the tool has been applied. Line (t_{VCG}) reports the time taken by the execution of the VCG strategy. Line (st) reports the time taken for solving the VCs, that is, proving their satisfiability. Lines (tt) and (at) report the total and average verification time, respectively. These times are computed on the (correct or incorrect) answers, excluding the time taken by problems which timed out. We have also reported in the last column the results obtained by running the HSF(C) tool alone, that is, using its own specific VC generator.

If we consider the VCs generated by applying the VCG strategy using the *SS* and *MS* semantics, QARMC provides more correct answers than Z3, MSATIC3[10] and, surprisingly, even than HSF(C).

Moreover, *SS* provides a higher precision (defined as the ratio between the number of programs which has been shown to be safe or unsafe, and the total number of programs) than *MS*. We note also that Z3 is the solver that provides the highest number of correct answers on unsafe problems.

[9] Using *MS* we are not able to generate the VCs for two problems due to a limitation in the number of arguments of compound terms in SICStus, when building the head of a new definition clause. This limitation can be easily overcome by using multiple terms or a list of terms, instead of a single term.

[10] MSATIC3 is only able to deal with Horn clauses which are linear, possibly after some preprocessing. However, in general the clauses produced by using the *MS* semantics may be nonlinear.

		Small-step(SS^s_f)			Multi-step (*MS*)			HSF(C)
		QARMC	Z3	MSATIC3	QARMC	Z3	MSATIC3	
c	Correct answers	221	209	212	212	196	178	189
s	safe problems	164	150	160	161	143	149	158
u	unsafe problems	57	59	52	51	53	29	31
i	Incorrect answers	5	0	2	3	0	0	12
f	false alarms	3	0	0	1	0	0	3
m	missed bugs	2	0	2	2	0	0	9
to	Timed-out problems	94	111	106	103	122	140	119
n	Total problems	320	320	320	318	318	318	320
t_{VCG}	VCG time	218.57	218.57	218.57	364.43	364.43	364.43	N/A
st	Solving time	3423.75	3446.84	3008.81	2924.62	2618.21	1616.81	N/A
tt	Total time	3642.32	3665.41	3227.38	3289.05	2982.64	1981.24	631.11
at	Average Time	16.12	17.54	15.08	15.30	15.30	11.13	3.14

Table 4. Verification results using QARMC, Z3, MSATIC3, and HSF(C). The time limit is five minutes. Times are in seconds.

Unfortunately, when executed on the VCs generated by the VCG strategy the Horn solvers also give some incorrect answers which are due to missed bugs. These incorrect answers could be eliminated by using a semantics that can deal with overflows during the evaluation of arithmetic expressions returning unsigned integers.

Note that QARMC provides less incorrect answers than HSF(C). Therefore, the overall correctness and precision of QARMC are improved by the use of our VCG strategy. This result demonstrates that, despite its generality, our method is also effective in practice.

If we examine line (*at*) reporting the average verification time, the best performance is achieved by HSF(C) followed by MSATIC3 (whose verification times are 3.5–5 times higher) and QARMC (about 5 times higher). (Recall that the verification times for QARMC and MSATIC3 include the times for VC generation taken by VeriMAP.)

The higher time taken by QARMC with respect to HSF(C) can be justified by the fact that it solves more verification problems whose size is substantial (up to two thousand lines of code). Indeed, if we consider the set of 170 problems for which QARMC and HSF(C) both provide an answer, the ratio between their verification times decreases to about 3.8 for *SS* and 2.9 for *MS*. In this set of problems there are eleven problems (`SVCOMP13-locks-test_locks*`) having the same structure, but different size, on which QARMC is particularly slow. If we remove these examples from the set, the ratio drops down to about 2.1 for *SS* and 1.2 for *MS*.

For QARMC, we also measured the overhead introduced by the VCG strategy, computed as the ratio between the VCG time and total time for the problems which did not time out. We found that this overhead is quite low and ranges from about 1.3% for *SS* to 7.9% for *MS*.

8. Related work and Conclusions

Constraint logic programming, or equivalently constrained Horn clause logic, has been shown to be a powerful, flexible formalism to reason about the correctness of programs [12, 13, 21, 23, 24, 27, 28, 32, 36], and as a common intermediate language for exchanging VCs between software verifiers [3, 10] to take advantage of the many special purpose solvers that are available nowadays.

In this paper we have shown that program transformation techniques, and more specifically, specialization of CLP programs, can be effectively applied for automatically generating VCs in the form of Horn clauses, starting from different CLP interpreters for the operational semantics of the programming language and for the logic in which the property of interest is specified.

Program specialization of a CLP interpreter for the small-step operational semantics of an imperative language has been initially proposed in [36]: in that approach the specialization process yields a residual CLP program on which analysis techniques based on abstract interpretation are subsequently applied. In [13] we presented a VC generation method overcoming some of the limitations of [36]: we introduced support for (non-recursive) functions, and improved scalability by encoding programs as sets of facts, instead of terms. In this paper, we have further extended the work of [13] by providing support for recursive functions and multi-step semantics.

Our approach shares the same objective of [39], where generic programming and monadic denotational semantics have been used to define a compositional method for building VC generators, which can be extended to new language features, or new languages.

A considerable effort has been placed in the area of automated VC generation, as it is evident from the many tools currently available, such as ESC/Java [5], Boogie [2], and Why3 [17]. These tools generate VCs by using Dijkstra's weakest precondition calculus.

ESC/Java generates VCs for Java programs with (user-provided) annotations. Boogie, besides using program annotations, takes advantage of abstract interpretation techniques to infer inductive invariants, and relies on front-ends that translate programs written in different languages (e.g. C, .NET) into the intermediate BoogiePL language. Similarly, the Why3 [17] verification platform generates VCs for C, Java, and Ada programs by converting them to an intermediate specification and programming language (WhyML). Similarly to Boogie, the Valigator tool [25], is able to infer loop invariants but using different techniques (symbolic summation, Gröbner basis computation, and quantifier elimination) and the strongest postcondition calculus.

The approach we have presented in this paper is able, like Boogie and Valigator, to automatically infer loop invariants. To this purpose, we can configure the specialization strategy by using suitable generalization operators [13]. Our method does not rely on a specific calculus to generate the VCs, and it is parametric with respect to the logic in which the property of interest is specified.

The generation of VCs based on theorem proving and operational semantics has been investigated in [31, 33]. In [33] the authors present a proof of concept method to prove partial correctness of programs that makes use of a small-step operational semantics. The semantics is explicitly expressed in the logic, and the VCs are generated as a by-product of the correctness proof. A related approach is described in [31], where an off-the-shelf theorem prover, and an operational semantics, are converted into a VC generator.

The design of general purpose abstract interpreters, parameterized with respect to the semantics of the programming language has been investigated in [6] and implemented in the TVLA system [30].

Finally, we would like to mention the K rewriting-based framework [38], which has been used for defining executable semantics of several programming languages (including ANSI C).

We believe that the use of transformational methods can play a key role in the development of highly parametric tools that support the verification of programs starting from the formal definition of the programming language semantics.

Acknowledgments

We thank to the anonymous referees of the conferences PPDP 2015 and CILC 2015, and we also thank John Gallagher for stimulating discussions on the subject. We acknowledge the financial support of INDAM-GNCS (Italy).

References

[1] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode Using Analysis and Transformation of Logic Programs. In *Proc. PADL '07*, LNCS 4354, pp 124–139. Springer, 2007.

[2] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. FMCO '05*, LNCS 4111, pp 364–387. Springer, 2006.

[3] N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *Proc. SMT'12*, pp 3–11, 2012.

[4] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *Proc. TACAS '13*, LNCS 7795, pp 93–107. Springer, 2013.

[5] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proc. CASSIS '04*, LNCS 3362, pp 108–128. Springer, 2005.

[6] P. Cousot. Abstract interpretation based static analysis parameterized by semantics. In *Proc. SAS '97*, LNCS 1302, pp 388–394. Springer, 2005.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. POPL '77*, pp 238–252. ACM, 1977.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL '78*, pp 84–96. ACM, 1978.

[9] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Specialization with Constrained Generalization for Software Model Checking. In *Proc. LOPSTR '12*, LNCS 7844, pp 51–70, Springer, 2013.

[10] E. De Angelis, F. Fioravanti, J. A. Navas, and M. Proietti. Verification of programs by combining iterated specialization with interpolation. In *Proc. HCVS '14*, EPTCS 169, pages 3–18, 2014.

[11] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Array Programs by Transforming Verification Conditions. In *Proc. VMCAI '14*, LNCS 8318, pp 182–202. Springer, 2014.

[12] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In *Proc. TACAS '14*, LNCS 8413, pp 568–574. Springer, 2014.

[13] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Science of Computer Programming*, 95, Part 2:149–175, 2014.

[14] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Proving Correctness of Imperative Programs by Linearizing Constrained Horn Clauses. In *Proc. Int. Conf. Logic Programming, ICLP '15*, 2015.

[15] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, LNCS 4963, pp 337–340. Springer, 2008.

[16] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.

[17] J.-C. Filliâtre and A. Paskevich. Why3-Where programs meet provers. In *Proc. ESOP '13*, LNCS 7792, pp 125–128. Springer, 2013.

[18] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In *Proc. LOPSTR '00*, LNCS 2042, pp 125–146. Springer, 2001.

[19] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. In *Theory and Practice of Logic Programming* 13(2): 175-199 2013

[20] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving reachability analysis of infinite state systems by specialization. *Fundamenta Informaticae*, 119(3-4):281–300, 2012.

[21] C. Flanagan. Automatic software model checking via constraint logic. In *Science of Computer Programming*, 50(1–3):253–270, 2004.

[22] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. *SIGPLAN*, 36(3):193–205, 2001.

[23] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier based on Horn Clauses. In *Proc. TACAS '12*, LNCS 7214, pp 549–551. Springer, 2012.

[24] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *Proc. PLDI '12*, pp 405–416. ACM, 2012.

[25] T. Henzinger, T. Hottelier, and L. Kovács. Valigator: A verification tool with bound and invariant generation. In *Proc. LPAR '08*, pp 333–342, 2008.

[26] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[27] J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded Symbolic Execution for Program Verification. In *Proc. RV '11*, LNCS 7186, pp 396–411. Springer, 2012.

[28] B. Kafle and J. P. Gallagher. Constraint Specialisation in Horn Clause Verification. In *Proc. PEPM '15*, pp 85–90. ACM, 2015.

[29] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.

[30] T. Lev-Ami, R. Manevich, and M. Sagiv. TVLA: A system for generating abstract interpreters. In *Building the Information Society*, Volume 156 of *IFIP*, pp 367–375. Springer, 2004.

[31] J. Matthews, J. Moore, S. Ray, and D. Vroon. Verification condition generation via theorem proving. In *Proc. LPAR '06*, LNCS 4246, pp 362–376. Springer, 2006.

[32] K. L. McMillan and A. Rybalchenko. Solving constrained Horn clauses using interpolation. MSR TR 2013-6, Microsoft Report, 2013.

[33] J. Moore. Inductive assertions and operational semantics. In *Proc. CHARME '03*, LNCS 2860, pp 289–303. Springer, 2003.

[34] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC '02*, LNCS 2304, pp 209–265. Springer, 2002.

[35] J. C. Peralta and J. P. Gallagher. Imperative Program Specialisation: An Approach Using CLP. In *Proc. LOPSTR'99*, LNCS 1817, pp 102–117. Springer, 2000.

[36] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In *Proc. SAS '98*, LNCS 1503, pp 246–261. Springer, 1998.

[37] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[38] G. Rosu and T. Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[39] A. van Leeuwen. Building verification condition generators by compositional extension. *ENTCS*, 191(0):73–83, 2007.