Semantics-based generation of verification conditions via program specialization[☆]

E. De Angelis[a,c,*], F. Fioravanti[a,c,*], A. Pettorossi[b,c,*], M. Proietti[c,*]

[a]*DEC, University "G. d'Annunzio" of Chieti-Pescara, Viale Pindaro 42, 65127 Pescara, Italy*
[b]*DICII, University of Rome Tor Vergata, Via del Politecnico 1, 00133 Roma, Italy*
[c]*CNR-IASI, Via dei Taurini 19, 00185 Roma, Italy*

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 generates the verification conditions by specializing, 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 additional language features. We have proved that the verification condition generation takes a number of transformation steps that is linear with respect to the size of the imperative program to be verified. Also the size of the verification conditions is linear with respect to the size of the imperative program. Besides the theoretical computational complexity analysis, we also 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.

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

1. Introduction

A well-established technique for the verification of program correctness relies on the generation of suitable *verification conditions* (VCs, for short) starting from the program code [2, 11, 29]. Verification conditions are logical formulas whose satisfiability implies program correctness, and the satisfiability check can be performed, if at all possible (because, in general, the problem of verifying program correctness is undecidable), by using special purpose theorem provers or *Satisfiability Modulo Theories* (SMT) solvers [4, 16]. Recently, *constrained Horn clauses* have been proposed as a common encoding format for software verification problems, thus facilitating the interoperability of different software verifiers, and efficient solvers have been made available for checking the satisfiability of Horn-based verification conditions [4, 10, 16, 28, 32]. The notion of a constrained Horn clause we use in this paper is basically equivalent to the notion of a *Constraint Logic Programming* (CLP) clause [33]. The choice of either terminology depends on the context of use. Constraints are assumed to be formulas of any first order theory.

[☆]This paper is an extended, improved version of [12].
[*]Corresponding author
Email addresses: `emanuele.deangelis@unich.it` (E. De Angelis), `fabio.fioravanti@unich.it` (F. Fioravanti), `adp@iasi.cnr.it` (A. Pettorossi), `proietti@iasi.cnr.it` (M. Proietti)

Typically, verification conditions are automatically generated, starting from the programs to be verified, by using *verification condition generators*. These generators are special purpose software components that implement algorithms for handling the syntax and the semantics of both the programming language in which programs are written and the class of properties to be verified. 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.

Having built a VC generator for a given programming language, to build a new VC generator for programs written in an extension of that language, or a different programming language, or even for programs written in the same language syntax but with a different language semantics (for instance, the big-step semantics, instead of the small-step semantics [52]) requires the design and the implementation of a new, ad hoc VC generator.

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

The use of CLP program specialization for analyzing programs is not novel. Peralta et al. [49] have used it for analyzing simple imperative programs and Albert et al. [1, 27] for analyzing Java bytecode. In a previous work of ours [11] VCs are generated from a small-step semantics, for verifying imperative programs using iterated specialization. Here we extend and further develop the VC generation technique based on CLP specialization, and we demonstrate its generality and flexibility by showing that suitable customizations of the CLP specialization strategy are able to effectively deal with a multi-step semantics and several variants thereof. By the term *multi-step semantics*, which is folklore, we mean a hybrid between small-step semantics and big-step semantics, where: (i) the execution of each command, different from a function call, is formalized as a one-step transition from a state to the next one, and (ii) the execution of a function call is formalized as a sequence of one-step transitions from the state where the function is called to the one where the function evaluation terminates. We also show the scalability of our technique for VC generation through both a theoretical complexity analysis and the results we have achieved by using our implementation. Finally, we show, in an empirical way, that our specialization strategy returns VCs which are of high quality, in the sense that these VCs can effectively be handled by state-of-the-art solvers for checking the satisfiability of Horn clause verification conditions [4, 16, 28, 32]. Actually, some solvers perform better on VCs generated by specialization, than on VCs generated by *ad hoc* algorithms.

Our verification method can be described as follows. Given an imperative program P and a property φ to be proved, we construct a CLP program I, which defines a nullary predicate `unsafe` such that P satisfies the property φ if and only if the atom `unsafe` is not derivable from I. The construction of the CLP program I depends on the following parameters: (i) the imperative program P, (ii) the operational semantics of the imperative language in which P is written, (iii) the property φ to be proved, and (iv) the logic that is used for specifying φ (in this case, the reachability of an unsafe state, that is, a state where φ does not hold).

The verification conditions are obtained by specializing program I with respect to its parameters. This specialization process is performed by applying semantics-preserving unfold/fold transformation rules [17]. The application of these rules is guided by a strategy particularly designed for verification condition generation, called *the VCG strategy*. Thus, the correctness of the verification conditions follows directly from the correctness of the unfold/fold transformation rules that are applied during program specialization.

When we perform the specialization of the CLP program I, we get the effect of removing from I the overhead due to the level of interpretation which is present in I because of the clauses defining the operational semantics of the imperative programming language. This specialization, called *the removal of the interpreter* in this paper, realizes the *first Futamura projection*, which is a well-known operation in the program specialization literature [35]. Indeed, by the first Futamura projection we have that the specialization of an interpreter written in a language L (CLP, in our case) with respect to a source program (written in C, in our case) has the effect of compiling the source program into L. The removal of the interpreter drastically simplifies program I by getting rid of the complex terms (including lists) that encode the commands of the imperative language and their operational semantics. Then, the simplified program, derived after the removal of the interpreter, is handled by using special purpose automatic tools for Horn clauses with linear integer arithmetic constraints [4, 16, 28, 32].

In our approach, similarly to what is done in other papers [6, 44, 48], we use a formal representation of the operational semantics of the language in which the imperative programs are written, as an explicit parameter of the verification problem. One of the most significant advantages of this technique is that it enables us to design widely applicable VC generators for programs written in different programming languages, and for different operational semantics of languages with the same syntax, by making small modifications only.

An Introductory example

Let us consider the program `sum_upto` in Figure 1. The final value of the program variable `z` is the sum of the integers from 1 to the initial value of the program variable `x` (which equals the final value of `x`, as that value never changes). The semantics of the program can be viewed as a relation between an *initial configuration*, where the global program variables `x` and `z` have values `X` and `Z`, respectively, and a *final configuration*, where `x` and `z` have values `X1` and `Z1`, respectively. Suppose we want to check the following safety property: for every computation that starts from an initial configuration where `X>=2` is true (we use Prolog syntax for constraints) and terminates, then in the final configuration `Z1>X1` is true. This property is equivalent to the Hoare triple {`x>=2`} `sum_upto` {`z>x`}. As mentioned above, our method works by introducing a CLP program I that encodes the *negation* of the safety property through the following clause:

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

where `initConf(C)` holds if `C` is an initial configuration where `X>=2` is true, `reach(C,C1)` holds if configuration `C1` can be reached from configuration `C`, and `errorConf(C1)` holds if `C1` is a final configuration where `Z1=<X1` holds. The predicate `reach` is defined in terms of predicates that encode the interpreter of the language. The CLP specialization technique presented in this paper will produce the new CLP program I_{sp} shown in Figure 1, which encodes the verification conditions for the program and property of interest. The predicates of I_{sp} correspond to some of the program points of `sum_upto`. We have indicated this correspondence in Figure 1.

By the correctness of specialization, the safety property holds for `sum_upto` *if and only if* $I_{sp} \not\models$ `unsafe`, or equivalently, $I_{sp} \cup \{\neg$ `unsafe`$\}$ is satisfiable. Thus, by using one of the state-of-the-art solvers for Horn clauses with linear integer constraints, we can attempt to check the satisfiability of the set $I_{sp} \cup \{\neg$ `unsafe`$\}$ of clauses. Now, (i) if that set is satisfiable, then the safety property holds, while (ii) if that set is unsatisfiable, then the safety property does not hold. Clearly, being the satisfiability of Horn clauses with linear integer constraints an undecidable problem, the solver may not terminate with a definite answer. Fortunately, in this example, by using the solver Z3 we get that $I_{sp} \cup \{\neg$ `unsafe`$\}$ is satisfiable, and hence the safety property holds. □

Program `sum_upto`

```
int x, z;
int f(int n) {
  int r;
  if (n <= 0)       /*new2*/
    r = 0;
  else
    r = f(n-1)+n;   /*new3*/
  return r;
}
void sum_upto() {
  z = f(x);         /*new1*/
}
```

Verification conditions I_{sp}

```
unsafe :- X>=2, Z1=<X1, new1(X,X1,Z1).
new1(X,X1,Z1) :- N=X, Z1=R, new2(N,X,X1,R).
new2(N,X,X1,R) :- N=<0, X1=X, R=0.
new2(N,X,X1,R) :- N>=1, new3(N,X,X1,R).
new3(N,X,X1,R) :- M=N-1, R=R1+N, new2(M,X,X1,R1).
```

Figure 1: The program `sum_upto` and the verification conditions ensuring the safety of the program.

The contributions of this paper can be summarized as follows.

- We have defined a multi-step operational semantics for a fragment of the C language manipulating integers and integer arrays.
- 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 program property of interest.
- We have presented two results about the computational complexity of the VCG strategy. First, we have shown that, under suitable conditions on unfolding, the VCG strategy always terminates in a number of transformation steps that is linear with respect to the size of the imperative program. Then, we have shown that the size of the generated verification conditions is linear with respect to the size of the imperative program.

- We have presented two transformation techniques aimed at reducing the number of arguments of the predicates used in the VCs. These techniques extend to the case of CLP programs analogous techniques that have been developed for logic programs [38, 51]. The first technique is a transformation strategy, called NLR (Non-Linking variable Removal), that removes variables occurring as arguments of an atom in the body of a clause and do not occur elsewhere in the clause. The second technique, called constrained FAR, is a generalization of liveness analysis, and removes arguments that are not actually used. Similarly to the case of logic programming, the reduction of predicate arity improves the time and space needed for matching atoms during satisfiability proofs. Through our experiments we show that this arity reduction is very effective in the case of large programs.
- 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 the more traditional small-step semantics. Indeed, although these two semantics are equivalent with respect to the input-output behavior of the programs, they show differences in the structure of the verification conditions that are generated and also in the subsequent ability of an automatic system to prove the program properties of interest.
- We have 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, additional language features.
- Finally, we have empirically proved the feasibility of the approach by performing an experimental evaluation. We have generated verification conditions in the cases of the multi-step semantics and the 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 ELDARICA [32], MathSAT [4], QARMC [28], and Z3 [16]. Our experiments also show that, when compared with the HSF(C) software model checker [28], which makes use of an *ad hoc* technique for generating VCs and then uses QARMC to test their satisfiability, our semantics-based approach to VC generation incurs in a relatively small increase of verification time but, interestingly enough, determines a significant improvement of accuracy over HSF(C) itself. We have also shown that if we apply the NLV and FAR techniques for removing redundant arguments, we can obtain VCs that are easier to be verified by Horn solvers.

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 effectively usable in practical software verification applications.

2. An Imperative Language and its Operational Semantics

We consider imperative programs manipulating integers and integer arrays, written in a language $\mathcal{L}$ which is a fragment of the C intermediate Language (CIL) [47]. The syntax of our imperative language $\mathcal{L}$ is shown in Table 1, where: (i) *Vars* is a set of integer variable identifiers, (ii) *AVars* is a set of integer array identifiers, (iii) *Functs* is a set of function identifiers, (iv) $\mathbb{Z}$ is the set of integers, and (v) *Labels* are non-negative integers. The language $\mathcal{L}$ is an extension of the one considered by De Angelis et al. [11]. In particular, in $\mathcal{L}$: (i) functions can be recursively defined, and (ii) there is an `abort` command which causes the abrupt termination of the execution of the program.

The *global variables* of a program are those introduced in the declarations of the program, and the *local variables* of the functions are those introduced in the declarations of the function definitions. We assume that local variables are suitably renamed so to avoid name clashes between the local and the global variables. In what follows we will feel free to say 'command', instead of 'labeled command'.

Language assumptions. We assume that: (i) every two distinct labeled commands have distinct labels, and labeled commands are linearly ordered according to the textual order of the program, (ii) the evaluation of expressions has no side effects, while the evaluation of functions may have side effects, (iii) in the `if (expr)` ℓ_1 `else` ℓ_2 commands, the labels ℓ_1 and ℓ_2 are different, and (iv) in every program there exists the definition of the function `void main()` whose first command has label ℓ_0 and whose last command is ℓ_h`:halt` and this is the only `halt` command in the program.

In our language there are neither blocks, nor structures, nor pointers. We can deal with commands of the form '`if (expr) cmd else cmd`' and '`while (expr) {cmd}`' by considering their translation in terms of `if-else` and `goto` commands. Jumps are allowed only to labeled commands which occur within the same function definition. Without loss of generality, we assume that the global variables of the program and the local variables of every function definition are not initialized, and every function definition has a unique `return` command and at most one `abort` command. For reasons of simplicity, we will consider one-dimensional arrays only.

$x, y, \ldots, i, j, \ldots$	$\in$ *Vars*	(integer variable identifiers)
$a, b, \ldots$	$\in$ *AVars*	(integer array identifiers)
$f, g, \ldots$	$\in$ *Functs*	(function identifiers)
k	$\in \mathbb{Z}$	(integer constants)
$\ell, \ell_0, \ell_1, \ldots$	$\in$ *Labels*	(labels)
type	$\in$ *Types*	(`void`, `int`, `char`, ...)
uop, *bop*	$\in$ *Ops*	(unary and binary operators: $-, +, *, =, \geq, \ldots$)

prog	::=	decl* fundef$^+$	(programs)
decl	::=	*type x*	(declarations)
fundef	::=	*type f* (decl*) { decl* lab_cmd$^+$ }	(function definitions)
lab_cmd	::=	ℓ:cmd	(labeled commands)
cmd	::=	x = expr | a[expr] = expr | $x = f$(expr*) | `goto` ℓ |	(commands)
		| `if` (expr) ℓ_1 `else` ℓ_2 | `return` expr | `abort` | `halt`	
expr	::=	k | x | *uop* expr | expr *bop* expr | a[expr]	(expressions)

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

In order to define the *multi-step* operational semantics, denoted *MS*, of our imperative language whose syntax is shown in Table 1, we need the following structures (see, for instance, [50]).

(i) A *global environment* δ which is a function that maps global variables to integers or integer arrays, and (ii) a *local environment* σ which is a function that maps the formal parameters and the local variables to integers or integer arrays. A global or local environment with domain V maps: (i) every integer variable identifier $x \in V$ to a value $v \in \mathbb{Z}$, and (ii) every array identifier $a \in V$, whose dimension is $dim(a)$, to a finite function from the set $\{0, \ldots, dim(a)-1\}$ to $\mathbb{Z}$, that is, to a sequence of $dim(a)$ integers.

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

Given any mapping $g : X \to D$, by $update(g, x, d)$, with $x \in X$ and $d \in D$, we denote the mapping g' that is equal to g, except that $g'(x) = d$. Given the mappings $g_1 : X_1 \to D$ and $g_2 : X_2 \to D$, with $X_1 \cap X_2 = \emptyset$, the pair of mappings $\langle g_1, g_2 \rangle : X_1 \cup X_2 \to D$ is defined as follows: $\langle g_1, g_2 \rangle(x) = if\ x \in X_1\ then\ g_1(x)\ else\ g_2(x)$. We extend the *update* function to act on pairs of mappings $\langle g_1, g_2 \rangle$ as follows: for any $x \in X_1 \cup X_2$, with $X_1 \cap X_2 = \emptyset$, and $d \in D$, $update(\langle g_1, g_2 \rangle, x, d) = if\ x \in X_1\ then\ \langle update(g_1, x, d), g_2 \rangle\ else\ \langle g_1, update(g_2, x, d) \rangle$.

Given a finite function $\overline{a}$ denoting an array of n elements, and given an integer i in $\{0, \ldots, n-1\}$ and an integer v, in what follows we will write $write(\overline{a}, i, v)$, instead of $update(\overline{a}, i, v)$. Thus, $write(\overline{a}, i, v)$ is a new array obtained from $\overline{a}$ by replacing the element of $\overline{a}$ at position i by v. We will use the *write* function to define the semantics of operations on arrays.

For any program P, for any label ℓ, (i) $at(\ell)$ denotes the command in P with label ℓ, and (ii) $nextlab(\ell)$ denotes the label of the command, if any, 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 $firstlab(f)$. For any expression e, any global environment δ, and any local environment σ, $[\![e]\!]\,\langle \delta\, \sigma \rangle$ denotes the value of e in $\langle \delta, \sigma \rangle$. For instance, if x is a global integer variable and $\delta(x) = \langle \delta\, \sigma \rangle(x) = 2$, then $[\![x+1]\!]\langle \delta\, \sigma \rangle = 3$.

2.1. Multi-step semantics MS

The *MS* semantics of an imperative program P is represented as a binary transition relation between configurations, denoted $\Longrightarrow$ which is defined by the following rules $R1$–$R5$. As usual, $\Longrightarrow^*$ denotes 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* of the transition relation $\Longrightarrow$ (or $\Longrightarrow^*$, respectively). In the *MS* semantics, similarly to the small-step semantics, the relation $\Longrightarrow$ formalizes the notion of 'one step of computation'. However, in the case of a function call, $\Longrightarrow$ is defined in terms of $\Longrightarrow^*$ (see rule ($R2$)), hence the name *multi-step semantics*. This semantics is different both from the small-step semantics, which defines the semantics of function calls by introducing a stack of calls, and from the big-step (or *evaluation*) semantics, which defines a relation $\Longrightarrow$ from configurations to final values [52].

($R1$) *Assignment.* If x is an integer (global or local) variable identifier:

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

If a is an integer (global or local) array identifier:

$$\langle\!\langle \ell\!:\!a[ie]\!=\!e,\ \langle\delta,\sigma\rangle\rangle\!\rangle \Longrightarrow \langle\!\langle \mathit{at}(\mathit{nextlab}(\ell)),\ \mathit{update}(\langle\delta,\sigma\rangle,\ a,\ \mathit{write}(\langle\delta,\sigma\rangle(a),\ [\![ie]\!]\langle\delta\,\sigma\rangle,\ [\![e]\!]\langle\delta\,\sigma\rangle))\rangle\!\rangle$$

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 two situations may occur: either the execution aborts (see rule ($R2a$)), or the execution 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.

($R2a$) $\langle\!\langle \ell\!:\!x\!=\!f(e_1,\ldots,e_k), \langle\delta,\sigma\rangle\rangle\!\rangle \Longrightarrow \langle\!\langle \ell_a\!: \texttt{abort}, \langle\bot,\delta',\sigma\rangle\rangle\!\rangle$

if $\langle\!\langle \mathit{at(firstlab}(f)), \langle\delta,\overline{\sigma}\rangle\rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_a\!: \texttt{abort}, \langle\bot,\delta',\sigma'\rangle\rangle\!\rangle$

($R2r$) $\langle\!\langle \ell\!:\!x\!=\!f(e_1,\ldots,e_k),\ \langle\delta,\sigma\rangle\rangle\!\rangle \Longrightarrow \langle\!\langle \mathit{at}(\mathit{nextlab}(\ell)), \mathit{update}(\langle\delta',\sigma\rangle, x, [\![e]\!]\ \langle\delta'\,\sigma'\rangle)\rangle\!\rangle$

if $\langle\!\langle \mathit{at(firstlab}(f)),\ \langle\delta,\overline{\sigma}\rangle\rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_r\!: \texttt{return}\ e,\ \langle\delta',\sigma'\rangle\rangle\!\rangle$

In these rules ($R2a$) and ($R2r$): (i) the arguments $e_1,\ldots,e_k$ are evaluated in the global and local environments of the caller and their values, say $v_1 = [\![e_1]\!]\ \langle\delta,\sigma\rangle, \ldots, v_k = [\![e_k]\!]\ \langle\delta,\sigma\rangle$, are bound to the formal parameters of the function f, and (ii) $\overline{\sigma}$ is the local environment for the evaluation of f. The environment $\overline{\sigma}$ is of the form:

$$\{\langle x_1,v_1\rangle,\ldots,\langle x_k,v_k\rangle,\ \ \langle y_1,n_1\rangle,\ldots,\langle y_h,n_h\rangle\},$$

where $n_1,\ldots,n_h$ are some values in $\mathbb{Z}$. (Recall that we assume that, when the local variables $y_1,\ldots,y_h$ are declared, they are not initialized.) Note that, since the values of $n_1,\ldots,n_h$ are left unspecified, the transition relation defined by these rules ($R2a$) and ($R2r$) is nondeterministic.

Informally, a function call either (i) aborts, if the execution of the function definition eventually leads to an aborted configuration (see Rule $R2a$), or (ii) updates the global environment using the value returned by the function call and then the computation continues by executing the command that occurs after the function call (see Rule $R2r$).

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

The `abort` command forces a transition from a regular configuration to an aborted configuration.

($R4$) *Conditional.*

If $[\![e]\!]\ \langle\delta\,\sigma\rangle\neq 0$: $\quad\langle\!\langle \ell\!: \texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2,\ \langle\delta,\sigma\rangle\rangle\!\rangle \Longrightarrow \langle\!\langle \mathit{at}(\ell_1),\ \langle\delta,\sigma\rangle\rangle\!\rangle$

If $[\![e]\!]\ \langle\delta\,\sigma\rangle = 0$: $\quad\langle\!\langle \ell\!: \texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2,\ \langle\delta,\sigma\rangle\rangle\!\rangle \Longrightarrow \langle\!\langle \mathit{at}(\ell_2),\ \langle\delta,\sigma\rangle\rangle\!\rangle$

Depending on the evaluation of the expression used in the condition, an `if-else` command follows either the 'then' branch or the 'else' branch, leaving unchanged the global environment δ and the local environment σ.

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

The `goto` ℓ' command simply makes the program execution to continue from the command with label ℓ', leaving unchanged the global environment δ and the local environment σ.

Note that rules are given neither for the `halt` command, nor the `return` commands, nor for aborted configurations, and rule ($R3$) 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 the notion of program safety and we show how to encode this notion as a CLP program. Given a program P whose global variables are $z_1,\ldots,z_r$, we define an *initial configuration* to be a triple of the form: $\langle\!\langle \ell_0\!:\!c_0,\ \langle\delta_{Init},\ \sigma_{Init}\rangle\rangle\!\rangle$, where: (i) $\ell_0\!:\!c_0$ is the first command of the function `main()` in 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} of the form: $\{\langle y_1, m_1\rangle, \ldots, \langle y_s, m_s\rangle\}$, where $y_1, \ldots, y_s$ are the local variables of the function `main()` and $m_1, \ldots, m_s$ are some given integers in $\mathbb{Z}$.

A *final configuration* is either an aborted configuration or a configuration whose labeled command is ℓ_h : `halt`. An *error configuration* is a final configuration in which an undesirable property holds, as we now specify.

Safety. We say that a program is safe when, starting from an initial configuration, it is impossible to reach an error configuration via an execution of the program.

In order to formalize this safety notion for any given program P with global variables $z_1 \ldots, z_r$, we introduce the notion of an *unsafety triple* of the form $\{\!\{Init\}\!\}\, P\, \{\!\{Err\}\!\}$, where *Init* and *Err* are formulas with free variables in $\{z_1 \ldots, z_r\}$, that denote a set $\mathbb{C}_{Init}$ of initial configurations and a set $\mathbb{C}_{Err}$ of error configurations, respectively.

We have that a configuration C is in the set $\mathbb{C}_{Init}$ iff C is an initial configuration and the global environment δ of C satisfies *Init*, that is, $Init[\delta(z_1)/z_1, \ldots, \delta(z_r)/z_r]$ holds. Likewise, we have that a configuration C is in the set $\mathbb{C}_{Err}$ iff C is a final configuration and the global environment δ of C satisfies *Err*, that is, $Err[\delta(z_1)/z_1, \ldots, \delta(z_r)/z_r]$ holds.

We say that a program P is *unsafe* with respect to *Init* and *Err*, that is, the unsafety triple $\{\!\{Init\}\!\}\, P\, \{\!\{Err\}\!\}$ holds, iff there exist a configuration C_i in $\mathbb{C}_{Init}$ and a configuration C_e in $\mathbb{C}_{Err}$ such that $C_i \Longrightarrow^* C_e$. We say that a program P is said to be *safe* iff it is not unsafe. Note that our definition of the program safety is independent of the particular values to which the global variables of the program P are initially bound.

In the next Section 3.1 we show how to encode the multi-step semantics *MS* and an unsafety triple as a CLP program.

3.1. CLP encoding of the interpreter for multi-step semantics MS

First, we recall some basic notions of Constraint Logic Programming (CLP) we need in this paper. For other notions not mentioned here the reader may refer to the book by Lloyd [42] or the paper by Jaffar and Maher [33]. In this paper we will consider constraint logic programs with linear constraints over the integers and one-dimensional integer arrays. These constraints are not standard in Prolog-based CLP systems, and for handling them we will use solvers for constrained Horn clauses such as ELDARICA [32], MathSAT [4], QARMC [28], and Z3 [16].

Atomic integer constraints are formulas of the form: $\mathtt{p}_1 = \mathtt{p}_2$, or $\mathtt{p}_1 \geq \mathtt{p}_2$, or $\mathtt{p}_1 > \mathtt{p}_2$, where $\mathtt{p}_1$ and $\mathtt{p}_2$ are linear polynomials with integer variables and coefficients. When writing polynomials the sum and the multiplication operations are denoted by + and $*$, respectively. An *integer constraint* is a conjunction of atomic integer constraints.

Atomic array constraints are constraints of the form: (i) `dim(A,N)`, denoting that `N` is the dimension of the array `A`, or (ii) `read(A,I,V)`, denoting that the `I`-th element of the array `A` has value `V`, or (iii) `write(A,I,V,B)`, denoting that the array `B` is equal to the array `A` except that the `I`-th element of B has value `V`. Indexes of arrays and elements of arrays are assumed to be integers. An *array constraint* is a conjunction of atomic array constraints. A *constraint* is either `true`, or `false`, or an integer constraint, or an array constraint, or a conjunction of constraints.

An *atom* is an atomic formula of the form $\mathtt{q}(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{m})$, where: (i) `q` is a `m`-ary predicate symbol different from =, $\geq$, >, `dim`, `read`, and `write`, and (ii) $\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{m}$ are terms constructed out of variables, constants, and function symbols different from + and $*$. Thus, for instance, the atom `q(2*X+1)` is replaced by the atom `q(Y)`, where `Y` is a new variable such that the constraint `Y=2*X+1` holds.

A CLP program is a finite set of clauses each of which is of the form `A :- c, B`, where `A` is an atom, `c` is a constraint, and `B` is a (possibly empty) conjunction of atoms. If `B` is an atom only, then '`c, B`' is said to be a *constrained atom*. As usual, in a clause `A :- c, B`, the atom `A` is called the *head* and the conjunction '`c, B`' is called the *body*. Without loss of generality, we assume that in every clause head, all occurrences of integer terms are distinct variables. For instance, the clause `p(X,X+1) :- X>0, q(X)` is written as `p(X,Y) :- Y=X+1, X>0, q(X)`. A clause `A :- c` is called a *constrained fact*. If in the clause `A :- c` the constraint `c` is `true`, then it is omitted and the resulting clause is called a *fact*. A CLP clause is said to be *linear* if it is of the form `A :- c, B`, where `B` consists of at most one atom. A CLP program is said to be *linear* if all its clauses are linear. By $vars(\varphi)$ we denote the set of all free variables in the formula φ. We extend this notation to sets of formulas so that, for instance, $vars(\{\varphi_1, \varphi_2\}) = vars(\varphi_1) \cup vars(\varphi_2)$.

Now we define the semantics of CLP programs. An $\mathcal{A}$*-interpretation* $\mathcal{D}$ is an interpretation such that:

(i) the carrier of $\mathcal{D}$ is the Herbrand universe [42] constructed out of the integers (that is, the elements of $\mathbb{Z}$), the finite sequences of integers (which provide the interpretation for arrays), and the function symbols of any (null or positive) arity, different from + and $*$,

(ii) $\mathcal{D}$ assigns to the function symbols + and $*$ the expected meaning in $\mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}$, and to the predicate symbols =, $\geq$, and > the expected meaning in $\mathbb{Z}\times\mathbb{Z}$,

(iii) for all sequences $a_0 \dots a_{n-1}$ of integers, for all integers d, $\mathtt{dim}(a_0 \dots a_{n-1}, d)$ is true in $\mathcal{D}$ iff d=n,

(iv) for all sequences $a_0 \dots a_{n-1}$ and $b_0 \dots b_{m-1}$ of integers, for all integers i and v,
$\mathtt{read}(a_0 \dots a_{n-1}, i, v)$ is true in $\mathcal{D}$ iff $0 \leq i \leq n-1$ and $v = a_i$, and
$\mathtt{write}(a_0 \dots a_{n-1}, i, v, b_0 \dots b_{m-1})$ is true in $\mathcal{D}$ iff $0 \leq i \leq n-1$ and $n = m$ and $b_i = v$ and
for $j = 0, \dots, n-1$, if $j \neq i$ then $a_j = b_j$,

(v) $\mathcal{D}$ is an Herbrand interpretation [42] for function and predicate symbols different from $+, *, =, \geq, >$, dim, read, and write.

We can identify an $\mathcal{A}$-interpretation $\mathcal{D}$ with the set of all ground atoms that are true in $\mathcal{D}$, and hence $\mathcal{A}$-interpretations are partially ordered by the set inclusion relation. Given a formula φ, if for every $\mathcal{A}$-interpretation $\mathcal{D}$, we have that φ is true in $\mathcal{D}$, then we write $\mathcal{A} \models \varphi$, and we say that *$\varphi$ is true in $\mathcal{A}$*. A constraint c is *satisfiable* iff $\mathcal{A} \models \exists(c)$, where for every formula φ, $\exists(\varphi)$ denotes the existential closure of φ. Likewise, $\forall(\varphi)$ denotes the universal closure of φ. A constraint is *unsatisfiable* iff it is not satisfiable.

The semantics of a CLP program Q is defined to be the *least $\mathcal{A}$-model* of Q, denoted $\mathcal{M}(Q)$, that is, the least $\mathcal{A}$-interpretation $\mathcal{D}$ such that every clause of Q is true in $\mathcal{D}$ [33].

For the multi-step semantics *MS*, the transition relation $\Longrightarrow$ between configurations and its reflexive, transitive closure $\Longrightarrow^*$ are encoded by the binary predicates tr and reach, respectively. These predicates, whose defining clauses are shown in Table 2 below, constitute the CLP *interpreter* for the multi-step semantics of our imperative language of Table 1. In Table 2 we have the clauses relative to: (i) assignments (clause 1), (ii) function calls (clauses 2a and 2r), (iii) aborts (clause 3), (iv) conditionals (clauses 4t and 4f), (v) jumps (clause 5), and (vi) reachability of configurations (clauses 6 and 7).

```
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,Sbar), firstlab(F,FL), at(FL,C),
        reach(cf(cmd(FL,C),(D,Sbar)), 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,Sbar), firstlab(F,FL), at(FL,C),
        reach(cf(cmd(FL,C),(D,Sbar)), 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.  reach(C,C).
7.  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 (encoding $\Rightarrow$) and reach (encoding $\Rightarrow^*$).

Configurations are represented as 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 $\perp$, 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) holds iff V is the value of the expression E in the global environment D and local environment S. The predicate eval_list extends the predicate eval to lists of expressions. The predicate beval(E,(D,S)) holds iff the value of the expression E is not 0 in the global environment D and the local environment S.

The predicate at(L,C) holds iff the command C has label L. The predicate nextlab(L,L1) holds iff L1 is the label of the command that is written in the given imperative program immediately after the command with label L. The predicate firstlab(F,L1) holds iff L1 is the label of the first command of the definition of the function F. The predicate build_funenv(F,Vs,Sbar) holds iff Sbar is the local environment needed for the execution of the body of the function F, and Vs is the list of the values of the actual parameters in the call 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 the command with label L. The predicate `update((D,S),X,V,(D1,S1))` holds iff the new global and local environments `D1` and `S1` are computed from the old global and local environments `D` and `S`, by binding the (global or local) variable `X` to the value `V`, using the function *update* acting on pairs of mappings (see Section 2).

Note that no constraint appears in Table 2. However, constraints are used for defining some predicates whose clauses are not shown, such as `eval`, `beval`, and `update`. Moreover, constraints are used in the encoding of an unsafety triple, as we now show.

3.2. CLP encoding of an unsafety triple

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

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

and also the clauses defining: (i) the predicates `tr` and `reach` that encode the interpreter (these clauses are given in Table 2), (ii) the predicates `initConf` and `errorConf` that encode the formulas *Init* and *Err*, respectively, denoting the sets of the initial and error configurations, and (iii) the predicates that encode the declarations and the function definitions of the imperative program *P* (among these clauses there are those defining the predicate `at` that encode the commands of *P*).

Since the focus of this paper is the generation of the verification conditions, we will restrict ourselves to *Init* and *Err* properties that are constraints. The interested reader may refer to a paper by De Angelis et al. [13] where it is shown how to deal with more complex *Init* and *Err* properties such as those defined by a set of recursive constrained Horn clauses.

Now to fix the ideas we give an example of an unsafety triple and its encoding via a CLP program, which we call *I*.

Let us consider the unsafety triple {{*Init*}} *gcd* {{*Err*}}, where: (i) *gcd* is the C program (shown in Column (a) of Table 3) that computes, as a final value of the variable x (and y), the greatest common divisor of the two positive integers which are the initial values of the variables x and y, respectively, (ii) *Init* is the constraint $x \geq 1 \land y \geq 1$, and (iii) *Err* is the constraint $x<0$.

The program *I* that encodes that triple, is made out of:

(i) the clauses 1–7 of Table 2 and clause 8 above,

(ii) the following clauses 9 and 10 encoding the constraints *Init* and *Err*, and

(iii) the clauses defining the predicates that encode the declarations and the function definitions of the program *gcd* (see Column (c) of Table 3).

Clauses 9 and 10 below and the clauses of Point (iii) are constructed by first translating the program *gcd* into a program written in the language $\mathcal{L}$ of Table 1. The resulting program is shown in Column (b) of Table 3. Note that in this translation the while-loop at line (n8) is replaced by using the conditional '`3: if(x!=y) 4 else 9`' and the two jumps '`6: goto 3`' and '`8: goto 3`'.

```
9. initConf(cf(cmd(3,C),[(x,X),(y,Y)],[])) :- at(3,C), X>=1, Y>=1.
10. errorConf(cf(cmd(9,C),[(x,X),(y,Y)],[])) :-at(9,C), X=<-1.
```

where: (a) the body of the clause for `initConf` is made out of the constraint *Init* and the atom `at(3,C)` which refers to the command '`3: if(x!=y) 4 else 9`' which is the first command of the `main` function (see Column (b) of Table 3), and (b) the body of the clause for `errorConf` is made out of the the constraint *Err* and the atom `at(9,C)` which refers to the command '`9: halt`' in the `main` function (see Column (b) of Table 3)

The clauses of Point (iii) defining the predicates that encode the declarations and the function definitions of the program *gcd* are shown in Column (c) of Table 3. They are constructed as follows starting from the program of Column (b) of that table.

In these clauses we have: (i) the predicate `globals` for encoding the list of identifiers of the global variables (see clause 11), (ii) the predicate `fun` for encoding function definitions (see clauses 12 and 16), and (iii) the predicate `at` for encoding labeled commands as indicated at the end of Section 3.1. In particular, clause 12 encodes the `sub` function that has two formal parameters `[a,b]`, one local variable `[r]`, and whose definition starts with the command '`1: r=a-b`' encoded by the constrained fact '`at(1,asgn(r,minus(a,b)))`' (see clause 13). Similarly, clause 15 encodes the `main` function that has no formal parameters and no local variables (hence, the two empty lists `[]`), and whose definition starts with the command '`3: if(x!=y) 4 else 9`' encoded by the constrained

	(a) the C program *gcd*	(b) *gcd* in the language $\mathcal{L}$ of Table 1	(c) the set of CLP clauses encoding *gcd*
(n1)	`int x, y;`	`int x, y;`	11. `globals([x,y]).`
(n2)	`int sub(int a, int b) {`	`int sub(int a, int b) {`	12. `fun(sub,[a,b],[r],1).`
(n3)	`  int r;`	`int r;`	
(n4)	`  r=a-b;`	`1: r=a-b;`	13. `at(1,asgn(r,minus(a,b))).`
(n5)	`  return r;`	`2: return r;`	14. `at(2,return(r)).`
(n6)	`}`	`}`	
(n7)	`void main() {`	`void main() {`	15. `fun(main,[],[],3).`
(n8)	`  while (x!=y) {`	`3: if(x!=y) 4 else 9;`	16. `at(3,ite(neq(x,y),4,9)).`
(n9)	`    if (x>y)`	`4: if(x>y) 5 else 7;`	17. `at(4,ite(gt(x,y),5,7)).`
(n10)	`      x=sub(x,y);`	`5: x=sub(x,y);`	18. `at(5,asgn(x,call(sub,[x,y]))).`
(n11)	`    } else {`	`6: goto 3;`	19. `at(6,goto(3)).`
(n12)	`      y=sub(y,x);`	`7: y=sub(y,x);`	20. `at(7,asgn(y,call(sub,[y,x]))).`
(n13)	`    }`	`8: goto 3;`	21. `at(8,goto(3)).`
(n14)	`  }`	`9: halt`	22. `at(9,halt).`
(n15)	`}`	`}`	

Table 3: The *gcd* program of the unsafety triple $\{\!\{\mathit{Init}\}\!\}$ *gcd* $\{\!\{\mathit{Err}\}\!\}$ (Column (a)), its translation into the language $\mathcal{L}$ (Column (b)), and its encoding CLP clauses (Column (c)).

fact '`at(3,ite(neq(x,y),4,9))`' (see clause 16). The first argument of the `ite` term in clause 16 is the condition `neq(x,y)` of the while-loop, and the second and the third arguments (that is, `4` and `9`) are the labels of the first commands occurring, respectively, in the 'then' and 'else' branches of the conditional. The jump commands '`6: goto 3`' and '`8: goto 3`' are encoded by the constrained facts '`at(6,goto(3))`' and '`at(8,goto(3))`' (see clauses 19 and 21, respectively).

Given any unsafety triple $\{\!\{\mathit{Init}\}\!\}$ P $\{\!\{\mathit{Err}\}\!\}$, its encoding program I constructed as indicated above is correct in the sense that the unsafety triple holds (and thus the program P is unsafe) iff the atom `unsafe` belongs to the least $\mathcal{A}$-model of I. This is a straightforward consequence of the fact that the `tr` and `reach` predicates of Table 2 are a correct encoding of the operational semantics. Thus, we have the following correctness result.

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

The proof of this theorem is similar to the proof of Theorem 1 of a paper by De Angelis et al. [11]. However, in this paper: (i) we use a slightly different representation of configurations (we do not use an execution stack for dealing with function calls), and (ii) the predicate `reach` has two arguments, instead of one argument 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), which we use for automatically generating verification conditions. From this section onwards, we consider the CLP program I that encodes an unsafety triple $\{\!\{\mathit{Init}\}\!\}$ P $\{\!\{\mathit{Err}\}\!\}$ as shown in Section 3, and we assume that the imperative program P is written in the language $\mathcal{L}$.

The VCG strategy is a specialization strategy for CLP programs. In general program specialization, or *partial evaluation*, is a program transformation technique that aims at customizing a general purpose program to a specific context of use [35, 37], thereby deriving a so called *residual program*. One prominent application of program specialization is program compilation and compiler generation via the so-called *Futamura projections*. Indeed, a compiler from a source language $\mathcal{L}_1$ to a target language $\mathcal{L}_2$ can be viewed as a program that specializes an interpreter, written in $\mathcal{L}_2$, with respect to a source program, written in $\mathcal{L}_1$. In particular, our VCG strategy can be viewed as a compiler from the imperative language $\mathcal{L}$ to CLP.

There are two main categories of program specializers:

- *online specializers*, that implement a strategy that makes decisions on what call to unfold and what call to fold (or *memoize*) on the basis of an analysis performed at specialization time; and
- *offline specializers*, that implement a two-stage strategy: (i) first a *binding time analysis* produces an annotation of the program to be specialized, which tells which call to unfold and which call to fold, and (ii) then the specializer works by using this annotation.

Usually, offline specializers are more efficient, while online specializers produce better quality residual programs. The VCG strategy can be considered as a strategy for offline specialization, as it is based on an *unfolding annotation* that is computed before the specialization is performed. We will show that VCG is very efficient, both in theory and in practice, and the quality of the VCs it generates is comparable to the one achieved by *ad hoc* VC generators.

The VCG strategy (see Figure 2 below) takes as input the CLP program I and produces as output a specialized CLP program I_{sp}, encoding a set of verification conditions, such that I_{sp} is equivalent to I with respect to the atom `unsafe`, that is, `unsafe` $\in \mathcal{M}(I)$ iff `unsafe` $\in \mathcal{M}(I_{sp})$. Thus, by Theorem 1, program P is safe with respect to *Init* and *Err* iff `unsafe` $\notin \mathcal{M}(I)$.

The VCG strategy works by performing the so-called *removal of the interpreter*, that is, by removing the overhead due to the level of interpretation which is present in the initial CLP program I because of the CLP clauses defining the operational semantics of the imperative programming language and the clauses encoding the commands of the program P. The set I_{sp} of specialized CLP clauses has the graph of predicate calls that can 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 `unsafe` $\in \mathcal{M}(I_{sp})$. However, by relying on the fact that `unsafe` $\in \mathcal{M}(I_{sp})$ iff $I_{sp} \cup \{\neg$`unsafe`$\}$ is unsatisfiable, we can prove that program P is safe (or unsafe) by showing that $I_{sp} \cup \{\neg$`unsafe`$\}$ is satisfiable (or unsatisfiable, respectively). Despite the undecidability of the verification problem in the general case, it is often the case in practice that this satisfiability check can successfully be performed by automatic tools that deal with Horn clauses with linear integer arithmetic constraints [4, 10, 16, 28]. Moreover, it turns out that checking the satisfiability of $I_{sp} \cup \{\neg$`unsafe`$\}$ is often easier than checking the satisfiability of $I \cup \{\neg$`unsafe`$\}$. This is due to the fact that when the VCG strategy specializes program I, it produces drastically simplified clauses by compiling away both the references to the commands of the program P and the references to the operational semantics of the imperative programming language.

4.1. The VCG strategy

During the application of the VCG strategy we use the following transformation rules: *unfolding*, *definition introduction*, and *folding* [17, 19]. The VCG strategy starts off by unfolding clause 8 of program I (see Section 3.2), which defines the top-level predicate `unsafe`, whose validity in the least $\mathcal{A}$-model $\mathcal{M}(I)$ establishes the unsafety of the given program P with respect to the formulas *Init* and *Err*. By unfolding the `initConf`, `errorConf`, and `reach` atoms, the VCG strategy performs a symbolic exploration of the control flow graph of the imperative program P.

Now we recall the definition of the transformation rules we use.

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

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

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

For the first execution of the assignment $SpC := Unf(C, \mathtt{A}, I)$, the VCG strategy selects the leftmost atom in the body of the input clause, that is, `initConf(C)`, so that the left argument in `reach(C,C1)` will be instantiated to the initial configuration. For the subsequent executions, the atom `A` will be the only atom in the body of clause C, as all clauses added to *InCls* have a single atom in their body.

The application of the unfolding rule during specialization is guided by an annotation of the atoms in the body of the clauses that tells us whether or not a clause should be unfolded with respect to an atom. This annotation guarantees that the Unfolding phase of the VCG strategy terminates (that is, only finite sequences of applications of the unfolding rule are generated) (see Figure 2).

The reader may refer to a paper by Leuschel and Bruynooghe [37] for a survey on related techniques which guarantee *finiteness* of unfolding. In Section 4.2, we will explain in detail how this annotation is generated. For the

```
Input: The CLP program $I$ encoding the given unsafety triple $\{\!\{Init\}\!\}\ P\ \{\!\{Err\}\!\}$.
Output: A CLP program $I_{sp}$ encoding a set of verification conditions, such that unsafe $\in \mathcal{M}(I)$ iff unsafe $\in \mathcal{M}(I_{sp})$.
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:
    $SpC := Unf(C, A, I)$   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
          $SpC := (SpC - \{D\}) \cup U$
          where: $U = Unf(D, A, I)$, if A is unfoldable once, and
                 $U = FullUnf(D, A, I)$, 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, (cf1,cf2) is a pair of terms
      representing configurations, 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 tuple of variables occurring in B, and
                       for some renaming substitution $\vartheta$, B$\vartheta$ = reach(cf1,cf2)
          then  $SpC := (SpC - \{E\}) \cup \{$ H :- e, L,newq(V)$\vartheta$,R $\}$;
          else  let $F$ be the clause:  newr(V) :- reach(cf1,cf2)
                where: newr is a predicate symbol not occurring in $I \cup Defs$, and
                       V is the tuple of variables occurring in reach(cf1,cf2);
                $InCls := InCls \cup \{F\}$;
                $Defs := Defs \cup \{F\}$;
                $SpC := (SpC - \{E\}) \cup \{$ H :- e, L, newr(V), R $\}$
      end-while;
    $InCls := InCls - \{C\}$;
    $I_{sp} := I_{sp} \cup SpC$;
  end-while;
```

Figure 2: The Verification Condition Generation (VCG) strategy.

description of the VCG strategy we only assume that 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 and an atom A which is unfoldable once, the unfolding rule derives the set $Unf(C,A,I)$ of clauses. For an atom A which is fully unfoldable, the unfolding rule derives the set $FullUnf(C,A,I)$ of clauses D recursively defined as follows: either (i) $D \in Unf(C,A,I)$ and D contains no unfoldable atom in its body, or (ii) $D \in FullUnf(D',B,I)$ for some $D' \in Unf(C,A,I)$ 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. Note that the order in which clauses and atoms in their bodies are selected for unfolding is not relevant.

At the end of the UNFOLDING phase new predicate definitions are introduced for calls to the predicate reach, by applying the following rule.

Definition Introduction Rule. A new predicate newr is introduced by the clause: newr(X) :- A, where A is an atom and the argument X of newr is a tuple of variables occurring in A. Clauses introduced by the definition introduction rule are called *definitions*.

Note that in the VCG strategy the atom A will always consist of an atom of the form reach(cf1,cf2) and X will be the tuple of all variables occurring in reach(cf1,cf2). However, in Section 6 we will present a transformation strategy, that in order to improve efficiency, aims at reducing the number of arguments of the predicates. In that strategy the tuple X is constructed by taking a *subset* of the variables of an atom A (which is not of the form reach(cf1,cf2)).

The new predicate introduced by the definition introduction rule can be viewed as a *generalization* of the constrained atom 'e, reach(cf1,cf2)' where the constraint e is replaced by the constraint true (which is implicit in the clause defining newr). For more sophisticated generalization techniques used in specialization-based verification approaches we refer to the literature [7, 8, 11, 14, 21].

Then, calls to reach, with complex arguments representing configurations, are replaced by calls to the newly introduced predicates, by applying the following folding rule.

Folding Rule. Let C: H :- e, L, B, R be a clause and D: newr(X) :- A be a (renamed apart) definition such that, for some renaming substitution ϑ: (i) B = Aϑ, and (ii) for every variable Y occurring in A and not in X, Yϑ does not occur in {H, e, L, R}. Then C is *folded* w.r.t. B by using D, thereby deriving the new clause H :- e, L, newr(X)ϑ, R.

The VCG strategy proceeds by adding the clause defining the new predicate newr to the set *InCls* to be specialized and to the set *Defs* of clauses introduced by the definition introduction rule. The strategy terminates when all clauses in *InCls* have been processed, and no new predicate definitions are added to that set because all clauses derived by unfolding (and different from constrained facts) can be folded by using clauses in *Defs*.

The correctness of the VCG strategy with respect to the least model semantics is a direct consequence of the correctness of the transformation rules [17, 20]. Indeed, we have the following result.

Theorem 2 (Correctness of the VCG strategy). *Suppose that, given the input program I, the VCG strategy terminates and upon termination it returns the CLP program I_{sp}. Then* unsafe $\in \mathcal{M}(I)$ *iff* unsafe $\in \mathcal{M}(I_{sp})$.

4.2. Termination

We will refer to the outer loop of the VCG strategy (that is, the loop with double vertical lines in Figure 2), as the *UDF-loop*. That loop consists of: (i) the Unfolding phase, and (ii) the Definition-Introduction & Folding phase. The VCG strategy may not terminate because of two reasons: (i) the non-termination of the while-loop of the Unfolding phase, and (ii) the non-termination of the UDF-loop (indeed, the termination of the while-loop of the Definition-Introduction & Folding phase is obvious because the number of clauses with the reach atom decreases).

As already mentioned, the unfolding annotation of the atoms occurring in the body of the clauses should guarantee finiteness of the Unfolding phase. First we need the following definition.

Definition 1. *We say that an unfolding annotation* guarantees finiteness *if, for every clause C such that the predicates occurring in the body of C are defined in the CLP program I, the* Unfolding *phase of the VCG strategy, with the given annotation, terminates.*

Now we define an unfolding annotation for the VCG strategy, denoted *UA*, that guarantees finiteness. The annotation *UA* also guarantees that the size of the specialized clauses generated by the VCG strategy is *linear* with respect to the size of the given clauses, and hence with respect to the size of the imperative program to be verified, that is, the number of labeled commands occurring in P (see Section 4.3).

The unfolding annotation *UA* is defined as follows:

(i) an atom whose predicate symbol is different from tr or reach is (annotated as) fully unfoldable;

(ii) an atom of the form tr(cf(LCmd,_),_) is unfoldable once, if LCmd is the term cmd(L,asgn(X,call(F,Es))) representing a function call; otherwise it is fully unfoldable;

(iii) an atom of the form reach(cf(cmd(L,Cmd),_),_) is unfoldable once, if Cmd is an assignment or a goto command, and L is neither the entry point of a function definition nor a label occurring in an ite or goto command; otherwise reach(cf(cmd(L,Cmd),_),_) is non-unfoldable.

This definition of the unfolding annotation *UA* is specific to the interpreter considered in Section 3.1, and it has been suggested by the following general principles:
(1) (*Finite unfolding*) the unfolding annotation should guarantee finiteness,
(2) (*Determinate unfolding*) the annotation should enforce that, after the first unfolding of a given predicate definition, every subsequent unfolding step derives at most one new clause (the notion of determinate unfolding has been considered in a paper by Gallagher [24] as a means for avoiding code explosion during program specialization), and
(3) (*Singular unfolding*) different variants of the same atom should not be unfolded while specializing different predicate definitions.
The above principles can be applied to design unfolding annotations in a systematic, possibly automatic, way for the interpreters of various programming language, similarly to the *Binding Time Analysis* performed before *offline specialization* [39, 40].

The following definition and lemmata are needed for the Termination Theorem 3 below.

When not presented in the text, the proofs of these and other lemmata and theorems will be found in the Appendix.

Definition 2 (Set of Definitions and Size of an Imperative Program). (i) *Let Δ be the set of new predicate definitions that are introduced during the execution of the VCG strategy.* (ii) *The size of an imperative program P written in the language $\mathcal{L}$ is the number of labeled commands occurring in P.*

Lemma 1. *The unfolding annotation UA guarantees finiteness.*

In order to show that the UDF-loop terminates, it is enough to prove that the set Δ is finite, and hence the set *InCls* will eventually become empty. The fact that Δ is finite follows from the facts that: (i) each clause introduced during the Definition-Introduction & Folding phase is of the form: `newr(V) :- reach(cf1,cf2)`, and (ii) for any given program *P*, there are finitely many pairs `(cf1,cf2)` of configurations.

The next lemma shows that the cardinality of the set Δ is linear with respect to the size of *P*.

Lemma 2. *Let Δ be the set of new predicate definitions introduced during the execution of the VCG strategy. Then every clause in Δ has the form*:
`newr(V) :- reach(cf1,cf2)`
where:
(i) `cf1` *is a configuration of the form* `cf(cmd(L1,Cmd1),(D1,S1))`;
(ii) `cmd(L1,Cmd1)` *is a labeled command in P*;
(iii) `cf2` *is a configuration of the form* `cf(cmd(L2,Cmd2),Env)`;
(iv) `Cmd2` *is either* `halt`, *or* `abort`, *or* `return(E)`, *and* `cmd(L2,abort)` *or* `cmd(L2,return(E))` *is the unique* `abort` *or* `return` *command, respectively, occurring in the definition of the function where* `L1` *occurs*;
(v) `Env` *is either* `(D2,S2)` *or* `(bot,D2,S2)`;
(vi) `D1`, `S1`, `D2`, *and* `S2` *are* (*global or local*) *environments, that is, finite functions represented as lists of the form* `[(x1,X1),(x2,X2),...]`, *where* `x1,x2,...` *are* (*global or local*) *variables of P and* `X1,X2,...` *are CLP variables*;
(vii) `(D1,S1)` *and* `(D2,S2)` *are uniquely determined, modulo CLP variable renaming, by* `L1` *and* `L2`, *respectively.*
Then, the cardinality of Δ is of the order of $O(n)$, where n is the size of P.

From Lemmata 1 and 2 we immediately get the following result.

Theorem 3 (Termination of the VCG strategy). *The VCG strategy using the unfolding annotation UA terminates on input CLP program I.*

Proof. The UDF-loop of the VCG strategy terminates because, by Lemma 2, a finite number of new definitions is introduced. The while-loop of the Unfolding phase terminates because, by Lemma 1, the unfolding annotation *UA* guarantees finiteness. Finally, the while-loop of the Definition-Introduction & Folding phase terminates because at each iteration the number of occurrences of the `reach` predicate decreases by one unit. □

4.3. Computational Complexity of Verification Condition Generation

Now we present two results concerning the time and space complexity of the VCG strategy. First, we show that the VCG strategy terminates in a number of transformation steps that is linear with respect to the size of the imperative program P. Then, we show that the size of the generated verification conditions is linear with respect to the size of P.

For reasons of simplicity, in our complexity analysis we count only the transformation steps that are: *either*
(i) the unfolding of a clause with respect to a `tr` atom or a `reach` atom (in particular, we do not count the transformation steps consisting in the unfolding of a clause with respect to atoms occurring in the definition of the operational semantics, and having different predicate symbols, such as `eval`, `update`, `nextlab`, and `at`), *or*
(ii) the introduction of a new definition, *or*
(iii) the folding of a clause.

First we show that, during the VCG strategy, by using the unfolding annotation *UA*, a linear number of unfolding steps is performed. We need the following definition.

A configuration is said to be a *return configuration* if its command is of the form: `return(E)`.

Lemma 3. *For every label* `L` *in the program P, for every final or return configuration* `cfz`*, there exists at most one clause which is unfolded w.r.t. an atom of the form* `reach(cf(cmd(L,Cmd),_),cfz)`*, during the execution of the VCG strategy.*

Next, we prove that the result of the Unfolding phase, performed according to the unfolding annotation *UA*, is a set of clauses whose size is bounded by a constant.

Definition 3 (Size of Clauses). (i) *The size of a clause C is the number $\alpha(C)$ of the atoms occurring in C.* (ii) *The size $\alpha(S)$ of a set S of clauses is the sum of the sizes of the clauses in S.*

Lemma 4. *There exists a positive integer k such that, for every clause C:* `newp(X) :- reach(cf1,cfz)`*, where* `cfz` *is a final or return configuration, the result of applying the* Unfolding *phase to C using the unfolding annotation UA is a set SpC of clauses with $\alpha(SpC) \leq k$.*

Now, we are able to show that the VCG strategy takes at most $O(n)$ transformation steps, where n is the size of the imperative program P to be verified.

Theorem 4 (Time Complexity of VCG). *Let I be the CLP program encoding a given unsafety triple $\{\!\{Init\}\!\}\ P\ \{\!\{Err\}\!\}$. The VCG strategy terminates on the input program I in $O(n)$ transformation steps, where n is the size of P.*

Proof. By Lemmata 2 and 3, $O(n)$ unfolding steps are performed during the VCG strategy. By Lemma 2, the definition introduction rule is applied $O(n)$ times. Finally, the VCG strategy applies the folding rule once for each atom occurring in the body of a clause in *SpC* at the end of the Unfolding phase, and hence, by Lemma 4, $O(n)$ folding steps are performed. □

Finally, we show that the size of the CLP program I_{sp}, which is the output of the VCG strategy, is linear with respect to the size of the program P.

Theorem 5 (Size of the Output of VCG). *Let I_{sp} be the output of the VCG strategy on the input program I. Then $\alpha(I_{sp})$ is of the order of $O(n)$, where n is the size of P.*

Proof. Suppose that the VCG strategy terminates after r iterations of the UDF-loop. Thus, the set Δ of new predicate definitions introduced by I has cardinality $|\Delta| = r$. Let $\Delta = \{C_1, \ldots, C_r\}$. By construction, $I_{sp} = \bigcup_{i=1}^{r} SpC_i$, where, for $i = 1, \ldots, r$, SpC_i is the set of clauses derived from C_i after one iteration of the UDF-loop. By Lemma 4 there exists a positive integer k (independent of I) such that the set of clauses derived by unfolding C_i using the unfolding annotation *UA* has size not larger than k. Since the folding rule replaces a single atom by another single atom, we have that $\alpha(SpC_i) \leq k$. Hence, $\alpha(I_{sp}) \leq k \cdot |\Delta|$ and, by Lemma 2, we get the thesis. □

4.4. An Example of Application of the VCG Strategy

Now we will see in action the VCG strategy of Figure 2, when given in input the CLP program *I* encoding the unsafety triple {{*Init*}} *gcd* {{*Err*}} presented in Section 3.2.

In order to generate a set of VCs for the *gcd* program, we use the unfolding annotation *UA* defined in Section 4.2. 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* = {8}.

First Iteration of the UDF-loop. By unfolding clause 8 w.r.t. the fully unfoldable atom `initConf(X)`, we get:

```
23. unsafe:- X>=1,Y>=1, reach(cf(cmd(3,ite(neq(x,y)),4,9),[(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 the command in its source configuration 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)),4,9),[(x,X),(y,Y)],[]),cf(cmd(9,halt),[(x,X1),(y,Y1)],[]))).
```

No atom in the body of clause 24 is unfoldable. 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*:

```
24. new1(X,Y,X1,Y1):-
     reach(cf(cmd(3,ite(neq(x,y)),4,9),[(x,X),(y,Y)],[]),cf(cmd(9,halt),[(x,X1),(y,Y1)],[]))).
```

where `new1` is a new predicate symbol. By folding clause 24 w.r.t. the atom `reach` using clause 24 we get:

```
25. unsafe:- X>=1,Y>=1,X1=<-1, new1(X,Y,X1,Y1).
```

Second Iteration of the UDF-loop. Now, we consider clause 24 in *InCls*, and we perform one more iteration of the UDF-loop. By unfolding clause 24 w.r.t. the atom in its body we get:

```
26. new1(X,Y,X1,Y1):- X=Y,
     reach(cf(cmd(9,halt),[(x,X),(y,Y)],[]),cf(cmd(9,halt),[(x,X1),(y,Y1)],[])).
27. new1(X,Y,X1,Y1):- X>=Y+1,
     reach(cf(cmd(4,ite(gt(x,y)),5,7),[(x,X),(y,Y)],[]),cf(cmd(9,halt),[(x,X1),(y,Y1)],[])).
28. new1(X,Y,X1,Y1):- X+1=<Y,
     reach(cf(cmd(4,ite(gt(x,y)),5,7),[(x,X),(y,Y)],[]),cf(cmd(9,halt),[(x,X1),(y,Y1)],[])).
```

Note that the symbolic evaluation of the while-loop condition `neq(x,y)` in clause 24 generates the three constraints `X=Y` (which is the exit condition of the loop), `X>=Y+1`, and `X+1=<Y` (which together make the loop condition `x!=y`) in clauses 26–28 .

We have that the atom in clause 26 is unfoldable, and hence, by reflexivity of the `reach` predicate (clause 6), we get the following constrained fact:

```
29. new1(X,Y,X,Y):- X=Y.
```

By unfolding clause 26 using clause 7, and then further unfolding this clause w.r.t. the `tr` atom in its body, we get the empty set of clauses because the predicate `tr` has no clauses defining a transition from the `halt` command.

No unfoldable atom occurs in the body of clauses 27 and 28. In order to fold clause 27 and 28 the following definition is introduced in *Defs* and added to *InCls*:

```
30. new2(X,Y,X1,Y1):-
     reach(cf(cmd(4,ite(gt(x,y)),5,7),[(x,X),(y,Y)],[]),cf(cmd(9,halt),[(x,X1),(y,Y1)],[])).
```

By folding clauses 27 and 28 using clause 30 we get:

```
31. new1(X,Y,X1,Y1):- X>=Y+1, new2(X,Y,X1,Y1).
32. new1(X,Y,X1,Y1):- X+1=<Y, new2(X,Y,X1,Y1).
```

Third Iteration of the UDF-loop. Now, we perform one more iteration of the UDF-loop starting from clause 30 in *InCls*. By unfolding the `reach` atom in clause 30 we get:

```
33. new2(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)],[])).
34. new2(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 33 and 34 correspond to the 'then' and 'else' branches of the conditional at line (n9) of the *gcd* program. No unfoldable atom occurs in the body of clauses 33 and 34 (note that the `reach` atoms are not unfoldable because the commands in their source configurations are function calls). In order to fold clause 33 the following definition is introduced in *Defs* and added to *InCls*:

```
35. new3(X,Y,X1,Y1):-
      reach(cf(cmd(5,asgn(x,call(sub,[x,y]))),[(x,X),(y,Y)],[]), cf(C,[(x,X1),(y,Y1)],[])).
```

By folding clause 33 using clause 35 we get:

```
36. new2(X,Y,X1,Y1):- X>=Y+1, new3(X,Y,X1,Y1)
```

Clause 34, corresponding to the 'else' branch, is processed in a similar way. We first introduce the following definition:

```
37. new4(X,Y,X1,Y1):-
      reach(cf(cmd(7,asgn(x,call(sub,[y,x]))),[(x,X),(y,Y)],[]), cf(C,[(x,X1),(y,Y1)],[])).
```

By folding 34 using definition 37 we get:

```
38. new2(X,Y,X1,Y1):- X=<Y, new4(X,Y,X1,Y1).
```

Fourth Iteration of the UDF-loop. Since we have introduced a new definition, namely clause 35, we start a new iteration of the UDF-loop (Clause 37 will be considered in the next iteration of the UDF-loop below). From clause 35, after some unfolding steps, we get:

```
39. new3(X,Y,X3,Y3):- 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)),4,9),[(x,X2),(y,Y1)],[]),
            cf(cmd(9,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 39 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 39 the following clause is introduced:

```
40. new5(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 39 using definition 40 we get:

```
41. new3(X,Y,X3,Y3):- A=X, B=Y, X2=R1,
      new5(X,Y,A,B,R,X1,Y1,A1,B1,R1),
      reach(cf(cmd(3,ite(neq(x,y)),4,9),[(x,X2),(y,Y1)],[]),cf(cmd(9,halt),[(x,X3),(y,Y3)],[]))).
```

In order to fold the second atom occurring in the body of clause 41 the VCG strategy does not require the introduction of any new definition. Indeed, it is possible to fold clause 41 using clause 24 in *Defs* and we get:

```
42. new3(X,Y,X3,Y3):- A=X, B=Y, X2=R1, new5(X,Y,A,B,R,X1,Y1,A1,B1,R1), new1(X2,Y1,X3,Y3).
```

Fifth Iteration of the UDF-loop. We perform one more iteration of the UDF-loop for clause 37 defining the new predicate `new4`. From clause 37, after some unfolding steps, we get:

```
43. new4(X,Y,X3,Y3):- A=Y, B=X, Y2=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)),4,9),[(x,X1),(y,Y2)],[]),
            cf(cmd(9,halt),[(x,X3),(y,Y3)],[]))).
```

In order to fold the atoms occurring in the body of clause 43 the VCG strategy does not require the introduction of any new definition. Indeed, it is possible to fold clause 43 using clauses 40 and 24 in *Defs*, and we get:

```
44. new4(X,Y,X3,Y3):- A=Y, B=X, Y2=R1, new5(X,Y,A,B,R,X1,Y1,A1,B1,R1), new1(X1,Y2,X3,Y3).
```

Sixth Iteration of the UDF-loop. We take clause 40 from *InDefs* and we start a new iteration of the UDF-loop. By unfolding clause 40 we get:

```
45. new5(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:

```
46. new5(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:

```
25. unsafe:- X>=1,Y>=1,X1=<-1, new1(X,Y,X1,Y1).
29. new1(X,Y,X,Y):- X=Y.
31. new1(X,Y,X1,Y1):- X>=Y+1, new2(X,Y,X1,Y1).
32. new1(X,Y,X1,Y1):- X+1=<Y, new2(X,Y,X1,Y1).
36. new2(X,Y,X1,Y1):- X>=Y+1, new3(X,Y,X1,Y1).
38. new2(X,Y,X1,Y1):- X=<Y, new4(X,Y,X1,Y1).
42. new3(X,Y,X3,Y3):- A=X, B=Y, X2=R1, new5(X,Y,A,B,R,X1,Y1,A1,B1,R1), new1(X2,Y1,X3,Y3).
44. new4(X,Y,X3,Y3):- A=Y, B=X, Y2=R1, new5(X,Y,A,B,R,X1,Y1,A1,B1,R1), new1(X1,Y2,X3,Y3).
46. new5(X,Y,A,B,R,X,Y,A,B,R1):- R1=A-B.
```

Now, by using either the solver ELDARICA, or MathSAT, or QARMC, or Z3, we are able to prove that $VC_{MS} \cup \{\neg$ `unsafe`$\}$ is satisfiable, and hence the *gcd* program is safe.

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 the semantics presented in a paper De Angelis et al. [11] to deal with the syntax presented in Table 1. We will also discuss the main differences between the verification conditions we obtain 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. We will not show here the rules for these commands and the interested reader may refer to the above mentioned paper by De Angelis et al. [11].

These two semantics differ in the way they deal with function calls and function `return`'s. 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, that is, 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 represented 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 $\ell\colon\ x = f(e_1,\ldots,e_k)$ 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, encoded by the following clause s1, the *SS* semantics pushes a new activation frame on top of the execution stack. The `loc_env(T,S)` predicate holds iff either (i) both `T` and `S` are empty lists, or (ii) `S` is the local environment component of the topmost activation frame in `T`.

```
s1. tr(cf(cmd(L,asgn(X,call(F,Es))),D,T),cf(cmd(FL,C),D,[frame(L1,X,FEnv)|T])):-
      firstlab(F,FL), at(FL,C), nextlab(L,L1), loc_env(T,S),
      eval_list(Es,D,S,Vs), build_funenv(F,Vs,FEnv).
```

When exiting from a function call, that is, when a command of the form $\ell\colon$ `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. Thus, the transition for a `return` command is encoded by the following clause:

```
s2. 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 transitions (hence, the semantics has been called *multi-step*) 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 6 and 7 of Table 2), while for the *SS* semantics it suffices to use a `reach` predicate that has only one argument 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 case of the *MS* semantics.

```
s3. unsafe :- initConf(C), reach(C).
s4. reach(C) :- tr(C,C1), reach(C1).
s5. 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 unsafety triple $\{\!\{x \geq 1 \wedge y \geq 1\}\!\}$ *gcd* $\{\!\{x < 0\}\!\}$, the VCG strategy for the *SS* semantics generates the following set VC_{SS} of verification conditions:

```
 ss1. unsafe :- X>=1, Y>=1, new_ss1(X,Y).
 ss2. new_ss1(X,Y) :- X>=1+Y, new_ss2(X,Y).
 ss3. new_ss1(X,Y) :- X+1=<Y, new_ss2(X,Y).
 ss4. new_ss1(X,Y) :- X=<-1, Y=X.
 ss5. new_ss2(X,Y) :- X=<Y, new_ss3(X,Y).
 ss6. new_ss2(X,Y) :- X>=Y+1, new_ss4(X,Y).
 ss7. new_ss3(X,Y) :- A=Y, B=X, new_ss5(X,Y,A,B,R).
 ss8. new_ss4(X,Y) :- A=X, B=Y, new_ss6(X,Y,A,B,R).
 ss9. new_ss5(X,Y,A,B,R) :- Y1=A-B, new_ss1(X,Y1).
ss10. new_ss6(X,Y,A,B,R) :- X1=A-B, new_ss1(X1,Y).
```

Linearity of clauses. The VCs generated by using the small-step semantics *SS* 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 might contain nonlinear clauses (see clauses 42 and 44 in Section 4.4). 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` that encodes the reflexive, transitive closure $\Longrightarrow^*$ of the relation $\Longrightarrow$. 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 8 that linear clauses are typically easier to analyze than nonlinear ones. Moreover, some Horn clause solvers are unable to deal with nonlinear clauses [4, 10].

Processing function calls. According to clause s1 each activation frame includes information about a single function call (the label where to jump after returning from the function call and the variable used for storing the value returned by the call). Hence, the new definition introduced for the entry point of a function contains information that makes it dependent on the context in which the function is called. A consequence of this fact is that such a definition cannot be used for folding all the `reach` atoms corresponding to the same entry point, that are reached from different calls to the same function. Therefore for each different function call the VCG strategy may need to repeat the specialization process corresponding to the function body.

Now let us illustrate this phenomenon by considering again the *gcd* example. The specialization of the small-step semantics *SS* introduces the following two definitions for the entry point of the function `sub` (in these definitions '...' stands for some term which is of no interest in our example here), while the specialization of the multi-step semantics *MS* produces the definition clause 40 only.

```
new_ss5(X,Y,A,B,R) :-
  reach(cf(cmd(1,asgn(r,minus(a,b))),[(x,X),(y,Y)],[frame(8,y,[(a,A),(b,B),(r,R)])|...])).
```

```
new_ss6(X,Y,A,B,R) :-
  reach(cf(cmd(1,asgn(r,minus(a,b))),[(x,X),(y,Y)],[frame(6,x,[(a,A),(b,B),(r,R)])|...])).
```

Both `reach` atoms refer to the entry point of the definition of the `sub` function. However, they encode different environments (see, in particular, the different return labels and the different variable names used for storing the value returned by each call). This difference prevents the VCG strategy from introducing a single definition for folding both atoms.

Recursive functions. If functions are recursively defined, then the specialization of *MS* generates better verification conditions than the one of *SS* in most examples. Indeed, in the presence of recursive definitions the specialization of the *SS* semantics will not be able to remove the dynamic data structure that encodes the execution stack, and this will make 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 for Horn clauses with constraints over integers and integer arrays.

Number of variables. The atoms occurring in the VCs generated when using the *MS* semantics often have more variables than those occurring in VCs generated when using the *SS* semantics. This is due to the fact that the *SS* semantics is encoded by a unary reachability relation `reach` on configurations, while the *MS* semantics is encoded by a binary reachability relation `reach` on configurations.

We will see in Section 8 that the differences between the VCs automatically generated using the *SS* and *MS* semantics have an impact on the effectiveness of the Horn clause solvers we use for proving satisfiability. Indeed, current Horn solvers are more effective at proving linear VCs, like those generated by the *SS* semantics, than at proving non-linear VCs, like those generated by the *MS* semantics.

6. Removing Redundant Arguments

It is well known that program specialization and transformation techniques often produce clauses with more arguments than those that are actually needed [25, 38, 51]. Thus, it is not surprising to observe that such a side-effect also occurs when generating VCs via program specialization. Indeed, it is often the case that some of the variables occurring in the CLP program I_{sp}, which is the output of the VCG strategy, are not actually needed to check whether or not $\texttt{unsafe} \in \mathcal{M}(I_{sp})$. Avoiding those unnecessary variables, and thus deriving predicates with smaller arity, can increase the effectiveness and the efficiency of applying Horn clause solvers. In this section we will present two transformation techniques aimed at reducing the number of variables occurring in the program I_{sp}. They are extensions to the case of CLP programs of analogous transformations of logic programs presented in other papers [38, 51].

1. *Non-Linking variable Removal Strategy* (*NLR*). We first consider a transformation strategy, called NLR (Non-Linking variable Removal) strategy, whose objective is to remove *non-linking variables*, that is, variables that occur as arguments of an atom in the body of a clause and do not occur elsewhere in the clause [51].

Definition 4 (Linking Variables). *Let C be a clause of the form* `H :- c, L, B, R`, *where* `c` *is a constraint,* `L` *and* `R` *are* (*possibly empty*) *conjunctions of atoms, and* `B` *is an atom. The set of the* linking variables *of the atom* `B` *in C, denoted* $\mathit{linkvars}(\texttt{B}, C)$, *is* $\mathit{vars}(\texttt{B}) \cap \mathit{vars}(\{\texttt{H},\texttt{c},\texttt{L},\texttt{R}\})$. *The set of the* non-linking variables *of* `B` *in C is* $\mathit{vars}(\texttt{B}) - \mathit{linkvars}(\texttt{B}, C)$.

Before presenting the NLR strategy, we show an example of its effect. Let us suppose that, by applying the VCG strategy, we get the set $P1$ of clauses in Figure 3, where the non-linking variables have been underlined. By applying the NLR strategy we will get the set $P2$ of clauses. Now, $P2$ is equivalent to $P1$ with respect to the query `unsafe`, that is, $\texttt{unsafe} \in \mathcal{M}(P1)$ iff $\texttt{unsafe} \in \mathcal{M}(P2)$. In particular, NLR replaces the predicates `newp1` and `newp2`, which are called with the non-linking variables `X2`, `Y1`, and `Y2` (see clauses 1 and 2), by the two new predicates `newp3` and `newp4`, respectively, which are called with linking variables only. Note that the removal of the two arguments `Y1` and `X2` of `newp1`, that are the non-linking variables in clause 1, determines in clause 2 the removal of the two arguments `Y1` and `X2`, that are *linking* variables of `newp2`. Thus, from `newp2` with six arguments in clause 2, by removing also the non-linking variable `Y2`, we get the predicate `newp4` with three arguments only.

*P*1: VCs obtained by VCG

```
1. unsafe:- X1>=0, Y2=<0, newp1(X1,Y1,X2,Y2).
2. newp1(X1,Y1,X2,Z2):- Z1=X1+1,
      newp2(X1,Y1,Z1,X2,Y2,Z2).
3. newp2(X1,Y1,Z1,X2,Y2,Z2):- Z1=<9, Z3=Z1+1,
      newp2(X1,Y1,Z3,X2,Y2,Z2).
4. newp2(X1,Y1,Z1,X1,Y1,Z1):- Z1>=10.
```

*P*2: VCs obtained by NLR

```
1′. unsafe:- X1>=0, Y2=<0, newp3(X1,Y2).
2′. newp3(X1,Z2):- Z1=X1+1,
          newp4(X1,Z1,Z2).
3′. newp4(X1,Z1,Z2):- Z1=<9, Z3=Z1+1,
          newp4(X1,Z3,Z2).
4′. newp4(X1,Z1,Z1):- Z1>=10.
```

Figure 3: Application of the Non-Linking variable Removal (NLP) Strategy.

The NLR strategy is similar to the VCG strategy, and now we will mention the differences between the two. The NLR strategy is obtained from the VCG strategy in Figure 2 by: (1) assuming that the Unfolding phase is performed with all atoms annotated as non-unfoldable (and thus, for each definition, only the first step of unfolding is performed), (2) replacing the Definition-Introduction & Folding phase with the Definition-Introduction of Figure 4, and (3) performing the Folding phase of Figure 5 at the end of the outermost loop of the VCG strategy (that is, at the end of the loop with double vertical lines in Figure 2), after *all* unfolding and definition introduction steps. We assume that the input of NLR is any CLP program *Prog*. To keep the notation simple, we will identify a tuple of variables with the set of variables occurring in it. The union of two tuples is constructed by erasing duplicate elements.

```
Definition-Introduction:
| while in SpC there is a clause E of the form:  H :- c, L, B, R,  such that E cannot be folded w.r.t. the atom B
|  using any clause in Defs  do
|  let F be newp(P) :- B, where newp is a predicate symbol not occurring in Prog ∪ Defs, and P = linkvars(B, E);
|    if    in Defs there is a clause D of the form  newq(Q) :- S  such that for some renaming substitution $\vartheta$, B$\vartheta$ = S
|    then | let G be newp(L) :- B, where L = P$\vartheta$ ∪ Q;
|         | Defs := (Defs − {D}) ∪ {G};   InCls := (InCls − {D}) ∪ {G};
|    else | Defs := Defs ∪ {F};
|         | InCls := InCls ∪ {F};
| end-while;
```

Figure 4: The Definition-Introduction phase.

```
Folding:
| while in SpC there is a clause E of the form  H :- c, L, B, R,  and in Defs there is a clause D of the form
|  newp(P) :- B (modulo variable renaming), and E can be folded w.r.t. B by using D  do
|     SpC := (SpC − {E}) ∪ {H :- c, L, newp(P), R};
| end-while;
```

Figure 5: The Folding phase.

The peculiarity of the NLR strategy lies in the careful treatment of the set of variables occurring in the head of the definition clauses during the Definition-Introduction phase.

Let *E* be a clause in *SpC* of the form: `H :- c, L, B, R`, where the predicate symbol of `B` occurs in *Prog*. If *E* cannot be folded with respect to the atom `B` using any clause in *Defs*, then we have to introduce a new definition clause as we now explain.

First, we consider a definition *F* whose head contains only the linking variables of the atom `B` in the clause *E*. Let *F* be `newp(P) :- B`, where `newp` is a predicate symbol not occurring in the set *Prog* ∪ *Defs*, and `P` is the set *linkvars* (`B`, *E*) of the linking variables of `B` in *E*.

If the set *Defs* contains a clause D of the form: `newq(Q) :- S` such that, for some renaming substitution ϑ, $\mathtt{B}\vartheta = \mathtt{S}$, then we replace clause D in *Defs* with the clause `newp(L) :- B`, where $\mathtt{L} = \mathtt{P}\vartheta \cup \mathtt{Q}$. Otherwise, we introduce the definition clause F and we add it to *Defs*.

The introduction of the definition F might seem to be the best choice in the sense that it contains exactly the head variables which are actually needed for folding clause E. However, (variants of) `B` may occur also in some other clauses to be folded. Thus, if we introduce definitions whose heads contain only the linking variables, we run the risk of introducing several definitions with the same atom in the body and different sets of variables in the head (modulo variable renaming).

In order to keep the number of definitions as low as possible (and this often enhances the ability of proving program correctness), instead of introducing multiple definitions containing the same atom in the body, by applying the NLR strategy, we merge them in a single definition whose set of head variables is the union of the head variables occurring in the merged definitions (modulo variable renaming).

Theorem 6 (Termination, Correctness, and Size of the Output of NLR). *Given any CLP program Prog, the NLR strategy terminates and produces a CLP program Prog′ such that:* (i) `unsafe` $\in M(Prog)$ *iff* `unsafe` $\in M(Prog')$, *and* (ii) $\alpha(Prog') \leq \alpha(Prog)$.

2. *Constrained FAR Algorithm* (*cFAR*). Now we present an extension to constraint logic programs of the FAR algorithm propoed by Leuschel and Sørensen [38] for removing redundant arguments from logic programs. This extension will be called constrained FAR algorithm, or cFAR, for short. The objective of the FAR algorithm is to remove arguments that are not actually used during any computation of the program at hand. Indeed, it has been shown by Henriksen and Gallagher [30] that the FAR algorithm (and thus, also the cFAR algorithm) can be seen as a generalization of the liveness analysis.

Below we show the effect of applying the cFAR algorithm to the CLP program $P2$ obtained by the NLR strategy (see Figure 3). The output of the algorithm is the CLP program $P3$. Note that in program $P3$ the predicate symbol `newp4` denotes a different relation with respect to the one in program $P2$, because in $P3$ it has arity 2 and not 3.

$P2$: VCs obtained by NLR

```
1′. unsafe:- X1>=0, Y2=<0, newp3(X1,Y2).
2′. newp3(X1,Z2):- Z1=X1+1, newp4(X1,Z1,Z2).
3′. newp4(X1,Z1,Z2):- Z1=<9, Z3=Z1+1,
           newp4(X1,Z3,Z2).
4′. newp4(X1,Z1,Z1):- Z1>=10.
```

$P3$: VCs obtained by cFAR

```
1″. unsafe:- X1>=0, Y2=<0, newp3(X1,Y2).
2″. newp3(X1,Z2):- Z1=X1+1, newp4(Z1,Z2).
3″. newp4(Z1,Z2):- Z1=<9, Z3=Z1+1,
         newp4(Z3,Z2).
4″. newp4(Z1,Z1):- Z1>=10.
```

Figure 6: Application of the constrained FAR (cFAR) algorithm.

In order to define the cFAR algorithm we need to introduce some preliminary notions, some of which have been adapted from the above cited paper by Leuschel and Sørensen [38].

Definition 5 (Erasure, Erased Atom, Erased Clause, Erased Program). (i) *An* erasure *is a set of pairs each of which is of the form* $(\mathtt{p}, k)$*, where* `p` *is a predicate symbol of arity n and* $1 \leq k \leq n$.
(ii) *Given an erasure E and an atom* `A` *whose predicate symbol is* `p`*, the* erased atom $\mathtt{A}|_E$ *is obtained by dropping all the arguments that occur at position k, for some* $(\mathtt{p}, k) \in E$.
(iii) *Given an erasure E and a clause C* (*respectively, a CLP program Prog*)*, the* erased clause $C|_E$ (*respectively, the* erased program $Prog|_E$) *is obtained by replacing all atoms* `A` *in C* (*respectively, in Prog*) *by* $\mathtt{A}|_E$.

In order to avoid the risk of collisions between predicate symbols after erasing some arguments, we assume that *Prog* does not contain identical predicate symbols with different arity.

Obviously, we are interested in removing redundant arguments without altering the semantics of the original program, in the sense captured by the following definition.

Definition 6 (Correctness of Erasure). *An erasure E is correct for a program Prog if, for all atoms* `A`*, we have that*: `A` $\in \mathcal{M}(Prog)$ *iff* `A`$|_E \in \mathcal{M}(Prog|_E)$.

Since we are dealing with constraint logic programs, the notion of multiple occurrences of a variable which is used in the original formulation of FAR [38], needs to be generalized as follows.

We assume that variables occurring in atomic constraints are distinct.

Definition 7 (Variable Constrained to Another Variable). *Given two distinct variables* `X` *and* `Y` *and a constraint* `c` *of the form* $\mathtt{c}_1 \wedge \ldots \wedge \mathtt{c}_h$*, where the* $\mathtt{c}_i$*'s are atomic constraints, we say that* `X` *is* constrained to `Y` (in `c`) *if there exists* $\mathtt{c}_j$*, with* $1 \leq j \leq h$*, such that either* (i) {`X`, `Y`} $\subseteq$ *vars*($\mathtt{c}_j$)*, or* (ii) *there exists a variable* `Z` *such that* (ii.1) {`X`, `Z`} $\subseteq$ *vars*($\mathtt{c}_j$) *and* (ii.2) `Z` *is constrained to* `Y` (*in* `c`).

Now we are ready to introduce the notion of *safe erasure* that will be used during the application of the constrained FAR algorithm.

Definition 8 (Safe Erasure). *Given a program Prog, an erasure E is a* safe erasure *if, for all* $(\mathtt{p}, k) \in E$ *and clauses* `H :- c,G` *in Prog, where* `H` *is of the form* `p(X1,...,Xn)` *and* `c` *is a constraint, we have that*: (i) `Xk` *is a variable in* {`X1,...,Xn`} *and* $\mathcal{A} \models \forall$`Xk.`$\exists$`Y1,...,Ym.c`*, with* {`Y1,...,Ym`} = *vars*(`c`) − {`Xk`}, (ii) `Xk` *is not constrained* (*in* `c`) *to any other variable occurring in* `H`*, and* (iii) `Xk` *is not constrained* (*in* `c`) *to any variable occurring in* `G`$|_E$.

By a proof similar to the one by Leuschel and Sørensen [38], it can be shown that if an erasure E is safe, then it is also correct.

The cFAR algorithm takes as input a CLP program *Prog*, computes a safe erasure E, and produces as output the program $Prog|_E$. The algorithm starts off by initializing the current erasure E to the *full erasure*, that is, the set of all pairs $(\mathtt{p}, k)$, where p is a predicate of arity n occurring in *Prog* and $1 \leq k \leq n$. Then, while E contains a pair $(\mathtt{p}, k)$ such that one of the conditions of Definition 8 is not satisfied, the pair $(\mathtt{p}, k)$ is removed from E. The algorithm terminates when it is no longer possible to remove a pair $(\mathtt{p}, k)$ from E, and thus E is a safe erasure.

The cFAR algorithm terminates and preserves the semantics and the size of the input program, as stated by the following theorem.

Theorem 7 (Termination, Correctness, and Size of the Output of cFAR). *Given any CLP program Prog, the cFAR algorithm terminates and produces a CLP program $Prog|_E$ such that* `unsafe` $\in \mathcal{M}(Prog)$ *iff* `unsafe` $\in \mathcal{M}(Prog|_E)$ *and* $\alpha(Prog) = \alpha(Prog|_E)$.

Finally, we would like to note that, even if the objectives of the NLR strategy and cFAR algorithm are similar, they work in a different way. While cFAR is goal independent, NLR starts from the predicate `unsafe` and proceeds by unfolding in a goal directed fashion, similarly to *redundant argument filtering* [38]. It can be shown that, in general, the NLR strategy and the cFAR algorithm have incomparable effects.

7. 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 call to f that is side-effect free.

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

If we use 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 4.4. If we annotate (either manually or by using an automated analysis) the `sub` function as side-effect free, then the VCG strategy generates a set of verification conditions which is identical to the set VC_{MS} of verification conditions obtained at the end of Section 4.4, except for the clauses 42, 44, and 46 defining the predicates `new3`, `new4`, and `new5`, respectively, which have to be replaced by the following ones:

42_{sef}. `new3(X,Y,X3,Y3):- A=X, B=Y, X2=R1, new5(X,Y,A,B,R,A1,B1,R1), new1(X2,Y1,X3,Y3).`
44_{sef}. `new4(X,Y,X3,Y3):- A=Y, B=X, Y2=R1, new5(X,Y,A,B,R,A1,B1,R1), new1(X1,Y2,X3,Y3).`
46_{sef}. `new5(X,Y,A,B,R,A,B,R1):- R1=A-B.`

Note that in clause 46_{sef} the predicate `new5`, encoding the body of the `sub` function, has two arguments less than the corresponding predicate `new5` in clause 46, which was obtained using rule $R2r$, instead of $R2r_{sef}$.

We observe that the same effect can also be obtained by applying the NLR strategy to the program VC_{MS}. However, if the information about the side effect freeness is already available, the use of a custom semantics rule for side effect free function calls allows us to avoid performing additional transformations.

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 (for instance, library functions or functions defined by external modules). In order to extend our semantics with this new feature, we should: (i) restrict the applicability of the rules ($R2a$) and ($R2r$) for function calls to defined functions only, and (ii) introduce the following two new rules ($R2a_u$) and ($R2r_u$) for dealing with an undefined function f_u.

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

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.

We also assume that, for each undefined function f_u, we are given an assertion $assn(f_u)$, which denotes an over-approximation of the set of values which may be returned by f_u.[1] The environment δ' is unknown.

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

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 of Section 4.4 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 a set of VCs which is identical to the set VC_{MS} except that: (i) the predicate `new5` is not defined (and thus clause 46 is erased), and (ii) clauses 42 and 44 are replaced by the following clauses 42_u and 44_u:

42_u. `new3(X,Y,X3,Y3):- A=X, B=Y,` $\underline{\texttt{X2>=0}}$`, new1(X2,Y1,X3,Y3).`
44_u. `new4(X,Y,X3,Y3):- A=Y, B=X,` $\underline{\texttt{Y2>=0}}$`, new1(X1,Y2,X3,Y3).`

In this replacement the atoms of clauses 42 and 44 with predicate `new5`, encoding the calls to the `sub` function, together with the constraints binding the return values to variables of the calling contexts, have been substituted by the underlined constraints.

[1] 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.

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 the local environments which led to the execution of the `abort` command. This can be done 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 (stack traces are represented by using the familiar list notation):

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

$(R2a_{st})$ $\quad \langle\!\langle \ell\!: x = f(e_1, \ldots, e_k), \langle \delta, \sigma \rangle, [\,] \rangle\!\rangle \implies \langle\!\langle \ell_a\!: \texttt{abort},\ \langle \bot, \delta', \sigma \rangle, [(\ell, \sigma) | s] \rangle\!\rangle$
if $\langle\!\langle \mathit{at}(\mathit{firstlab}(f)),\ \langle \delta, \overline{\sigma} \rangle, [\,] \rangle\!\rangle \implies^* \langle\!\langle \ell_a\!: \texttt{abort},\ \langle \bot, \delta', \sigma' \rangle, s \rangle\!\rangle$

Tuning the VCG strategy. An additional value of the rule-based transformational approach to VC generation is that it gives to the verification engineer a fine-grained control over the shape of the VCs which can be generated. For example, as shown in the *gcd* example above, by using the unfolding annotation *UA* presented in Section 4.2 we are guaranteed that the size of the VCs is linear with respect to the size of the imperative program. Thus, we avoid a well-known risk of potential exponential explosion of the number of VCs, and automatically obtain an effect similar to that described by Flanagan and Saxe [23].

In some situations, however, it could be advantageous to use different unfolding annotations. For example, by enlarging the set of `reach` atoms that are annotated as *unfoldable once*, we could derive the following set of VCs for *gcd* example which is considerably smaller than VC_{MS}:

```
a1. unsafe:- X>=1, Y>=1, X1=<-1, new6(X,Y,X1,Y1).
a2. new6(X,Y,X2,Y2):- Y1=Y-X, X+1=<Y, new6(X,Y1,X2,Y2).
a3. new6(X,Y,X2,Y2):- X1=X-Y, X>=Y+1, new6(X1,Y,X2,Y2).
a4. new6(X,Y,X,Y):- X=Y.
```

Of course, care must be taken to ensure that the chosen unfolding annotation still guarantees the termination of the VCG strategy.

Conversely, if we reduce the set of atoms that are annotated as *unfoldable*, the termination of the VCG strategy is always guaranteed, but more definitions are introduced and, consequently, the set of VCs tends to grow. For example, we may tune the unfolding annotation so that the VCG strategy introduces a definition for each program point. For the *gcd* example, we obtain the following set of VCs:

```
 b1. unsafe :- X>=1, Y>=1, X1=<-1, new7(X,Y,X1,Y1).
 b2. new7(X,Y,X1,Y1):- new8(X,Y,X1,Y1).                                             %main
 b3. new8(X,Y,X1,Y1):- X+1=<Y, new9(X,Y,X1,Y1).                                     %loop
 b4. new8(X,Y,X1,Y1):- X>=Y+1, new9(X,Y,X1,Y1).                                     %loop
 b5. new8(X,Y,X,Y):- X=Y.                                                           %loop
 b6. new9(X,Y,X1,Y1):- X=<Y, new10(X,Y,X1,Y1).                                      %else
 b7. new9(X,Y,X1,Y1):- X>=Y+1, new11(X,Y,X1,Y1).                                    %then
 b8. new11(X,Y,X2,Y2):-A=X,B=Y,R1=X1, new12(X,Y,A,B,R,A1,B1,R1), new8(X1,Y,X2,Y2).  %sub
 b9. new10(X,Y,X2,Y2):-A=Y,B=X,R1=Y1, new12(X,Y,A,B,R,A1,B1,R1), new8(X,Y1,X2,Y2).  %sub
b10. new12(X,Y,A,B,R,A1,B1,R1):- A-B=R, new13(X,Y,A,B,R,A1,B1,R1).                  %asgn
b11. new13(X,Y,A,B,R,A,B,R).                                                        %return
```

8. 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 the satisfiability of the VCs.

We have applied our VCG strategy for generating the VCs for several verification problems taken from the literature, using both the *SS* semantics 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 [28]), (ii) Z3 [16] using the PDR engine, (iii) MSATIC3 (a version of MathSAT [4] optimized for Horn solving), and (iv) ELDARICA [32]. 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,[2] 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 [10], a software model checking tool written in SICStus Prolog and based on program transformation of CLP programs. Our prototype implementation of the VC generator consists of three modules. (1) A front-end module, based on the C Intermediate Language (CIL) [47], 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 ELDARICA and the SMT-LIBv2 format for the Z3 and MSATIC3 solvers.

Technical resources. The experiments have been performed using GNU Parallels [54] on 24 to 32 logical cores of an Intel Xeon CPU E5-2640 2.00GHz processor with 64GB of memory under the GNU Linux operating system CentOS 7 (64 bit). Timings are computed as if the experiments were run sequentially. A time limit of five minutes has been set for all problems. (The experimental settings are slightly different from those used in a previous work of ours [12].)

Generating the VCs. Now we discuss the performance and the scalability of the VC generation process. In a previous paper [11] we have shown that our verification framework can be effectively used to generate the VCs from a small-step semantics for a subset of the language presented in Table 1. In the present work, besides experimenting with different formalizations of the operational semantics and different unfolding strategies, we have also implemented several optimizations for increasing the scalability of our method. In particular, we have introduced more efficient procedures for: (i) checking the satisfiability of constraints, and (ii) computing the set *FullUnf* for atoms annotated as fully unfoldable. Regarding Point (i), the specialization strategy presented in De Angelis et al. [11] makes use of the *psat* operator, which checks the satisfiability of a constraint and projects it over a given set of variables. However, since projection is not needed when applying the unfolding rule, we have implemented a more efficient operator, called *sat*, which only performs the satisfiability check. Regarding Point (ii), we have implemented the full unfolding of an atom A simply by evaluating the query A via the `findall` Prolog predicate and collecting all the answers, hence avoiding the computational overhead due to repeated applications of the meta-level unfolding operation.

We report the results we have obtained in Table 4.

		n	t_{VCG}	t_{VCG}^{216}
Small-step	1. SS_o^p	216	180.43	180.43
	2. SS_o^s	320	1215.62	38.66
	3. SS_f^p	317	4475.19	40.67
	4. SS_f^s	320	221.68	15.17
Multi-step	5. MS	320	141.85	10.24

Table 4: 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.

[2]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.

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 [11] 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 of the *SS* semantics is obtained (see line 4) 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. 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).

The VC generation process using the *MS* semantics is faster than using the *SS* semantics. Moreover, the VCs generated using the *MS* semantics are more compact than those obtained by the *SS* semantics. Indeed, the number of clauses of the VCs generated using *MS* is about the 37% lower than that of the VCs generated using SS^s_f. (The number of the clauses of the VCs is not shown in Table 4.)

In order to compare the VC generation times obtained by varying the version of the semantics and the settings used, we consider the subset of the benchmark consisting of the 216 verification problems for which the specializer is able to generate the VCs (within the time out) whichever semantics and setting is used. In Column (t^{216}_{VCG}) we report the total time required to generate the VCs on this subset of the benchmark set.

We note that for this subset the VC generation speedup with respect to SS^p_o reaches 12× for SS^s_f and 17.5× 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 40 minutes, for each problem. This time drops dramatically if we generate the VCs using SS^s_f (about 8s, for each problem) and *MS* (about 3.5s, 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, MSATIC3, and ELDARICA on the VCs generated by our tool are reported in Table 5.

		Small-step (SS^s_f)				Multi-step (*MS*)				HSF(C)
		QARMC	Z3	MSAT	ELD	QARMC	Z3	MSAT	ELD	
c	Correct answers	217	208	205	217	210	196	177	182	189
s	safe problems	161	150	158	158	160	144	147	141	158
u	unsafe problems	56	58	47	59	50	52	30	41	31
i	Incorrect answers	5	0	3	2	3	0	1	0	12
f	false alarms	3	0	1	0	1	0	1	0	3
m	missed bugs	2	0	2	2	2	0	0	0	9
to	Timeouts	98	112	112	101	120	124	142	138	119
n	Total problems	320	320	320	320	320	320	320	320	320
t_{VCG}	VCG time	221.68	221.68	221.68	221.68	141.85	141.85	141.85	141.85	N/A
st	Solving time	3656.24	4221.39	2988.86	8809.58	2674.00	2704.95	1896.96	2779.18	N/A
tt	Total time	3877.92	4443.07	3210.54	9031.26	2815.85	2846.80	2038.81	2921.03	631.11
at	Average Time	17.87	21.36	15.66	41.62	13.41	14.52	11.52	16.05	3.14

Table 5: Verification results using QARMC, Z3, MSATIC3 (MSAT, for short), ELDARICA (ELD, for short), and HSF(C). The time limit is five minutes. Times are in seconds.

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 (or unsatisfiability). 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[3].

If we consider the VCs generated by applying the VCG strategy using the *SS* semantics, QARMC and ELDARICA provide the highest number of correct answers, while if we consider the VCs generated by using the *MS* semantics, QARMC provides more correct answers than Z3, ELDARICA, MSATIC3[4] and, surprisingly, even than HSF(C).

Moreover, the *SS* semantics 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 the *MS* semantics. We note also that Z3 and ELDARICA provide the highest number of correct answers on unsafe problems.

Unfortunately, when executed on the VCs generated by the VCG strategy, most Horn solvers also give some incorrect answers which are due to missed bugs.

The incorrect answers are due to the fact that we have considered an idealized semantics for C expressions, which are viewed as expressions in the theory of integer arithmetics. This idealized semantics does not model correctly the overflows that may occur during the evaluation of C expressions. Indeed, the missed bugs of Table 5 are due to unsigned integer expressions occurring in the conditions of if-then-else commands that, when evaluated in the theory of integer arithmetics, are unsatisfiable and make one branch of the conditional unfeasible. This idealized semantics is often adopted by verifiers, such as HSF(C), that do not focus on the problem of handling overflows, and for a fair comparison among the verifiers, we have considered the same idealized semantics. However, our approach is parametric with respect to the constraint solvers used during specialization, and we can define a semantics that agrees with the standard behavior of C arithmetic expressions by using, for instance, a solver that supports modular integer arithmetics (one such solver is available in the Z3 system).

If we examine line (*at*) reporting the average verification time, the best performance is achieved by HSF(C) followed by MSATIC3 and QARMC (whose verification times are 3.6–5.7 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 large (up to two thousand lines of code). Indeed, if we consider, for example, the set of 190 problems for which an answer (either correct or incorrect) is provided by both HSF(C) and QARMC (on the VCs generated by using the *MS* semantics), the ratio between their verification times decreases from 4.46 to about 3.29. 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 1.29.

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 5.7% for the *SS* semantics to 5% for the *MS* semantics.

Improving effectiveness of solving. In Table 6 we show the results obtained by using Z3 after the application of the auxiliary transformations realized by the NLR strategy and the cFAR algorithm presented in Section 6. Those transformations are applied only to the verification conditions for which Z3 was not able to provide an answer. In particular, column 'VCG ; Z3' reports the results we get by applying the VCG strategy for the *MS* semantics, and

[3]For technical reasons we were only able to run HSF(C) on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system Ubuntu 12.10 (64 bit). Note that, however, the power of the cores of this processor is comparable to the power of the cores of the processor used for running the other experiments.

[4]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.

		VCG ; Z3	VCG ; NLR ; Z3	VCG ; NLR ; cFAR ; Z3
c	Correct answers	196	7	9
s	safe problems	144	3	7
u	unsafe problems	52	4	2
to	Timeouts	124	117	108
n	Total problems	320	124	117
t_{VCG}	VCG time	40.65	20.48	4.57
t_{NLR}	NLR time	–	58.39	9.53
t_{cFAR}	cFAR time	–	–	304.84
st	Solving time	2704.95	988.15	649.56
tt	Total time	2745.60	1067.02	968.50
at	Average time	14.01	152.43	107.61

Table 6: Verification results obtained for the *MS* semantics by applying the VCG strategy, possibly followed by the NLR strategy and the cFAR algorithm, and then by the Z3 solver. The time limit is five minutes. Times are in seconds.

then by applying Z3 on the VCs generated. Column 'VCG ; NLR ; Z3' reports the results we get after the application of the NLR strategy on the VCs obtained by applying the VCG strategy. Column 'VCG ; NLR ; cFAR ; Z3' reports the results we get after the application of the cFAR algorithm on the VCs obtained by applying the NLR strategy. Lines (t_{VCG}), (t_{NLR}), and (t_{cFAR}) report the time taken by the execution of the VCG, NLR, and cFAR transformations on the correct answers, respectively.

The NLR strategy enables Z3 to prove 7 additional verification problems. In particular, it allows Z3 to prove the program `ntdrvsimpl-cdaudio_simpl1_unsafeil.c`, that is the largest program in the benchmark set (2.1 KLOC). Concerning the time required for executing the NLR strategy, we want to point out that this program takes the 91% of the total NLR time (t_{NLR}), that is 53.04 seconds. Therefore, the remaining six programs only require 5.35 seconds to be transformed. The cFAR algorithm allows Z3 to prove 9 additional verification problems. In this case, about 89% of the total cFAR time (t_{cFAR}), that is 271.62 seconds, is required for specializing two programs, namely `ntdrvsimpl-diskperf_simpl1_safeil.c` (98.82 seconds) and `ntdrvsimpl-floppy_simpl3_safeil.c` (172.80 seconds) whose size is about 1 KLOC each.

9. Related Work and Conclusions

Constraint logic programming, also named constrained Horn clauses, has been shown to be a powerful, flexible formalism to reason about the correctness of programs [1, 10, 11, 22, 27–29, 34, 36, 45, 49]. It can also be used as a common intermediate language for exchanging VCs between software verifiers [1, 3, 9, 26] 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 by Peralta et al. [49]. In their approach the specialization process yields a residual CLP program on which analysis techniques based on abstract interpretation are subsequently applied. In a previous work [11] we have presented a VC generation method which overcomes some limitations which were present in Peralta et al.'s paper [49]. In particular, we have introduced support for (non-recursive) functions, and we have improved scalability by encoding programs as sets of facts, instead of terms. In this paper, we have provided support also for recursive functions and multi-step semantics.

Specialization of interpreters has also been used by Albert et al. [1, 27] to decompile (that is, translate) Java bytecode into Prolog programs. Then, the residual program is given as input to the CiaoPP analyzer for Prolog that makes use of abstract interpretation techniques to infer properties of the original Java bytecode. Although the work by Albert et al. and ours share some objectives and methods, they also exhibit several differences.

First of all, the source languages and their semantics are different: Java bytecode with a big-steps semantics on one side, and C with a multistep semantics on the other side. Also the target languages are different: Prolog on one side, and a CLP language over integers and integer arrays on the other side. As a consequence, the type of verification tools that can be applied and the properties that can be proved are different. In particular, the approach presented in this paper does not produce a program which is directly executable by a CLP (or Prolog) system, but it enables the use of SMT solvers that can prove theorems in the theories of integers and integer arrays. These theorems (such as those expressing properties related to the contents of arrays whose size is a parameter) may not be proved by the CiaoPP analyzer. Another distinctive feature of our approach is that we specialize the interpreter of the program together with the property to be verified, hence enabling a property-directed generation and transformation of verification conditions. We have shown in other papers that this feature may allow us to prove many complex properties besides safety, including recursively defined properties and program equivalence [13, 15].

Also the specialization method of Albert et al. is different from the one presented in this paper. The former follows the approach formalized by Lloyd and Shepherdson [43], while we use unfold/fold rules. While the two approaches are semantically equivalent, we believe that the rule-based approach has some advantages: (i) it gets for free (via folding) the renaming mechanism performed by the *codegen* procedure, and more importantly (ii) it can easily be combined with the many transformations that can be expressed via unfold/fold rules. Another difference lies in the specialization strategies: Albert et al.'s strategy can be classified, by using the terminology of partial evaluation, as a hybrid online-offline control strategy (some unfolding decisions are taken at specialization time), while our VCG strategy is purely offline.

Finally, the two approaches are similar also with respect to scalability, as they both present specialization strategies that run in linear time and produce residual programs that have linear size with respect to the size of the input bytecode. However, here we have proved a more refined property, as we compute the size of the residual programs in terms of the number of atoms, instead of the number of clauses. It would have been interesting to compare the scalability of the two approaches in practice, but this is not straightforward due to the difference of source programs.

Our approach shares the same objective of the work by van Leeuwen [55], 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 [18]. 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 for inferring 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 [18] 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 [31] is able to infer loop invariants, but it uses 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 VCG specialization strategy by using suitable generalization operators [11]. 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 by Moore and Matthews et al. Moore [46] presents 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. Matthews et al. [44] describe a related approach, where it is shown how an off-the-shelf theorem prover and an operational semantics can be converted into a VC generator.

The design of general purpose abstract interpreters, parameterized with respect to the semantics of the programming language has been studied by Cousot [6] and implemented in the TVLA system [41].

Finally, somewhat related to our work, we would like to mention the K rewriting-based framework [53], 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 an important role in the development of highly parametric tools that support the verification of programs, starting from the formal definition of the programming language semantics and the logic of the properties to be proved.

10. 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 M. Hanus, editor, *Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science 4354, pages 124–139, Springer, 2007.

[2] M. Barnett, B.-Y. E. Chang, R. De Line, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, Lecture Notes in Computer Science 4111, pages 364–387, Springer, 2006.

[3] N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories, SMT-COMP '12*, pages 3–11, 2012.

[4] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In N. Piterman and S. Smolka, editors, *Proceedings of TACAS*, Lecture Notes in Computer Science 7795, pages 93–107, Springer, 2013.

[5] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 108–128, Springer-Verlag, 2005.

[6] P. Cousot. Abstract interpretation based static analysis parameterized by semantics. In *Proceedings of the 4th International Symposium on Static Analysis*, SAS '97, pages 388–394, London, UK, Springer-Verlag, 1997.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the 4th ACM-SIGPLAN Symposium on Principles of Programming Languages, POPL '77*, pages 238–252. ACM, 1977.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, POPL '78*, pages 84–96. ACM, 1978.

[9] E. De Angelis, F. Fioravanti, J. A. Navas, and M. Proietti. Verification of programs by combining iterated specialization with interpolation. In *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014*, Electronic Proceedings in Theoretical Computer Science, volume 169, pages 3–18, 2014.

[10] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '14*, Lecture Notes in Computer Science 8413, pages 568–574, Springer, 2014. Available at: http://www.map.uniroma2.it/VeriMAP.

[11] 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. Selected and extended papers from Partial Evaluation and Program Manipulation 2013.

[12] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 91–102. ACM, 2015.

[13] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Proving correctness of imperative programs by linearizing constrained Horn clauses. *Theory and Practice of Logic Programming*, 15(4-5):635–650, 2015.

[14] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. A rule-based verification strategy for array manipulating programs. *Fundamenta Informaticae*, 140(3-4):329–355, 2015.

[15] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Relational Verification through Horn Clause Transformation. In *Proceedings of the 23rd International Symposium on Static Analysis, SAS '16*, Lecture Notes in Computer Science 9837, *To appear*, Springer, 2016.

[16] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, Lecture Notes in Computer Science 4963, pages 337–340, Springer, 2008.

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

[18] J.-C. Filliâtre and A. Paskevich. Why3 - Where programs meet provers. In *Programming Languages and Systems, 22nd European Symposium on Programming, ESOP '13, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS '13, Rome, Italy, March 16–24, 2013. Proceedings*, Lecture Notes in Computer Science 7792, pages 125–128, Springer, 2013.

[19] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In K.-K. Lau, editor, *Proceedings of the Tenth International Workshop on Logic-based Program Synthesis and Transformation, LOPSTR '00, London, UK, 24-28 July 2000*, Lecture Notes in Computer Science 2042, pages 125–146, Springer-Verlag, 2001.

[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] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming. Special Issue on the 25th Annual GULP Conference*, 13(2):175–199, 2013.

[22] C. Flanagan. Automatic software model checking via constraint logic. *Sci. Comput. Program.*, 50(1–3):253–270, 2004.

[23] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. *SIGPLAN Not.*, 36(3):193–205, 2001.

[24] J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.

[25] J. P. Gallagher and B. Kafle. Analysis and Transformation Tools for Constrained Horn Clause Verification. *Theory and Practice of Logic Programming*, 14(4-5):90–101 (Supplementary Materials), 2014.

[26] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Horn clauses as an intermediate representation for program analysis and transformation. *Theory and Practice of Logic Programming*, 15(4-5):526–542, 2015.

[27] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51(10):1409–1427, October 2009.

[28] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier based on Horn Clauses. In C. Flanagan and B. König, editors, *Proc. of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '12*, Lecture Notes in Computer Science 7214, pages 549–551, Springer, 2012.

[29] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 405–416, 2012.

[30] K. S. Henriksen and J. P. Gallagher. Abstract interpretation of pic programs through logic programming. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM '06*, pages 103–179, 2006.

[31] T. Henzinger, T. Hottelier, and L. Kovács. Valigator: A verification tool with bound and invariant generation. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, pages 333–342, 2008.

[32] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems. In D. Giannakopoulou and D. Méry, editors, *FM '12: Formal Methods, 18th International Symposium, Paris, France, August 27–31, 2012. Proceedings*, Lecture Notes in Computer Science 7436, pages 247–251, Springer, 2012.

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

[34] J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded Symbolic Execution for Program Verification. In *Proceedings of the 2nd International Conference on Runtime Verification, RV '11*, Lecture Notes in Computer Science 7186, pages 396–411, Springer, 2012.

[35] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[36] B. Kafle and J. P. Gallagher. Constraint Specialisation in Horn Clause Verification. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM '15, Mumbai, India, January 15–17, 2015*, pages 85–90, ACM, 2015.

[37] 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.

[38] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '96, Stockholm, Sweden*, Lecture Notes in Computer Science 1207, pages 83–103, Springer-Verlag, 1996.

[39] M. Leuschel, S. S.J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 340–375, Springer Berlin Heidelberg, 2004.

[40] M. Leuschel and G. Vidal. Fast offline partial evaluation of logic programs. *Information and Computation*, 235:70–97, 2014.

[41] T. Lev-Ami, R. Manevich, and M. Sagiv. Tvla: A system for generating abstract interpreters. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 367–375, Springer US, 2004.

[42] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.

[43] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[44] J. Matthews, J. Moore, S. Ray, and D. Vroon. Verification condition generation via theorem proving. In M. Hermann and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science 4246, pages 362–376, Springer Berlin Heidelberg, 2006.

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

[46] J. Moore. Inductive assertions and operational semantics. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science 2860, pages 289–303, Springer Berlin Heidelberg, 2003.

[47] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Horspool, editor, *Compiler Construction*, Lecture Notes in Computer Science 2304, pages 209–265, Springer, 2002.

[48] J. C. Peralta and J. P. Gallagher. Imperative Program Specialisation: An Approach Using CLP. In A. Bossi, editor, *Logic Programming Synthesis and Transformation, 9th International Workshop, LOPSTR'99, Venezia, Italy, September 22–24, 1999, Selected Papers*, Lecture Notes in Computer Science 1817, pages 102–117, Springer, 2000.

[49] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In G. Levi, editor, *Proceedings of the 5th International Symposium on Static Analysis, SAS '98*, Lecture Notes in Computer Science 1503, pages 246–261, Springer, 1998.

[50] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[51] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.

[52] C. J. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

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

[54] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[55] A. van Leeuwen. Building verification condition generators by compositional extension. *Electronic Notes in Theoretical Computer Science*, 191(0):73–83, Proceedings of the Doctoral Symposium affiliated with the Fifth Integrated Formal Methods Conference (IFM 2005), 2007.

Appendix

Proof of Lemma 1

Proof. In this paper we have assumed that the initial and error properties are expressed by constraints, and hence both the predicates `initConf` and the predicate `errorConf` is defined by a constrained fact. Thus, the full unfolding of the `initConf` and `errorConf` atoms terminates. Similarly, we assume that the full unfolding of atoms having predicate different from `tr` and `reach` (that is, `eval`, `update`, etc.) terminates. All `tr` atoms, except those corresponding to the semantics of function calls, are defined by non recursive predicates. Thus, the full unfolding of these atoms terminates. Let us now consider a sequence of unfolding steps performed during the while-do loop of the Unfolding phase by using the annotation *UA*. We will show that the sequence is finite. We refer to the command occurring in the source configuration of `tr` or `reach` as the *source command*. We have the following properties:
(i) by full unfolding a `tr` atom whose source command is not a function call, the `tr` atom is deleted;
(ii) by unfolding once a `tr` atom whose source command is a function call, the `tr` atom is replaced by an atom of the form `reach(cf(cmd(L,Cmd),_),_)` which is non-unfoldable because `L` is the entry point of a function definition;
(iii) by unfolding once a `reach` atom whose source command is an assignment, and then by fully unfolding the derived `tr` atom, the `reach` atom is replaced by a new `reach` atom whose source configuration has a command with a greater label (according to the linear order of the labels);
(iv) by unfolding once a `reach` atom whose source command is a jump `goto(L)`, and then by full unfolding of the derived `tr` atom, the `reach` atom is replaced by an atom of the form `reach(cf(cmd(L,Cmd),_),_)` which is non-unfoldable because `L` occurs in a `goto` command;
(v) by unfolding once a `reach` atom whose source command is the `abort` command, and then unfolding the derived atoms, the `reach` atom is deleted.

By properties (i)–(v), during the sequence either (a) the number of unfoldable atoms in the body of a clause decreases or (b) it does not increase and the source configuration of a `reach` atom contains a command with a greater label. Thus, the sequence of unfolding steps is finite. □

Proof of Lemma 2

Proof. After the Unfolding phase, all atoms in the body of clauses in *SpC* have the `reach` predicate symbol. Indeed, all other atoms are unfoldable (fully or once). Then, as prescribed by the Definition & Folding phase, only clauses of the form `newr(V) :- reach(cf1,cf2)` will be introduced for folding clauses in *SpC*. Points (i)–(vii) follow from the definition of the semantics in Table 2. In particular, observe the following facts.

Points (i)–(iii) follow from the definition of `tr` as a binary relation between configurations.

Point (iv) follows from the fact that an error configuration can only contain `halt` or `abort` commands, and by unfolding a function call we can only derive new `reach` atoms whose target configuration contains either an `abort` or a `return` command.

Point (v) directly follows from the definition of `tr`.

Point (vi) follows from the fact that the domain of every global environment is determined by the set of global variables occurring in the program, and the domains of the local environments `S1` and `S2` are determined by the formal parameters and local variables of the functions where `L1` and `L2` appear.

Point (vii) follows from the fact that the set of program variables belonging to the domain of the environment at a given label is uniquely determined by the label itself, and their values are represented by distinct CLP variables.

By Points (i)–(vii), every clause in Δ contains a label of P in its source configuration and, for every label of P, there are at most three definitions whose source configuration contains that label. Hence the thesis. □

Proof of Lemma 3

Proof. The proof is by contradiction. Suppose that there exist two distinct definitions:

*D*1. `new1(X) :- reach(cf1,_)`

*D*2. `new2(X) :- reach(cf2,_)`

such that `cf1` and `cf2` have commands with distinct labels and by unfolding (possibly several times) *D*1 and *D*2, respectively, we get two clauses of the form:

*D*3. `new1(X) :- ...,reach(cf(cmd(L,Cmd),_),cfz)`

*D*4. `new2(X) :- ...,reach(cf(cmd(L,Cmd),_),cfz)`
and then `reach(cf(cmd(L,Cmd),_),cfz)` is unfolded in both clauses. We may assume that *D*3 and *D*4 are the first (in the unfolding order) pair of clauses where this happens. Since `L` is reached from two distinct labels, it must occur in some `ite` or `goto`, or it is the entry point of a function definition. Thus, according to the unfolding annotation *UA*, `reach(cf(cmd(L,Cmd),_),cfz)` is non-unfoldable, and we get a contradiction. □

Proof of Lemma 4

Proof. We prove this lemma by taking $k=6$.

If the command in `cf1` is `halt`, `abort`, or `return`, then by unfolding `reach(cf1,cfz)` in *C* we derive at most one clause of the form:
*D*1. `newp(X)`ϑ
where ϑ is the most general unifier of `cf1` and `cfz`. Hence, the lemma holds. Otherwise, by unfolding `reach(cf1,cfz)` in *C* we derive a clause of the form:
*D*2. `newp(X) :- tr(cf1,Y), reach(Y,cfz)`
and $SpC = \{D2\}$. There are the following two cases.
(Case 1: `tr(cf1,Y)` *is fully unfoldable.*) By unfolding `tr(cf1,Y)` in *D*2 we derive at most two clauses. Indeed, precisely two clauses (*D*3, *D*4 below) in the case where the command in `cf1` is an if-then-else, and one clause (*D*3) otherwise:
*D*3. `newp(X) :- c, reach(cf2,cfz)`
*D*4. `newp(X) :- d, reach(cf3,cfz)`
where `c` and `d` are constraints. In the case where the command in `cf1` is an if-then-else both `reach(cf2,cfz)` and `reach(cf3,cfz)` is non-unfoldable. Thus, $\alpha(SpC) = \alpha(\{D3, D4\}) = 4$ and we get the thesis.

Otherwise, $SpC = \{D3\}$ and `reach(cf2,cfz)` may be unfoldable. If the command in `cf2` is `abort`, then from *D*3 we derive a clause with empty body. If the command in `cf2` is either an assignment or a jump, then from *D*3 we derive, by unfolding `reach(cf2,cfz)` and then by unfolding also the generated `tr` atom, a clause of the form:
*D*5. `newp(X) :- e, reach(cf4,cfz)`
where `e` is a constraint. Otherwise, `reach(cf2,cfz)` is non-unfoldable. The unfolding of *D*5 will eventually terminate (see Lemma 1) and will derive precisely one clause with at most one atom in its body. Thus, by unfolding, from *D*2 we derive (possibly in many steps), a set *SpC* consisting of at most one clause with at most one atom in its body. In this case $\alpha(SpC) \leq 2$ and we get the thesis.
(Case 2: `tr(cf1,Y)` *is unfoldable once.*) This case occurs when `cf1` contains a function call. By unfolding `tr(cf1,Y)` in *D*2 we derive two clauses of the form:
*D*3. `newp(X) :- reach(cf2,cf3), reach(cf4,cfz)`
*D*4. `newp(X) :- reach(cf2,cf5), reach(cf6,cfz)`
where: (i) `cf2` corresponds to the entry point of a function, `cf3` corresponds to an `abort` command, `cf5` corresponds to a `return` command, and `cf4`, `cf6` are the configurations reached after the exit from the function call. Then, by the unfolding annotation *UA*, `reach(cf2,cf3)` and `reach(cf2,cf5)` are non-unfoldable. The atoms `reach(cf4,cfz)` and `reach(cf6,cfz)` are either unfoldable once or non-unfoldable. Since the unfolding of a `reach` atom (possibly followed by the unfolding of a `tr` atom) which is annotated as unfoldable once has the effect of replacing that atom by at most one new atom, when the Unfolding phase terminates, from *D*3 and *D*4 we derive a set *SpC* consisting of two clauses with at most two atoms in their body. Thus, $\alpha(SpC) \leq 6$. □

Proof of Theorem 6

Proof. (*Termination.*) The while-loop within the Unfolding phase terminates because no predicate is annotated as unfoldable, and hence exactly one unfolding is performed for each clause in *InCls*. The Definition-Introduction phase in Figure 4 terminates because it introduces a finite number of definitions, at most one for each atom `B` occurring in the body of a clause in *SpC*. The while-loop that iterates the Unfolding and Definition-Introduction phases terminates because each new definition is of the form `newp(L) :- B`, where `L` is a tuple of variables occurring in `B`.

The while-loop of the Folding phase in Figure 5, terminates because at each iteration the number of occurrences of atoms that can be folded decreases by one.

(*Correctness.*) The correctness of the NLR strategy follows from the correctness of the transformation rules presented in Section 4.1.
(*Size of the Output.*) For any given atom `B` to be folded by using a definition introduced by the NLR strategy, the Definition-Introduction in Figure 4 either (*then* branch) replaces the renamed apart definition `H :- B` in *Defs* with a definition of the form `F :- B` such that $vars(\mathtt{H}) \subseteq vars(\mathtt{F})$ or (*else* branch) adds a definition of the form `F :- B` where $vars(\mathtt{F}) \subseteq vars(\mathtt{B})$. Therefore, at the end of the Definition-Introduction phase, for each atom `B` to be folded, there exists a single definition that can be used to fold all the renamed apart variants of `B` occurring in the clauses in *SpC*. Hence, we have that the NLR strategy does not increase the number of predicates with respect to those introduced by the VCG strategy. By construction, each predicate p' in $Prog'$ corresponds to a predicate p in *Prog* such that p and p' are defined by sets of clauses with the same number of atoms (possibly, with smaller arity). □

Proof of Theorem 7

Proof. (*Termination.*) Termination follows from the fact that the erasure E is finite and its size decreases at each iteration of the loop of the cFAR algorithm.
(*Correctness.*) Since all the erasures that are not safe are removed by the cFAR algorithm, for all atoms `A`, we have that $\mathtt{A} \in \mathcal{M}(Prog)$ iff $\mathtt{A}|_E \in \mathcal{M}(Prog|_E)$. Thus, in particular, $\mathtt{unsafe} \in \mathcal{M}(Prog)$ iff $\mathtt{unsafe} \in \mathcal{M}(Prog|_E)$.
(*Size of the Output.*) The program $Prog|_E$ is obtained from the program *Prog* by replacing every atom `A` in *Prog* with the atom $\mathtt{A}|_E$. Thus all clauses $C|_E$ in $Prog|_E$ have the same number of atoms of the corresponding clause C in *Prog*. Since, by definition, the size of a clause C is the number of the atoms occurring in C and the size of a set S of clauses is the sum of the sizes of the clauses in S, we have that $\alpha(Prog) = \alpha(Prog|_E)$. □