Counter-Example Guided Program Verification

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Bui Phi Diep

Uppsala University, Sweden
{parosh,mohamed_faouzi.atig,bui.phi-diep}@it.uu.se

Abstract. This paper presents a novel counter-example guided abstraction refinement algorithm for the automatic verification of concurrent programs. Our algorithm proceeds in different steps. It first constructs an abstraction of the original program by slicing away a given subset of variables. Then, it uses an external model checker as a backend tool to analyze the correctness of the abstract program. If the model checker returns that the abstract program is safe then we conclude that the original one is also safe. If the abstract program is unsafe, we extract an "abstract" counter-example. In order to check if the abstract counter-example can lead to a real counter-example of the original program, we add back to the abstract counter-example all the omitted variables (that have been sliced away) to obtain a new program. Then, we call recursively our algorithm on the new obtained program. If the recursive call of our algorithm returns that the new program is unsafe, then we can conclude that the original program is also unsafe and our algorithm terminates. Otherwise, we refine the abstract program by removing the abstract counter-example from its set of possible runs. Finally, we repeat the procedure with the refined abstract program. We have implemented our algorithm, and run it successfully on the concurrency benchmarks in SV-COMP15. Our experimental results show that our algorithm significantly improves the performance of the backend tool.

1 Introduction

Leveraging concurrency effectively has become key to enhancing the performance of software, to the degree that concurrent programs have become crucial parts of many applications. At the same time, concurrency gives rise to enormously complicated behaviors, making the task of producing correct concurrent programs more and more difficult. The main reason for this is the large number of possible computations caused by many possible thread (or process) interleavings. Unexpected interference among threads often results in Heisenbugs that are difficult to reproduce and eliminate. Extensive efforts have been devoted to address this problem by the development of testing and verification techniques. Model checking addresses the problem by systematically exploring the state space of a given program and verifying that each reachable state satisfies a given property. Applying model checking to realistic programs is problematic, due to the state explosion problem. The reason is that we need (1) to exhaustively explore the

entire reachable state space in all possible interleavings, and (2) to capture and store a large number of global states.

Counter-Example Guided Abstraction Refinement (CEGAR) (e.g., [5,15,4,11,17]) approach is one of the successful techniques for verifying programs. This approach consists in four basic steps:

- *Abstraction step:* Construct a finite-state program as an abstraction of the original program using predicate abstraction (with a set of predicates) and go to the *Verification step*.
- *Verification step:* Use a model checker to check if the constructed finite state program satisfies the desired property. If it is the case, then the original program satisfies also the property and the verification algorithm terminates; otherwise extract a counter-example and go to the *Analysis step*.
- *Analysis Step:* Check if the retuned counter example is spurious or not. If it is not, then we have a real bug in the original program and the verification algorithm terminates; otherwise go to the *Refinement step*.
- *Refinement Step:* If the counter-example is spurious, refine the set of used predicates in the *Abstraction step* to eliminate the counter example. Return to the *Abstraction step* with this new refined set of predicates.

The CEGAR approach has been successfully implemented in tools, such as SLAM [4], BLAST [5], MAGIC [8] and CPAchecker [6]. However, CEGAR may also suffer from the state-space exploring problem in the case of concurrent programs due to the large number of possible interleavings.

In this paper we present a variant of the CEGAR algorithm (called Counter-Example Guided Program Verification (CEGPV)) that addresses the state-space explosion problem encountered in the verification of concurrent programs. The work-flow of our CEGPV algorithm is given in Fig. 1. The algorithm consists of four main modules, the *abstraction*, the *counter-example mapping*, the *reconstruction* and the *refinement*. It also uses an external model checker tool.

The *abstraction* module takes as input a concurrent program $\mathcal{P}$ and a subset V_0 of its shared variables. It then constructs an over-approximation of the program $\mathcal{P}$, called $\mathcal{P}'$, as follows. First, it keeps variables in the set V_0 and slices away all other variables of the program $\mathcal{P}$. Occurrences of the sliced variables are replaced by non-deterministic values. Second, some instructions, where the sliced variables occur, in the program $\mathcal{P}$ can be removed. Then, the *model checker* takes as input $\mathcal{P}'$, and checks whether it is safe or not. If the *model checker* returns that $\mathcal{P}'$ is safe, then $\mathcal{P}$ is also safe, and our algorithm terminates. If $\mathcal{P}'$ is unsafe, then the *model checker* returns a counter-example π'.

The *counter-example mapping* module takes the counter-example π' as its input. It transforms the run π' to a run π of the program resulting of the *abstraction* module (using V_0 as its set of shared variables).

The *reconstruction* module takes as input the counter-example π of $\mathcal{P}'$. It checks whether π can lead to a real counter-example of $\mathcal{P}$. In particular, if $\mathcal{P}'$ is identical to $\mathcal{P}$, then the algorithm concludes that $\mathcal{P}$ is unsafe, and terminates. Otherwise, the *reconstruction* adds back all omitted variables and lines of codes

to create a program $\mathcal{P}_1$ while respecting the flow of the instructions in π and the valuation of the variables in V_0. Hence, $\mathcal{P}_1$ has as its set of variables only the omitted ones. Then, CEGPV algorithm then recursively calls itself to check $\mathcal{P}_1$ in its next iteration. If the iteration returns that $\mathcal{P}_1$ is unsafe, then the run π leads to a counter-example of the program $\mathcal{P}$. The algorithm concludes that $\mathcal{P}$ is unsafe and terminates. Otherwise, the run π cannot lead to a counter-example of $\mathcal{P}$. Then the algorithm needs to discard the run π from $\mathcal{P}'$.

The *refinement* adds π to the set of spurious counter-examples of $\mathcal{P}'$. It then refines $\mathcal{P}'$ by removing all these spurious counter-examples from the set of runs of $\mathcal{P}'$. The new resulting program is then given back to the *model checker* tool.

Our CEGPV algorithm has two advantages. First, it reduces the number of variables in the model-checked programs to prevent the state-space explosion problem. Second, all modules are implemented using code-to-code translations.

In order to evaluate the efficiency of our CEGPV algorithm, we have implemented it as a part of an open source tool, called CEGPV, for the verification of C/pthreads programs. We used CBMC version 5.1 as the backend tool [10]. We then evaluated CEGPV on the benchmark set from the Concurrency category of the TACAS Software Verification Competition (SV-COMP15) [2]. Our experimental results show that CEGPV significantly improve the performance of CBMC, showing the potential of our approach.

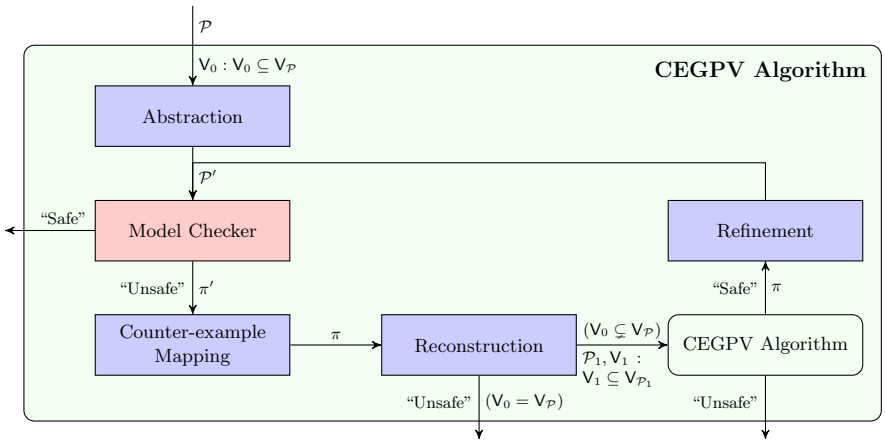


Fig. 1: An overview of the CEGPV algorithm.

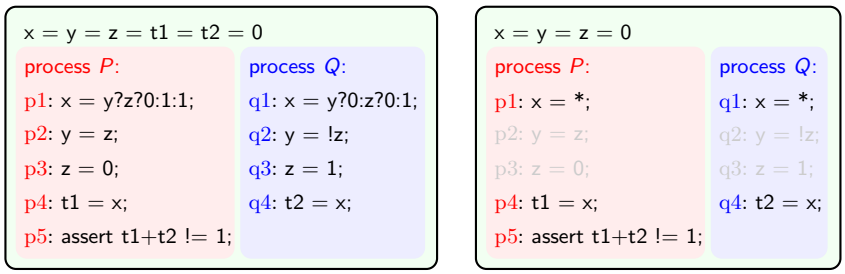
Related Work. CEGAR is one of the successful techniques used in program verification. Our CEGPV algorithm can be seen as a new instance of the CEGAR algorithm that can be implemented on the top of any verification tool. In contrast with the classical CEGAR algorithms (e.g., [9,5,11,18,12]) where the programs are abstracted using a set of predicates, our CEGPV algorithm uses variable slicing techniques to obtain the abstract program.

Variable slicing is also one of the verification guided approaches to address the state-space exposing problem. In [18], an analysis tool for detecting memory

leaks is presented based on slicing some of the program variables. Each generated abstract program is then checked by a backend tool. RankChecker [7] is a testing tool based on an assumption that most concurrency bugs have a small number of variables involved. To reduce the search space, it forces processes in a concurrent program to interleave at certain points that access a subset of variables. Corral [15] abstracts the input program by only keeping track of a subset of variables. If the counter-example of the abstract program is spurious, Corral then refines the abstraction by decreasing the set of omitted variables. The algorithm terminates once the counter-example corresponds to a run of the original program. Our CEGPV algorithm also abstract programs by slicing away some variables (as it is also done by the localization reduction techniques [13,14]). However, our CEGPV algorithm has the feature to recursively call itself in order to check if the counter-example can lead to a real one while trying to keep the number of variables of the model-checked programs as small as possible.

2 Motivating Example

In this section, we informally illustrate the main concepts of our algorithm.



(a) A simple program $\mathcal{P}$ (b) Abstract program $\mathcal{P}_1$

Fig. 2: A toy example and its abstraction

Fig. 2a is a simplified version of a program in the concurrent C benchmark in SVCOMP [2]. The program $\mathcal{P}$ has two processes, called P and Q, running in parallel. Processes communicate through five shared variables which are x, y, z, t1 and t2, ranging over the set of integers. All variables are initialized to 0. The behavior of a process is defined by a list of C-like instructions. Each instruction is composed of a unique label and a statement. For example, in process P, the instruction p1: x = y ? z ? 0 : 1 : 1 has p1 as a label, and x = y ? z ? 0 : 1 : 1 as a statement. That statement is a ternary assignment in which it assigns 0 to x if both y and z are equal to 1, and assigns 1 to x otherwise. The assertion labeled by p5 holds if the expression t1 + t2 is different from 1, and in that case the program is declared to be safe. Otherwise, the program is unsafe.

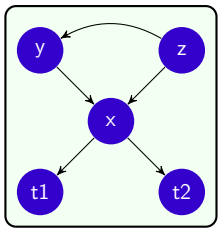


Fig. 3: Dependency graph of $\mathcal{P}$.

In order to apply our algorithm, we first need to determine a subset of program variables that will be sliced away. To that aim, we construct a dependency graph between variables. The dependency graph consists of a number of vertices and directed edges. Each vertex corresponds to a variable of the program. The edges describe the flow dependency between these variables. The dependency graph of the program $\mathcal{P}$ is given in Fig. 3. For instance, x depends on both y and z due to the two assignments labeled by p1 and q1. Similarly, the assignment labeled by p2 creates a dependency between the variables y and z. We use the dependency graph to decide the first set of variables to be sliced away. In general, we keep variables that influence the safety of the program. In the settings of the example, the variables t1 and t2 are used in the assertion at p5 and therefore we keep track of the variables t1 and t2. Furthermore, we keep also track of x since t1 and t2 are dependent on x.

Once we have the subset of variables {t1, t2, x} to be preserved, we need to slice away the variables {y, z}. To do that, we abstract the program by replacing occurrences of the variables y and z by a non-deterministic value *. Assignments labeled by p1 and q1 are transformed to x = * ? * : 0 ? 1 ? 1 and x = * ? 0 : * ? 0 ? 1, respectively. We make a further optimization to transform these assignments to x = *. Since we are not anymore keeping track of the variables y and z, instructions which are assignments to these variables can be removed. In this case, we remove the instructions labeled by p2, p3, q2 and q3 from the abstract program. All the other instructions remain the same. Resulting abstract program, called $\mathcal{P}_1$, is given in Fig. 2b. $\mathcal{P}_1$ has only three variables t1, t2 and x, and five instructions.

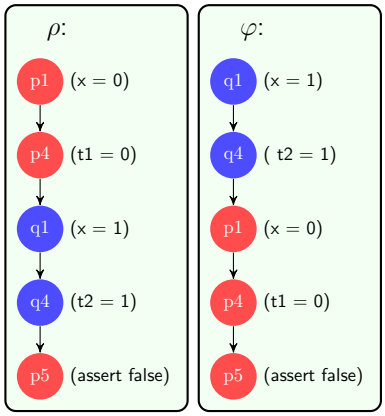


Fig. 4: Counter-examples of $\mathcal{P}_1$

The next step of our algorithm is to feed the abstract program to a model checker. The model checker checks whether the program is safe or not. If the program is unsafe, the model checker returns a counter-example. In our case, since $\mathcal{P}_1$ is unsafe, we assume the model checker returns a counter-example, called ρ, given in Fig. 4. In the obtained counter-example ρ, the process P executes the instruction labeled by p1. At that instruction, the non-deterministic symbol * returns the value 0, and therefore x is assigned to 0. Then the process P executes the instruction labeled by p4 and sets the value of t1 to 0. The control then switches to the process Q which executes the instructions labeled by q1 and q4. They evaluate both x and t1 to 1. Then, the assertion in the instruction labeled by p5 is checked. The expression in the assertion, t1 + t2 != 1, is evaluated to false, so the program is unsafe.

Although ρ is the counter-example of $\mathcal{P}_1$, ρ is not identified to be a counter-example of $\mathcal{P}$ since $\mathcal{P}_1$ is an abstraction of $\mathcal{P}$. In order to check whether ρ

can lead to a counter-example of $\mathcal{P}$, we need to add back some of the omitted variables and lines of codes. Adding back this information to ρ will result in a new program, called $\mathcal{S}_\rho$. In this case, we add y and z to ρ.

The program $\mathcal{S}_\rho$ is given in Fig. 5. When adding back variables, several instructions are restored such as the instructions labeled by p2, p3, q2 and q3. Variables, which appear in the counter-example, can be discarded since their values are known. For example, x at p1 in ρ is 0. We replace the occurrence of x in q1 by 1. We also transform the assignment in the instruction labeled by p1 to an assumption to check whether the value of x is equal to the value of right hand side of assignment, i.e. assume 0 == y ? z ? 0 : 1 : 1. The assumption blocks the execution until the expression in the assumption is evaluated to true. Similarly, the instruction labeled by p4 is transformed to assume 0 == 0. Then, we remove assumptions that are trivially true such as assume 0 == 0. Since $\mathcal{S}_\rho$ needs to respect the order of instructions in ρ, the instruction labeled by p1 is only executed after the instruction labeled by q3.

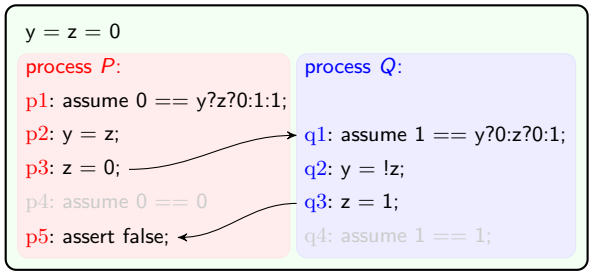


Fig. 5: The program $\mathcal{S}_\rho$

The model checker checks $\mathcal{S}_\rho$ and returns that $\mathcal{S}_\rho$ is safe. This means ρ can not lead to a counter-example of $\mathcal{P}$. We then need to refine $\mathcal{P}_1$ to exclude ρ from its set of runs. Therefore, we create a refinement of $\mathcal{P}_1$, called $\mathcal{P}_2$ and given in Fig. 6, as follows. We use an observer to check whether the actual run is identical to the run ρ. Two runs are identical if (1) their orders of executed instructions are the same, and (2) valuations of variables after each instruction are the same in both runs. If the actual run is identical to the run ρ, then that run is safe. For the sake of simplicity, we model the observer as a sequence of conditional statements. After each instruction in the run ρ, except the assertion at the end of ρ, we create a conditional statement to re-evaluate values of variables. For instance, if x == 0 follows the assignment x = * at p1, where 0 is the value of x at instruction labeled by p1 in ρ. If if x == 0 is passed, then the execution can check if t1 == 0 after running assignment t1 = x at p4. Otherwise, the execution is no longer followed by the observer. If an execution passes all conditional statements of the

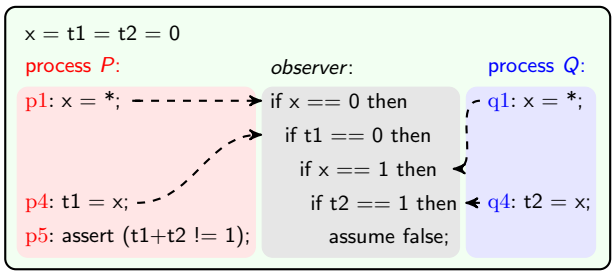


Fig. 6: The refined program $\mathcal{P}_2$

observer, then the actual run is identical to ρ. The assumption assume false at the end of observer is to prevent the execution of the assertion at p5. Hence, $\mathcal{P}_2$ excludes ρ from its runs.

The model checker checks $\mathcal{P}_2$. It returns a counter-example, called φ, as given in Fig. 4. In φ, the instructions of the process Q, which are labeled by q1 and q4, are issued first. After that, the instructions of P, which are labeled by p1, p4 and p5, are performed. Similar to the way we verify ρ, we add y and z back to φ and construct a new program to simulate φ, called $\mathcal{S}_\varphi$. $\mathcal{S}_\varphi$ is presented in Fig 7. In the counter-example $\mathcal{S}_\varphi$, the variables x, t1 and t2 are replaced by their values in φ. Then, instructions labeled by p4 and q4 are removed due to the optimization. We also force $\mathcal{S}_\varphi$ to respect the flow of the counter-example φ. For instance, the instruction labeled by p1 only runs after the instruction labeled by q3.

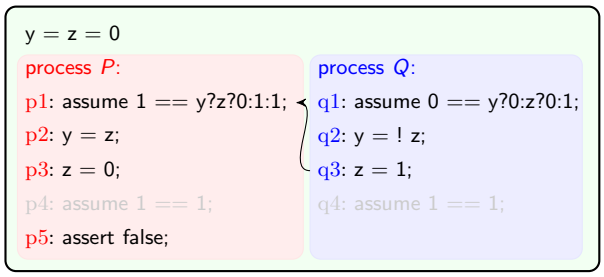


Fig. 7: The program $\mathcal{S}_\varphi$

The model checker checks $\mathcal{S}_\varphi$. It then concludes that $\mathcal{S}_\varphi$ is unsafe with a proof by a counter-example, called π, given in Fig. 8. We need to verify whether π can lead to a counter-example of $\mathcal{P}$ by adding more variables and lines of codes, and then constructing a new program that respects the flow of instructions in π. However, all variables of the program $\mathcal{P}$ are used, so π is a counter-example of $\mathcal{P}$. Thus, $\mathcal{P}$ is unsafe and the algorithm stops.

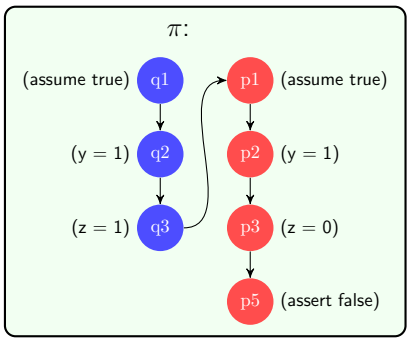


Fig. 8: Counter-example of $\mathcal{S}_\varphi$

3 Concurrent programs

In this section, we describe the syntax and semantics of programs we consider but before that we will introduce some notations and definitions.

For A a finite set, we use $|A|$ to denote its size. Let A and B be two sets, we use $f : A \mapsto B$ to denote that f is a function that maps any element of A to an element of B. For $b \in B$, we use $b \in f$ to denote that there is an $a \in A$ such that $f(a) = b$. For $a \in A$ and $b \in B$, we use $f[a \hookleftarrow b]$ to denote the function f' where $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \neq a$.

Syntax. Fig. 9 gives the grammar for a C-like programming language that we use for defining concurrent programs. A concurrent program $\mathcal{P}$ starts by defining a set of shared variables. Each shared variable is defined by the command `var`

followed by a unique identifier. We assume that the variable ranges over some (potentially infinite) domain $\mathbb{D}$. Then the program $\mathcal{P}$ defines a set of processes (or threads). Each process has a unique identifier p and its code is a sequence consists of instructions (which is placed between `begin` and `end`). An instruction ins is of the form "loc: stmt", where loc is a label (or control location), and stmt is a statement. We use *label*(ins) to denote the label loc of the instruction ins and *stmt*(loc) to denote the statement stmt. We use $\mathsf{V}_{\mathcal{P}}$ to denote the set of variables, $\mathsf{Proc}_{\mathcal{P}}$ to denote the set of processes of the program $\mathcal{P}$. For a process $P \in \mathsf{Proc}_{\mathcal{P}}$, let $\mathbb{I}_P$ be the set of instructions in the code of P and $\mathbb{Q}_P$ be the set of labels appearing in its code. We assume w.l.o.g. that each instruction has a unique label. Let $\mathbb{I}_{\mathcal{P}} := \cup_{P \in \mathsf{Proc}_{\mathcal{P}}} \mathbb{I}_P$, and $\mathbb{Q}_{\mathcal{P}} := \cup_{P \in \mathsf{Proc}_{\mathcal{P}}} \mathbb{Q}_P$. We assume that we dispose of a function $init : \mathsf{Proc} \mapsto \mathbb{Q}_{\mathcal{P}}$ that returns the label of the first instruction to be executed by each process.

A skip statement corresponds to the empty statement that leaves the program state unchanged. A goto statement of the form "$\texttt{goto}\ \mathsf{loc}_1, \ldots \mathsf{loc}_n$" jumps nondeterministically to an instruction labeled by loc_t for some $t \in \{1, \ldots, n\}$. An assignment statement (asg for short) of the form "$x := expr$" assigns to the variable x the current value of the expression *expr*. An assumption statement (asp) of the form "`assume` *expr*" checks whether the expression *expr* evaluates to *true* and if not, the process execution is blocked till that the value of *expr* is *true*. An assertion statement (asr) of the form "`assert` *expr*" checks whether the expression *expr* evaluates to *true*, and if not the execution of the program is aborted. A conditional statement (cnd) of the form "$\texttt{if}\ \langle expr \rangle\ \texttt{then}\ inst_1\ \texttt{else}\ inst_2\ \texttt{fi}$" executes the instruction $inst_1$, if the expression *expr* evaluates to *true*. Otherwise, it executes the instruction $inst_2$. We assume w.l.o.g. that the label of $inst_1$ is different from the label of $inst_2$. We assume a language of expressions *expr* interpreted over $\mathbb{D}$. Furthermore, in order to allow nondeterminism, *expr* can receive the non-deterministic value $*$. We use Expr to denote the set of all expressions in $\mathcal{P}$. Let $\mathsf{Var}_{expr} : \mathsf{Expr} \mapsto 2^{\mathsf{V}_{\mathcal{P}}}$ be a function that returns the set of variables appearing in a given expression (e.g., $\mathsf{Var}_{expr}(\mathsf{y} + \mathsf{z} + 1) = \{\mathsf{y}, \mathsf{z}\}$).

$$
\begin{array}{rcl}
\langle \textit{c-prog} \rangle & ::= & \langle var \rangle^+\ \langle process \rangle^+ \\
\langle var \rangle & ::= & \texttt{var}\ x\ ; \\
\langle process \rangle & ::= & \texttt{process}\ p\ \texttt{begin}\ \langle inst \rangle^*\ \texttt{end} \\
\langle inst \rangle & ::= & \mathsf{loc}{:}\langle stmt \rangle\ ; \\
\langle stmt \rangle & ::= & \texttt{skip} \\
 & & \mid x := \langle expr \rangle \\
 & & \mid \texttt{goto}\ \mathsf{loc}_1, \ldots \mathsf{loc}_n \\
 & & \mid \texttt{assume}\ \langle expr \rangle \\
 & & \mid \texttt{assert}\ \langle expr \rangle \\
 & & \mid \texttt{if}\ \langle expr \rangle \\
 & & \texttt{then}\ \langle inst \rangle \texttt{else}\ \langle inst \rangle\ \texttt{fi} \\
\langle expr \rangle & ::= & \langle expr \rangle | *
\end{array}
$$

Fig. 9: Syntax of concurrent programs

Semantics. We describe the semantics informally and progressively. Let us first consider the case of a (sequential) program $\mathcal{P}_s$ that has only one process P (i.e., $\mathsf{Proc}_{\mathcal{P}_s} = \{P\}$). A sequential configuration c is then defined by a pair $(\mathsf{loc}, \mathsf{state})$ where $\mathsf{loc} \in \mathbb{Q}_P$ is the label of the next instruction to be executed by the process P, and $state : \mathsf{V}_{\mathcal{P}} \mapsto \mathbb{D}$ is a function that defines the valuation of each shared variable. The initial sequential configuration $c_{init}(\mathcal{P}_s)$ is defined

by $(init(P), state_{init})$ where $state_{init}(x) = 0$ for all $x \in \mathsf{V}_{\mathcal{P}_s}$. In other words, at the beginning of the program, all variables have value 0 and the process P will execute the first instruction in its code. The transition relation $\rightarrow_{\mathcal{P}_s}$ on sequential configurations is defined as usual: For two sequential configurations c, c', we write $c \rightarrow_{\mathcal{P}_s} c'$ to denote that the program $\mathcal{P}_s$ can move from c to c'.

Now, we consider the case of the concurrent program $\mathcal{P}$ that has at least two processes (i.e., $|\mathsf{Proc}_{\mathcal{P}}| \geq 2$). For every $P \in \mathsf{Proc}_{\mathcal{P}}$, let $\mathcal{P}_P$ be the sequential program constructed from $\mathcal{P}$ by deleting the code of any process $P' \neq P$ (i.e., $\mathcal{P}_P$ contains only the instructions of the process P). We define a function *label definition* $\bar{q} : \mathsf{Proc}_{\mathcal{P}} \mapsto \mathbb{Q}_{\mathcal{P}}$ that associates for each process $P \in \mathsf{Proc}_{\mathcal{P}}$, the label $\bar{q}(P) \in \mathbb{Q}_P$ of the next instruction to be executed by P. A concurrent configuration (or simply configuration) c is a pair $(\bar{q}, state)$ where $\bar{q}$ is a label definition, and *state* is a memory state. We use $\mathsf{LabelOf}(c)$, $\mathsf{StateOf}(c)$ to denote $\bar{q}$ and *state* respectively. The initial configuration $c_{init}(\mathcal{P})$ is defined by $(\bar{q}_{init}, state_{init})$ where $\bar{q}_{init}(P) = init(P)$ for all $P \in \mathsf{Proc}_{\mathcal{P}}$, and $state_{init}(x) = 0$ for all $x \in \mathsf{V}_{\mathcal{P}}$. In other words, at the beginning, each process starts at the initial label, and all variables have value 0. We use $C(\mathcal{P})$ to denote the set of all configurations of the program $\mathcal{P}$. Then, the transition relation between configurations is defined as follows: For two given configurations $c = (\bar{q}, state)$ and $c' = (\bar{q}', state')$ and a label $\mathsf{loc} \in \mathbb{Q}_P$ of some process P, we write $c \xrightarrow{\mathsf{loc}}_{\mathcal{P}} c'$ to denote that program $\mathcal{P}$ can move from the configuration c to the configuration c' by executing the instruction labeled by loc of the process P. Formally, we have $c \xrightarrow{\mathsf{loc}}_{\mathcal{P}} c'$ iff $(\bar{q}(P), state) \rightarrow_{\mathcal{P}_P} (\bar{q}'(P), state')$, $\bar{q}(P) = \mathsf{loc}$, and for every $P' \in (\mathsf{Proc}_{\mathcal{P}} \setminus \{P\})$, we have $\bar{q}(P') = \bar{q}'(P')$.

A run π of $\mathcal{P}$ is a finite sequence of the form $c_0 \cdot \mathsf{loc}_1 \cdot c_1 \cdot \mathsf{loc}_2 \cdots \mathsf{loc}_m \cdot c_m$, for some $m \geq 0$ such that: (1) $c_0 = c_{init}(\mathcal{P})$ and (2) $c_i \xrightarrow{\mathsf{loc}_{i+1}}_{\mathcal{P}} c_{i+1}$ for all $i \in \{0, \ldots, m-1\}$. In this case, we say that π is labeled by the sequence $\mathsf{loc}_1\mathsf{loc}_2 \ldots \mathsf{loc}_m$ and that the configuration c_m is reachable by $\mathcal{P}$. We use $\mathsf{Trace}(\pi)$ and $Target(\pi)$ to denote the sequence $\mathsf{loc}_1 \cdot \mathsf{loc}_2 \ldots \mathsf{loc}_m$ in π and the configuration c_m, respectively. We use $\Pi_{\mathcal{P}}$ to denote the set of all runs of the program $\mathcal{P}$. The program $\mathcal{P}$ is said to be safe if there is no run π reaching a configuration $c = (\bar{q}, state)$ (i.e., $Target(\pi) = c$) such that $\bar{q}(P)$, for some process $P \in \mathsf{Proc}_{\mathcal{P}}$, is the label of an assertion statement of the form "`assert` *expr*" where the expression *expr* can be evaluated to *false* at the configuration c.

4 Counter-Example Guided Program Verification

In this section, we present our Counter-Example Guided Program Verification (CEGPV) algorithm. The CEGPV algorithm takes a program $\mathcal{P}$ as its input and returns whether $\mathcal{P}$ is safe or not. The work-flow of the algorithm is given in Fig. 1. The algorithm consists of four modules, the *abstraction*, the *counter-example mapping*, the *reconstruction* and the *refinement*. It also uses an external *model checker* as a back-end tool. Recall that $\mathsf{V}_{\mathcal{P}}$ denotes the set of variables of the program $\mathcal{P}$. The algorithm starts by selecting a subset of variables $\mathsf{V}_0 \subseteq \mathsf{V}_{\mathcal{P}}$ using a dependency graph (not shown in Fig. 1 for sake of simplicity).

The *abstraction* takes $\mathcal{P}$ and V_0 as its input. It then constructs an over-approximation of $\mathcal{P}$, called $\mathcal{P}'$, as follows. First, it keeps variables in V_0 and slices away all other variables of $\mathcal{P}$. Occurrences of the sliced variables are replaced by a non-deterministic value. Second, some instructions, where the sliced variables occur, in $\mathcal{P}$ can be discarded. After that, $\mathcal{P}'$ is given to a *model checker*. Observe that $\mathcal{P}'$ has V_0 as its set of shared variables.

Then, the *model checker* takes as input $\mathcal{P}'$, generated by the *abstraction* module or the *refinement* module, and checks whether it is safe or not. If the *model checker* returns that the program is safe, then $\mathcal{P}$ is also safe, and our algorithm terminates. If the program is unsafe, then the *model checker* returns a counter-example π' of the form $c_0 \cdot \mathsf{loc}_1 \cdot c_1 \cdot \mathsf{loc}_2 \cdots \mathsf{loc}_m \cdot c_m$.

The *counter-example mapping* takes the counter-example π' as its input. It transforms the run π' to a run of the program resulting of the *abstraction* module.

The *reconstruction* takes always as input a counter-example π of $\mathcal{P}'$ (which results from the application of the *abstraction* module to the program $\mathcal{P}$). It then checks whether π can lead to a real counter-example of $\mathcal{P}$. In particular, if $\mathsf{V}_0 = \mathsf{V}_{\mathcal{P}}$, i.e. no variable was sliced away from $\mathcal{P}$, then $\mathcal{P}'$ is identical to $\mathcal{P}$. Therefore, π is also a counter-example of $\mathcal{P}$. The algorithm concludes that $\mathcal{P}$ is unsafe, and then terminates. Otherwise, the *reconstruction* adds back all omitted variables (i.e., $V_{\mathcal{P}} \setminus V_0$) and lines of codes to create a program $\mathcal{P}_1$. The program $\mathcal{P}_1$ also needs to respect the flow of the instructions in π. In other words, the instruction labeled by loc_i, for some $i \in \{1, \ldots, m\}$, in $\mathcal{P}_1$ can only be executed after executing all the instructions labeled by loc_j for all $j \in \{1, \ldots, i-1\}$. For each run of the program $\mathcal{P}_1$, let c'_i be the configuration after executing the instruction labeled by loc_i. The configuration c'_i needs to satisfy $\mathsf{StateOf}(c'_i)(x) = \mathsf{StateOf}(c_i)(x)$ for all $x \in \mathsf{V}_0$, i.e. each value of variable in the set V_0 at the configuration c'_i is equal to its value in the configuration c_i.

Then CEGPV recursively calls itself to check $\mathcal{P}_1$ in its next iteration. Inputs of the next iteration are $\mathcal{P}_1$, and a subset of variables $\mathsf{V}_1 \subseteq \mathsf{V}_{\mathcal{P}_1} = (V_{\mathcal{P}} \setminus V_0)$, which is selected using the dependency graph. If the iteration returns that $\mathcal{P}_1$ is unsafe, then the run π leads to a counter-example of $\mathcal{P}$. The algorithm concludes that $\mathcal{P}$ is unsafe and terminates. Otherwise, π cannot lead to a counter-example of $\mathcal{P}$. Then the algorithm needs to discard π from the set of runs of $\mathcal{P}'$.

The *refinement* adds π to the set of spurious counter-examples of $\mathcal{P}'$ (resulting from the application of the *abstraction* module to $\mathcal{P}$). It then refines $\mathcal{P}'$ by removing all these spurious counter-examples from the set of runs of $\mathcal{P}'$. The new resulting program is then given back to the *model checker*.

In the following, we explain in more details each module of our CEGPV algorithm. The *counter-example mapping* module is described at the end of the subsection dedicated to the explanation of the *refinement* module (Section 4.3).

4.1 The Abstraction

The abstraction transforms $\mathcal{P}$ into a new program $\mathcal{P}'$ by slicing away all variables in $\mathsf{V}_{\mathcal{P}} \setminus \mathsf{V}_0$ and some lines of codes. In particular, we define a map function $[\![.]\!]_{ab}$

that rewrites $\mathcal{P}$ into $\mathcal{P}'$. The formal definition of the map $[\![.]\!]_{ab}$ is given in Fig. 10. In the following, we informally explain $[\![.]\!]_{ab}$.

The map $[\![.]\!]_{ab}$ keeps only the variables in V_0 and removes all other variables of $\mathcal{P}$. The map $[\![.]\!]_{ab}$ also keeps the same number of processes as in $\mathcal{P}$, and transforms the code of each process of $\mathcal{P}$ to a corresponding process in $\mathcal{P}'$.

For each instruction in a process, the map $[\![.]\!]_{ab}$ keeps the label and transforms the statement in that instruction. The map $[\![.]\!]_{ab}$ replaces occurrences of sliced variables in the statement by the nondeterministic value $*$. First, the skip and goto statements remain the same since they do not make use of any variable. Second, for an assignment statement of the form "$x := expr$", if the variable x is not in V_0, then that statement is transformed to the skip statement. If at least one discarded variable occurs in the expression $expr$, then the assignment is transformed to "$x := *$". Otherwise, the assignment remains the same. Third, for both an assumption statement of the form "`assume` $expr$" and an assertion of the form "`assert` $expr$", the map $[\![.]\!]_{ab}$ replaces the expression $expr$ by the nondeterministic value $*$, if at least one discarded variable occurs in $expr$. Otherwise, the assumption and assertion remain the same. For a conditional statement, the map $[\![.]\!]_{ab}$ transforms its guard to be non-deterministic if it makes use of one of the discarded variables. The consequent instruction and alternative instruction are also transformed in a similar manner by the map $[\![.]\!]_{ab}$. Finally, we remove any instruction that *trivially* does not affect the behaviors of $[\![\mathcal{P}]\!]_{ab}$.

$$
\begin{array}{rcl}
[\![\langle c\text{-}prog\rangle]\!]_{ab} & \stackrel{\text{def}}{=} & [\![\langle \texttt{var } x\rangle]\!]^{+}_{ab}[\![\langle process\rangle]\!]^{+}_{ab} \\
[\![\langle \texttt{var } x\rangle]\!]_{ab} & \stackrel{\text{def}}{=} & \begin{cases} \texttt{var } x\texttt{;} & \text{if } x \in \mathsf{V}_0 \\ \cancel{\texttt{var } x\texttt{;}} & \text{otherwise} \end{cases} \\
[\![\langle process\rangle]\!]_{ab} & \stackrel{\text{def}}{=} & \texttt{process p begin } [\![\langle inst\rangle]\!]^{*}_{ab} \texttt{ end} \\
[\![\langle inst\rangle]\!]_{ab} & \stackrel{\text{def}}{=} & \mathsf{loc}\texttt{:}\quad [\![\langle stmt\rangle]\!]_{ab}\texttt{;} \\
[\![\texttt{skip}]\!]_{ab} & \stackrel{\text{def}}{=} & \texttt{skip} \\
[\![\texttt{goto } \mathsf{loc}_1,\ldots,\mathsf{loc}_n]\!]_{ab} & \stackrel{\text{def}}{=} & \texttt{goto } \mathsf{loc}_1,\ldots,\mathsf{loc}_n \\
[\![\langle x := \langle expr\rangle]\!]_{ab} & \stackrel{\text{def}}{=} & \begin{cases} \texttt{skip} & \text{if } x \notin \mathsf{V}_0 \\ x := [\![\langle expr\rangle]\!]_{ab} & \text{otherwise} \end{cases} \\
[\![\texttt{assume}\langle expr\rangle]\!]_{ab} & \stackrel{\text{def}}{=} & \texttt{assume } [\![\langle expr\rangle]\!]_{ab} \\
[\![\texttt{assert}\langle expr\rangle]\!]_{ab} & \stackrel{\text{def}}{=} & \texttt{assert } [\![\langle expr\rangle]\!]_{ab} \\
[\![\texttt{if } \langle expr\rangle \texttt{ then } \langle inst_1\rangle \texttt{ else } \langle inst_2\rangle \texttt{ fi}]\!]_{ab} & \stackrel{\text{def}}{=} & \texttt{if } [\![\langle expr\rangle]\!]_{ab} \texttt{ then } [\![\langle inst_1\rangle]\!]_{ab} \texttt{ else } [\![\langle inst_2\rangle]\!]_{ab} \texttt{ fi} \\
[\![\langle expr\rangle]\!]_{ab} & \stackrel{\text{def}}{=} & \begin{cases} * & \text{if } \mathsf{Var}_{expr}(expr) \cap (\mathsf{V}_{\mathcal{P}_0} \setminus \mathsf{V}_0) \neq \varnothing \\ expr & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 10: Translation map $[\![.]\!]_{ab}$

Lemma 1. *If* $[\![\mathcal{P}]\!]_{ab}$ *is safe, then* $\mathcal{P}$ *is safe.*

4.2 The Reconstruction

Let π be a counter-example of the program $[\![\mathcal{P}]\!]_{ab}$ of the form $c_0 \cdot \mathsf{loc}_1 \cdot c_1 \cdot \mathsf{loc}_2 \cdots \mathsf{loc}_m \cdot c_m$. The reconstruction transforms $\mathcal{P}$ to a new program $\mathcal{P}_1$ by forcing $\mathcal{P}$ to respect the sequence of configurations and labels in π. In particular, we define a map function $[\![.]\!]_{co}$ to rewrite the program $\mathcal{P}$ into the program $\mathcal{P}_1$. The formal definition of the map $[\![.]\!]_{co}$ is given in Fig 11. For a label loc, let $\mathsf{IndexOf}(\mathsf{loc}) = \{i \in \{1,\ldots,m\} \mid \mathsf{loc}_i = \mathsf{loc}\}$ be the set of positions where the label loc occurs in the run π. Let newloc be a function that returns a fresh label that has not used so far. The map $[\![.]\!]_{co}$ starts by adding a new variable cnt. The variable cnt is used to keep track of the execution order of the instructions in π. All variables in V_0 are removed by the map $[\![.]\!]_{co}$ since their values is determined

$$
\begin{array}{rcl}
[\![\langle \textit{c-prog}\rangle]\!]_{co} & \stackrel{\text{def}}{=} & \texttt{var } cnt; [\![\langle \texttt{var } x\rangle]\!]^{+}_{co} [\![\langle \textit{process}\rangle]\!]^{+}_{co} \\
[\![\langle \texttt{var } x\rangle]\!]_{co} & \stackrel{\text{def}}{=} & \begin{cases} \texttt{var } x; \text{ if } x \notin \mathsf{V}_0 \\ \cancel{\texttt{var } x;} \text{ otherwise} \end{cases} \\
[\![\langle \textit{process}\rangle]\!]_{co} & \stackrel{\text{def}}{=} & \texttt{process p begin } [\![\langle \textit{inst}\rangle]\!]_{co} \texttt{ end} \\
[\![\langle \textit{inst}\rangle]\!]_{co} & \stackrel{\text{def}}{=} & \begin{cases} [\![\mathsf{loc} : \langle \textit{stmt}\rangle]\!]_{co,ab} \quad \text{if } \mathsf{loc} \in \mathbb{I}_{[\![\mathcal{P}]\!]_{ab}} \\ [\![\mathsf{loc} : \langle \textit{stmt}\rangle]\!]_{co,oth}; \text{ otherwise} \end{cases} \\
[\![\mathsf{loc} : \langle \textit{stmt}\rangle]\!]_{co,oth} & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{loc}\texttt{: if } (cnt == 0) \texttt{ then } [\![\langle \textit{stmt}\rangle]\!]^{0}_{co,oth}\texttt{; else} \\ \quad \ldots \\ \quad \texttt{if } (cnt == m) \texttt{ then } [\![\langle \textit{stmt}\rangle]\!]^{m}_{co,oth}\texttt{;} \\ \quad \texttt{else skip; fi;} \ldots \texttt{fi;} \end{cases} \\
[\![\mathsf{loc} : \langle \textit{stmt}\rangle]\!]_{co,ab} & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{loc}\texttt{: if } (cnt + 1 \in \mathsf{IndexOf}(\mathsf{loc}) \wedge cnt == 0) \texttt{ then} \\ \quad [\![\langle \textit{stmt}\rangle]\!]^{0}_{co,ab}\texttt{; else} \\ \quad \ldots \\ \quad \texttt{if } (cnt + 1 \in \mathsf{IndexOf}(\mathsf{loc}) \wedge cnt == m - 1) \texttt{ then} \\ \quad [\![\langle \textit{stmt}\rangle]\!]^{m-1}_{co,ab}\texttt{; else assume false; fi;} \ldots \texttt{fi;} \\ \mathsf{newloc} : cnt := cnt + 1; \end{cases} \\
[\![\texttt{skip}]\!]^{i}_{co,-} & \stackrel{\text{def}}{=} & \texttt{skip where } - \in \{ab, oth\} \\
[\![\texttt{goto } \mathsf{loc}_1, \ldots, \mathsf{loc}_n]\!]^{i}_{co,-} & \stackrel{\text{def}}{=} & \texttt{goto } \mathsf{loc}_1, \ldots, \mathsf{loc}_n \text{ where } - \in \{ab, oth\} \\
[\![\texttt{assume } \langle \textit{expr}\rangle]\!]^{i}_{co,-} & \stackrel{\text{def}}{=} & \texttt{assume } [\![\langle \textit{expr}\rangle]\!]^{c}_{co} \text{ where } - \in \{ab, oth\} \\
[\![\texttt{assert } \langle \textit{expr}\rangle]\!]^{i}_{co,-} & \stackrel{\text{def}}{=} & \texttt{assert } [\![\langle \textit{expr}\rangle]\!]^{c}_{co} \text{ where } - \in \{ab, oth\} \\
[\![x := \langle \textit{expr}\rangle]\!]^{i}_{co,ab} & \stackrel{\text{def}}{=} & \texttt{assume } \mathsf{StateOf}(c_{i+1})(\mathrm{x}) == [\![\langle \textit{expr}\rangle]\!]^{i}_{co} \\
[\![x := \langle \textit{expr}\rangle]\!]^{i}_{co,oth} & \stackrel{\text{def}}{=} & x := [\![\langle \textit{expr}\rangle]\!]^{i}_{co} \\
[\![\texttt{if } \langle \textit{expr}\rangle \texttt{ then } \langle \textit{inst}_1\rangle \texttt{ else } \langle \textit{inst}_2\rangle \texttt{ fi}]\!]^{i}_{co,ab} & \stackrel{\text{def}}{=} & \begin{cases} \texttt{assume } [\![\langle \textit{expr}\rangle]\!]^{i}_{co} == \texttt{true; } [\![\langle \textit{inst}_1\rangle]\!]_{co} & \text{if } \textit{label}(\textit{inst}_1) \in \mathsf{LabelOf}(c_{i+1}) \\ \texttt{assume } [\![\langle \textit{expr}\rangle]\!]^{i}_{co} == \texttt{false; } [\![\langle \textit{inst}_2\rangle]\!]_{co} & \text{otherwise} \end{cases} \\
[\![\texttt{if } \langle \textit{expr}\rangle \texttt{ then } \langle \textit{inst}_1\rangle \texttt{ else } \langle \textit{inst}_2\rangle \texttt{ fi}]\!]^{i}_{co,oth} & \stackrel{\text{def}}{=} & \texttt{if } [\![\langle \textit{expr}\rangle]\!]^{i}_{co} \texttt{ then } [\![\langle \textit{inst}_1\rangle]\!]_{co} \texttt{ else } [\![\langle \textit{inst}_2\rangle]\!]_{co} \texttt{ fi} \\
[\![\langle \textit{expr}\rangle]\!]^{i}_{co} & \stackrel{\text{def}}{=} & \langle \textit{expr}\rangle[\forall x \in \mathsf{V}_0 : x \hookleftarrow \mathsf{StateOf}(c_i)(x)]
\end{array}
$$

Fig. 11: Translation map $[\![.]\!]_{co}$

by π. The map $[\![.]\!]_{co}$ also keeps the same number of processes as in the program $\mathcal{P}$, and transforms the code of each process.

The map $[\![.]\!]_{co}$ transforms instructions in each process as follows. Instructions that occur in $[\![\mathcal{P}]\!]_{ab}$, are transformed by the map $[\![.]\!]_{co,ab}$, while other instructions are transformed by the map $[\![.]\!]_{co,oth}$. For an instruction of the form "$\mathsf{loc} : \textit{stmt}$", the map $[\![.]\!]_{co,oth}$ keeps the label loc and creates $m+1$ copies of the statement $\textit{stmt}$. The i-th copy of $\textit{stmt}$, with $i \in \{0, \ldots, m\}$, is executed after reaching the configuration c_i in the run π. Therefore, the i-th copy only can be only executed under the condition "$cnt == i$". Then, the statement $\textit{stmt}$ is transformed based on the configuration c_i in the run π, denoted by $[\![.]\!]^{i}_{co,oth}$. Similarly, the map $[\![.]\!]_{co,ab}$ keeps the label loc and creates m copies of the statement $\textit{stmt}$ (which corresponds to number of instructions in the run π). The i-th copy of $\textit{stmt}$, with $i \in \{1, \ldots, m\}$, is executed if the label loc appears at position i in the run π. Therefore, the i-th copy can be executed under the condition "$cnt + 1 \in \mathsf{IndexOf}(\mathsf{loc})$" (i.e., the label loc appears at the position $cnt + 1$) and that $cnt = i - 1$ (i.e., after reaching the configuration c_{i-1}). Then, the map $[\![.]\!]_{co,ab}$ transforms the statement $\textit{stmt}$ based on the configurations c_{cnt-1} and c_{cnt} (i.e, the configurations before and after executing the instruction labeled by loc) in the run π, denoted by $[\![.]\!]^{cnt}_{co,ab}$. The variable cnt is then increased by one to denote that one more instruction in the run π has been executed.

In general, the map $[\![.]\!]^i_{co,ab}$, for some $i \in \{0,\ldots,m-1\}$ rewrites all expressions in statements. The skip and goto statement remain the same. For both an assertion of the form "`assert` *expr*" and assumption "`assume` *expr*", $[\![.]\!]^c_{co,ab}$ transforms their expressions *expr*. For an assignment of the form "$x := expr$", it rewrites that assignment by an assumption checking that, the value of x in the configuration c_{i+1} is equal to the value of *expr* at the configuration c_i. For a conditional statement of the form "`if` $\langle expr \rangle$ `then` $inst_1$ `else` $inst_2$ `fi`", $[\![.]\!]^c_{co,ab}$, we first check which branch has been taken in the run π. To do that, we check the labels appearing in the configuration c_{i+1}. After that, we add an assumption to check whether the branch has been correctly selected in the counter-example. if *expr* is evaluated to `true` at the configuration c_i and the label of $inst_1$ appears at the configuration c_{i+1}, then it executes the instruction $[\![inst_1]\!]^i_{co,ab}$. Otherwise, it executes the instruction $[\![inst_2]\!]^i_{co,ab}$. Finally, all occurrences of variables in V_0 in any expressions *expr* are replaced by their values in the configuration c_i.

The map $[\![.]\!]^i_{co,oth}$, for some $i \in \{0,\ldots,m\}$, transforms statements as follows. The skip and goto statement remain the same. For assignment, assumption, and assertion, $[\![.]\!]^i_{co,oth}$ rewrites expressions in these statements. For a conditional statement, it also rewrites the guards, the consequent instruction and the alternative instruction. The expression is transformed by replacing occurrences of variables in V_0 in that expression by their values in the configuration c_i.

Lemma 2. *If* $[\![\mathcal{P}]\!]_{co}$ *is unsafe, then* $\mathcal{P}$ *is unsafe.*

4.3 The Refinement

Given a set of runs R of $[\![\mathcal{P}]\!]_{ab}$, the *refinement* module constructs a program $\mathcal{P}'$ from $[\![\mathcal{P}]\!]_{ab}$ by discarding the set of runs in R from the set of runs of $[\![\mathcal{P}]\!]_{ab}$. Before giving the details of this module, we introduce some notations and definitions.

For a run π of the form $c_0 \cdot \mathsf{loc}_1 \cdot c_1 \ldots \mathsf{loc}_m \cdot c_m$, let $\mathsf{Loc}(\pi) = \{\mathsf{loc}_1,\ldots,\mathsf{loc}_m\}$ be the set of all labels occurring in π, and $\mathsf{Con}(\pi) = \{c_0, c_1, \ldots, c_m\}$ be the set of all configurations in π. Let $\mathsf{R}_{loc} = \bigcup_{\pi\in\mathsf{R}} \mathsf{Loc}(\pi)$ and $\mathsf{R}_{con} = \bigcup_{\pi\in\mathsf{R}} \mathsf{Con}(\pi)$. Let $\mathsf{Prefix}(\pi) = \{c_0 \cdot \mathsf{loc}_1 \cdot c_1 \ldots \mathsf{loc}_i \cdot c_i | i \in \{0,\ldots,m-1\}\}$ be the set of prefixes of π and $\mathsf{R}_{prefix} = \bigcup_{\pi\in\mathsf{R}} \mathsf{Prefix}(\pi)$ be the set of all prefixes of all runs in R.

Then, we construct a graph (or a tree) G_R to represent in concise manner the set of runs in R. The graph $G_R = (V, E)$ consists of a number of vertices V and directed edges E where $V = \mathsf{R}_{prefix}$ and $E = \{(v, v') | \exists \mathsf{loc} \in \mathsf{R}_{loc}, c \in \mathsf{R}_{con}$ and $v' = v \cdot \mathsf{loc} \cdot c\}$. In other words, each vertex corresponds to a prefix in R_{prefix}, and each edge describes the transition from one prefix to another one.

Let $v \in V$, $P \in \mathsf{Proc}_{[\![\mathcal{P}]\!]_{ab}}$, and $\mathsf{loc} \in \mathbb{Q}_P$. Let $\mathsf{Next}(v, \mathsf{loc}) = \{c | c \in \mathsf{R}_{con} : v \cdot \mathsf{loc} \cdot c \in (V \cup \mathsf{R})\}$ be the function that returns the set of configurations which can be reached from v through executing the instruction labeled by loc. Let $\mathsf{Reach}(v, P) = \{\mathsf{loc} | \mathsf{loc} \in \mathbb{Q}_P, \exists c \in C([\![\mathcal{P}]\!]_{ab})$ and $\exists v' \in \Pi([\![\mathcal{P}]\!]_{ab}) : (v' = v \cdot \mathsf{LabelOf}(\mathit{Target}(v))(P) \cdot c) \wedge (v' \notin (V \cup \mathsf{R})) \wedge (\mathsf{loc} = \mathsf{LabelOf}(c)(P))\}$ be the function that returns the set of all possible labels loc of the process P that can be reached by a run $v' \notin \mathsf{R} \cup V$ which is an extension of the prefix v by

executing an instruction of the process P. In order to force the execution of $[\![\mathcal{P}]\!]_{ab}$ to perform a different run than the ones in R, we make sure that $[\![\mathcal{P}]\!]_{ab}$ follows the prefix $v \in \mathsf{R}_{prefix}$, and then performs the instruction of the process P that leads to a new prefix p' which was not part of R_{prefix} or R. Then, we create the output program $\mathcal{P}'$ of the *refinement* module from $[\![\mathcal{P}]\!]_{ab}$ by adding (1) an *observer* process to simulate the execution of the prefix v', and (2) a *controller* per process to continue execution of each process from the reached location after executing the prefix v'. We add a new variable, called *label*, used by the *observer* to communicate to each *controller* where the execution will resume for each process after leaving the *observer*.

We construct an *observer* as given in Fig. 12. The *observer* is executed before any processes in $[\![\mathcal{P}]\!]_{ab}$. It starts by non-deterministically jumping to a node v_i (representing a prefix of a run in R), where v_i represents a vertex of G_R. At the node v_i, values of variables are updated to the valuation at $Target(v_i)$. Then, the observer decides, in non-deterministic manner, to execute an instruction of a process $P_j \in [\![\mathcal{P}]\!]_{ab}$. If the execution of an instruction of P_j, from the prefix v_i, does not lead a new prefix which is not in $\mathsf{R} \cup \mathsf{R}_{prefix}$ (i.e., $\mathsf{Reach}(v, P_j)$ is empty), then the execution of the *observer* terminates (and so of the program $\mathcal{P}'$). If $\mathsf{Reach}(v, P_j)$ is not empty, we first distinguish the case where the next instruction to be executed by P_j is a non-deterministic assignment to some variable x. Then, the observer ensures that the new value assigned to x is different from its value in any configuration which can be reached from v_i through executing this non-deterministic assignment by P_j. After that, the *observer* communicates the new label of P_j by setting the variable *label* to it. Finally, it sets the variable $flag$ to one to enable the execution of other processes and communicates to them their starting instruction by setting the variable *label*.

```
start: goto v_1, v_2, ..., v_n;
...
v_i:  for all x ∈ V_0: x := StateOf(Target(v_i))(x);
      goto (v_i, P_1), ..., (v_i, P_m);
      ...
      (v_i, P_j): if Reach(v_i, P_j) ≠ ∅ then
        loc := LabelOf(Target(v_i))(P_j);
        if stmt(loc) of the form "x := *" then
          x := *;
          assume x ∉ {StateOf(c)(x) | c ∈ Next(v_i, loc)};
        else assume false ; fi;
        label := *;
        assume label ∈ Reach(v_i, P_j);
        flag := 1;
        for all P ∈ Proc_[[P]]_ab \ {P_j}
          label := LabelOf(Target(v_i)) (P);
      fi;
      assume false ;
      ...
...
```

Fig. 12: Pseudocode of Observer with $V = \{v_1, \ldots, v_n\}$ and $\mathsf{Proc}_{[\![\mathcal{P}]\!]_{ab}} = \{P_1, \ldots, P_m\}$

Each process P in $[\![\mathcal{P}]\!]_{ab}$ is controlled by a controller, given in Fig. 13. The controller is placed at the top of the code of P. The controller then checks if the label stored in the variable *label* is in indeed belongs to P, if it is the case, it jumps to that label. Otherwise, P needs to wait until one of its label is written.

```
assume flag == 1;
if label ∈ Q_P then goto label;
else assume false ;
...
```

Fig. 13: Pseudocode of Controller of the process P

Finally, we can easily define a mapping *map* that maps any run of $\mathcal{P}'$ to a run of $[\![\mathcal{P}]\!]_{ab}$. This mapping *map* is used in the *Counter-example mapping* module. We can also extend the definition of the mapping *map* to sets of runs in the straightforward manner.

Lemma 3. $map(\Pi(\mathcal{P}')) = \Pi([\![\mathcal{P}]\!]_{ab}) \setminus R$.

5 Optimizations

In this section, we present two optimizations of our CEGPV algorithm. The first optimization concerns the reduction of the number of iterations of our GEGPV algorithm by considering several counter-examples instead of one at each iteration. The second optimization concerns an efficient implementation of the *reconstruction* and *refinement* modules when considering SMT/SAT based model-checkers such as CBMC [10].

Combining counter-examples. Our *reconstruction* module takes as input a counter-example π of the form $c_0 \cdot \mathsf{loc}_1 \cdot c_1 \cdot \mathsf{loc}_2 \cdots \mathsf{loc}_m \cdot c_m$ of the program $[\![\mathcal{P}]\!]_{ab}$, and construct the program $\mathcal{P}_1$ which needs to respect the flow of the instructions in π and also the evaluation of the set of shared variables in V_0. To do so efficiently, we drop the constraint that the program $\mathcal{P}_1$ should follow the valuations of the shared variables in V_0 in our code-code translation $[\![.]\!]_{co}$. This means that the constructed program $\mathcal{P}_1$ should only make sure to execute the instruction labeled by loc_i, for some $i \in \{1, \ldots, m\}$, after executing all the instructions labeled by loc_j for all $j \in \{1, \ldots, i-1\}$. We also modify the *refinement* module to discard all the runs π' in the set of runs of $[\![\mathcal{P}]\!]_{ab}$ such that $\mathsf{Trace}(\pi') = \mathsf{Trace}(\pi)$ in case that the program $\mathcal{P}_1$ is declared safe by *model-checker*.

We can furthermore optimize our CEGPV algorithm by not imposing any order on the execution of two instructions labeled by loc_i and loc_j if they can be declared to be independent (as done in stateless model-checking techniques [3])

SMT based optimization. The CEGPV algorithm can be integrated into SMT/SAT based model-checkers such as CBMC [10]. Recall that in Section 4.2, we force a program running in a specific order of instructions, and in Section 4.3, we forbid that order of instructions in a program. These operations can be easily done performed using clock variables [16]. Indeed, for each label loc in the program, we associate to a clock variable $\mathsf{clock}_{\mathsf{loc}}$ ranging over the naturals. The clock variable $\mathsf{clock}_{\mathsf{loc}}$ is assigned 0 if the instruction labeled by loc is not executed. Given labels loc_1 and loc_2, in order to force the execution of the instruction labeled by loc_1 before the execution of the instruction labeled by loc_2, we need only to make sure that $0 < \mathsf{clock}_{\mathsf{loc}_1}$ and $\mathsf{clock}_{\mathsf{loc}_1} < \mathsf{clock}_{\mathsf{loc}_2}$. In the similar way, we can write a formula to force the SMT/SAT based model checker to return a counter-example different from the already encountered ones.

		CBMC 5.1			CEGPV		
sub-catergory	#programs	pass	fail	time	pass	fail	time
pthread-wmm-mix-unsafe	466	466	0	40301	466	0	1076
pthread-wmm-podwr-unsafe	16	16	0	286	16	0	21
pthread-wmm-rfi-unsafe	76	76	0	958	76	0	141
pthread-wmm-safe-unsafe	200	200	0	12578	200	0	917
pthread-wmm-thin-unsafe	12	12	0	252	12	0	15
pthread-unsafe	17	12	5	441	17	0	302
pthead-atomic-unsafe	2	2	0	2	2	0	2
pthread-ext-unsafe	8	4	4	7	8	0	7
pthread-lit-unsafe	3	2	1	3	2	1	2
pthread-wmm-rfi-safe	12	12	0	3154	12	0	138
pthread-wmm-safe-safe	104	102	2	352	104	0	114
pthread-wmm-thin-safe	12	12	0	28	12	0	12
pthread-safe	14	7	7	124	13	1	63
pthead-atomic-safe	8	7	1	76	8	0	10
pthread-ext-safe	45	19	26	938	31	14	569
pthread-lit-safe	8	3	5	8	3	5	5

Table 1: Performance of CEGPV in comparison to CBMC on benchmarks of the SV-COMP15 *Concurrency category* [2]. Each row corresponds to a sub-category of the SV-COMP15 benchmarks, where we report the number of checked programs. The column *pass* gives the number of correct answers retuned by each tool. An answer is considered to be correct for a (un)safe program if the tool return "(un)safe". The columns $fail$ report the number of unsuccessful analyses performed by each tool. An unsuccessful analysis includes crashes, timeouts. The columns *time* gives the total running time in seconds for the verification of each benchmark. Observe that we do not count, in the total time, the time spent by a tool when the verification fails.

6 Experiment Results

In order to evaluate the efficiency of our CEGPV algorithm, we have implemented it as a part of an open source tool, called CEGPV [1], for the verification of C/pthreads programs. We used CBMC version 5.1 as a backend tool [10]. We then evaluated CEGPV on the benchmark set from the Concurrency category of the TACAS Software Verification Competition (SV-COMP15) [2]. The set consists of 1003 C programs. We have performed all experiments on an Intel Core i7 3.5Ghz machine with 16GB of RAM. We have used a 10GB as memory limit and a 800s as timeout parameter for the verification of each program.

In the following, we present two sets of results. The first part concerns the unsafe programs and the second part concerns safe ones. In both parts, we compare CEGPV results to the ones obtained using CBMC 5.1 tool [10]. To ensure a faire comparison between the two tools, we use the same loop-unwinding and thread duplication bounds for each program. Table 1 shows that CEGPV is highly competitive. We observe that, for unsafe programs, CEGPV significantly outperforms CBMC. CEGPV is more than 10 times faster (on average) than CBMC, except for few small programs. CEGPV also manages to verify almost

all the unsafe benchmarks (except one) while CBMC fails in the verification of 10 programs due to timeout. For safe benchmarks, CEGPV still outperforms CBMC in the running time. In many programs, CEGPV succeeds to prove the safety of several programs (except 20 programs), while CBMC fails to prove the safety of 41 programs. Finally, we observe that, for the benchmark $pthread-lit$, the results of both tools are almost the same. The reason is that the programs in that benchmark only use few variables. Therefore, CEGPV does not slice away variables in these programs.

References

1. CEGPV. https://github.com/diepbp/SlicingCBMC
2. SV-COMP home page. http://sv-comp.sosy-lab.org/2015/
3. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.F.: Optimal dynamic partial order reduction. In: POPL. pp. 373–384. ACM (2014)
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL. pp. 1–3. ACM (2002)
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. STTT 9(5-6), 505–525 (2007)
6. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: CAV. LNCS, vol. 6806, pp. 184–190. Springer (2011)
7. Bindal, S., Bansal, S., Lal, A.: Variable and thread bounding for systematic testing of multithreaded programs. In: ISSTA. pp. 145–155. ACM (2013)
8. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Software Eng. 30(6), 388–402 (2004)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
10. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. pp. 58–70. ACM (2002)
12. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. In: CAV. LNCS, vol. 8044, pp. 846–862. Springer (2013)
13. Kurshan, R.P.: Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach. Princeton University Press (1994)
14. Kurshan, R.P.: Program verification. In: Notices of the AMS 47(5). pp. 534–545 (2000)
15. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: CAV. LNCS, vol. 7358, pp. 427–443. Springer (2012)
16. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
17. Saïdi, H.: Model checking guided abstraction and analysis. In: SAS. LNCS, vol. 1824, pp. 377–396. Springer (2000)
18. Valdiviezo, M., Cifuentes, C., Krishnan, P.: A method for scalable and precise bug finding using program analysis and model checking. In: APLAS. LNCS, vol. 8858, pp. 196–215. Springer (2014)