УДК 004.415.52+519.681

Enhancing Verification Condition Generation for Reflex Programs Through Simple Static Analysis[*]

Ishchenko A.D. (Institute of automatics and electrometry, Siberian Branch of the Russian Academy of Sciences)

Control systems play a crucial role in various domains necessitating high reliability and proof of correct software operation. To address this, formal methods, such as deductive verification, are employed to ensure the correctness of safety-critical software mathematically. This process involves generating verification conditions from the program's axiomatic semantics and its requirements. Main disadvantage of manual verification condition generation is its labor-intensity and prone to human error, which leads to the development of automated verification condition generators. However, they produce excessive or overly complex verification conditions that do not accurately reflect the possible program's operational paths. To mitigate the second issue, domain-oriented languages like Reflex have been introduced, which adopt a process-oriented paradigm to streamline programming and simplify verification condition generation. Nevertheless, the inherent switch-case structure of Reflex can lead to an increase in the number of verification conditions, exacerbating the first issue and complicating the selection of relevant ones. This paper proposes a simple static analysis system designed to optimize the generation of verification conditions for Reflex programs, enhancing the efficiency of the verification process. It is based on attaching attributes to different statements. Then, throughout the verification condition generation these attributes are collected and checked to be compatible with previously collected. If attributes are incompatible then verification condition are discarded.

Key words: process-oriented, static analysis, attributes, deductive verification

1. Introduction

Control systems are widely used in many areas: from the Internet of Things to programmable industrial controllers. This requires advanced reliability for such systems and, in particular, proof of correct operation of used software.

Nowadays, the most common method to ensure correctness is to test the system on digital models or real-world control objects. It is popular for its simplicity and relative cheapness. However, this method is incomplete because it does not cover all possible cases. This leads to

[*] This work was supported by the Russian Ministry of Education and Science, project no. 122031600173-8.

a danger of appearing of rare errors among widely used devices which looked completely safe.

To ensure the correctness of safety-critical software, formal methods are used. They use formal semantics of the program and its requirements to simplify this challenge by transforming it into some mathematical problem.

One of these formal methods is deductive verification [5]. It requires defining the axiomatic semantics of programming language in some logical inference system and formalizing requirements. Then program and its requirements are transformed into a set of logical formulas. This process is called verification condition (VC) generation. Doing it manually is time-consuming and dangerous because of human factors. To automate this process, verification condition generators are developed. A generator goes through all possible paths of control flow and creates VCs for them.

However, the naive realization of the generator may create a lot of excess verification conditions that do not represent the ways the program really could work. Therefore, it is important to find ways for reducing the amount of VCs and their simplification.

For simplification of creating and verification of control software special domain-oriented languages are used. One of them is Reflex language [10] created for writing control programs. It is designed in the style of a process-oriented paradigm. This paradigm represents the program as a list of processes that are executed sequentially once in a certain period called activation time thus forming a control loop. Each process consists of several active states created by the programmer and two inactive states: stop and error. Each state is a list of instructions that include both ordinary imperative statements (for example, conditional statements) and special process-oriented statements designed to interact with other processes. The use of the Reflex language rebuilds the programming style for control programs, which leads, in particular, to the definition of VCs in terms of control processes and their states and thus makes them conceptually simpler. However, this also leads to an increase in the number of VCs caused by the switch-case nature of processes of a Reflex program, and some of these VCs are 'false-positive' since they correspond to the paths of the program that are non-reachable in its actual execution.

In this paper, we propose a static analysis system for Reflex programs that finds such VCs to optimize the process of VCs generation.

2. Reflex language

A Reflex program consists of processes, process states, process-oriented statements and C-like statements.

For interaction with processes and their states, the following instructions are defined:

- *restart*, *start p* – sets the current process or process p into its first state;
- *stop*, *stop p* – sets current process or process p into stop state;
- *error*, *error p* – sets the current process or process p into error state;
- *set state s*, *set next state* – sets the current process into state s or state defined after the current one;
- *reset timer* – set local time of the current process to 0;
- *process p in state k* – checks whether process p is in state kind $k \in \{$*active*, *inactive*,*stop*,*error*$\}$.

Besides them, the following C-like constructs are used:

- C-like expressions;
- if-else statements;
- switch-case statements.

3. Static analysis

An increase in the number of VCs occurs in processing the following statements:

- *process ...* – enumeration of different process states;
- *timeout t ...* – checking cases when timeout t exceeded and when not;
- *if (expr) ... else ...* – checking cases when `expr` is true or false;
- *switch (expr) {case ...}* – checking when `expr` corresponding to different cases;

In this paper we focus on the first two cases, and for the second one only analyze cases when timeout parameter t is presented by a constant value. For example (Fig. 1), if in the process of VC generation some process A starts another process B that contextually appears after the first one, then, when process B will be processed, it must be in its first state.

Before the analysis program is transformed into canonical form with the conversion of short forms of operators into full ones:

- *restart* $\longrightarrow$ *start* **p** where **p** is the current process;
- *stop* $\longrightarrow$ *stop* **p** where **p** is the current process;
- *error* $\longrightarrow$ *error* **p** where **p** is the current process;
- *set next state* $\longrightarrow$ *set state* **s** where **s** is the next state of the current process.

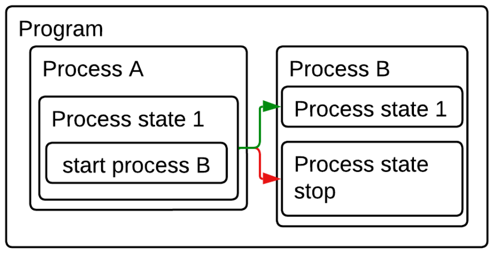

Figure 1: Possible and impossible paths. They are shown by green and red arrows correspondingly

The result of static analysis is a set of attributes. They are used in verification condition generation to avoid the creation of VCs for impossible paths. They are defined for declarations of processes, process states and statements included in the bodies of process state declarations. Let **p**.A means the value of the attribute A of the declaration of process **p**, **p**.**s**.A does the value of attribute A of declaration of the state **s** of process **p**, **p**.**s**.**st**.A does the value of attribute A after execution of the statement **st** of state **s** of process **p**.

The following list describes evaluated attributes:

1. $p.state := s$ – contains the current state of process p;
2. $p.reachE := true/false$ – shows whether process p ever reaches error state;
3. $p.reachS :=$ *true/false* – shows whether process p ever reaches stop state;
4. $p.startS :=$ *true/false* – shows whether process p could start in stop state;
5. $p.s.st.procChange := f$ – contains the partial function from process to $\{start, stop\ or\ error\}$ where, for example, f(p')=start means that process p' was started after execution of statement p of process state p of process p;
6. $p.s.st.reset := true/false$ – shows whether timer was reset;

Attributes $reachE$, $reach$ and $reset$ have *false* value by default. Attribute $startS$ has default value $false$ for the first process and *true* for other processes. Attributes *state* and $procChange$ are not defined initially.

A general static analysis scheme is presented in figure 2. The algorithm takes an abstract syntax tree (AST) of the analyzed program and sets initial attribute values to corresponding statements. Then, the lifting algorithm is applied to evaluate attributes of more general constructions. After that, attributes of more complex properties are set. In conclusion, AST with set attributes and attributes compatibility rules are provided to the VC generation algorithm.

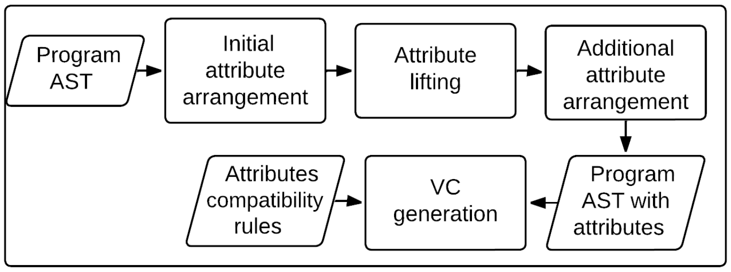
Program AST
Initial attribute arrangement
Attribute lifting
Additional attribute arrangement
Program AST with attributes
VC generation
Attributes compatibility rules

Figure 2: Static analysis scheme

3.1. Initial attribute arrangement

Firstly, we define attributes *start*, *stop*, *error*, *reset*. To do this we traverse AST of a program and set attributes values in accordance on type of statement *st*:

```
if(st ≡ start p′) then
    st.procChange.add(p′,start)
    if (p ≡ p′) then st.reset := true
if(st ≡ stop p′) then
    st.procChange.add(p′,stop)
    if (p′≡p) then st.reset := true
if(st ≡ error p′) then
    st.procChange.add(p′,stop)
    if (p′≡p) then st.reset := true
if(st ≡ reset timer) then st.reset := true
if(st ≡ set state s′) then st.reset := true
```

where s and p are current state and process, respectively, and ≡ denotes syntactic coincidence.

Attributes *reachE* and *reachS* define whether a process ever reaches states *stop* and *error* by being put into them by some processes. So initialization of these attributes is done by the following algorithm:

```
if(∃p′∈processes(r)|∃s∈states(p′)|(error p∈body(s))
    then p.reachE := true)
if(∃p′∈processes(r)|∃s∈states(p′)|(stop p∈body(s))
    then p.reachS := true)
```

where r is a program, processes(r) returns list of processes of r, states(p) returns list of states of process p and body(s) returns a list of all statements of state s.

3.2. Lifting algorithm

Lifting is an algorithm designed for evaluating more general properties by lifting attributes to higher-level statements. Attributes may be lifted to them in two cases: if the attribute belongs to a statement that is part of a linear sequence of statements, or if the attribute appears at all branches of a statement with several paths: *if-else*, *switch-case*.

For a statement sequence *st* lifting is done by the following algorithm:

```
f := ⊥; reset := false;
for st′ in statements(st)
    if (st'.reset) then reset := true;
    f := f ∪ st′.procChange;
st.reset := reset; st.procChange := f;
```

where *statements*(*st*) returns a sequence of statements of compound statement *st*.

For a branching statement *st*, the algorithm has the following form:

```
reset := true; procChange := (first(statements(st))).procChange;
for st′ in statements(st)
    if (!st′.reset) then reset := false;
    procChange := procChange ∩ st′.procChange;
st.reset := reset; st.procChange := procChange;
```

where $first(l)$ returns the first element of the list l.

3.3. Additional attribute arrangement

The definition of the attribute *startS* depends on the value of this attribute in previous processes and the process starting in them. It is done by the following algorithm:

```
for p in processes(r)
    if (∃ p′ ∈ processes(r)| p′.id < p.id ∧ p′.startS = false ∧
        p ∈ first(states(p′)).start) then p.startS := false;
```

where *id* is the field defining the number of the process in the order of definition in the program. In the following example (Fig. 3) processes **P2** and **P4** become with attributes *startS* equal to false because these processes will be start in the first iteration by process **P1**. Process **P3** remains with *startS* equal to true because it is executed before **P4**, so statement $\{startP3\}$ will take effect only on the second iteration.

The value of attribute *state* is defined during the generation of VCs. If the generation algorithm reaches the state s of process p, the attribute *p.state* is set into the s.

3.4. VC generation

After all attributes are set and lifted to the corresponding structure verification condition generation starts. The original VC generation algorithm is described in [6] and realized on GitHub [1]. It traverses the abstract syntax tree building up the list of preconditions *acc* and

[1]https://github.com/bearhug15/ReflexVCG/tree/before_analysis_add

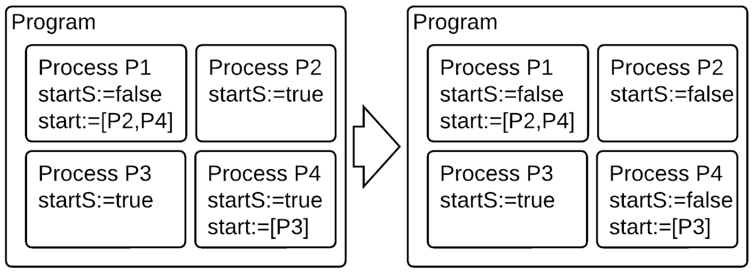

Figure 3: Attributes change due to start attribute

stack of marks made on statements which. Later when the formation of the current VC ends it jumps to the next mark, drops excess preconditions, and continues VC generation from the new place. Usage of attributes updates described algorithm. In addition to the *acc*, a *passed* list is filled. *Passed* describes the passed path as a sequence of language constructs which allows excluding some impossible continuations of this path, based on checking the compatibility of values of attributes of the constructs. After the completion of VC generation and dropping of excess preconditions excess path also dropped.

Updated VC generation function for statement *st* of state *st* of process *p* with checking of VC possibility and filling of *passed* list of statements is done by the following algorithm:

```
if (checkComp(passed, p,s,st)) then
    add(passed, st); updatePrec(st, acc);
else cutToMark(passed, acc);
```

where *add* adds structure *st* into *passed* list, *checkComp* checks whether attributes of *st* are compatible with attributes of structures in *passed* path and return true if they are compatible, *cutToMark* cuts *passed* and *acc* lists to last mark. *updatePrec* updates generated VC based on the existing generation algorithm. The current realization of *checkComp* is done in table 1:

Realization of created algorithms is on Github [2].

4. Related works

The increasing complexity of programs has led to a significant rise in the number of verification conditions (VCs), posing a major challenge in deductive program verification. As highlighted by Hähnle et al. [4] verification of cyber-physical systems, including programmable logic controllers, is itself one of the modern challenges in the field of deductive verification, and this issue only makes it harder. Additionally, various techniques aimed at simplifying VCs

[2]https://github.com/bearhug15/ReflexVCG

Condition	Return value
$p.reachE = false \land p.state = error$	$false$
$p.reachS = false \land p.startS = false \land p.state = stop$	$false$
$\exists p', s', st' \in passed.\ p \in p'.s'.st'.start \land p.state \neq first(p)$	$false$
$\exists p', s', st' \in passed.\ p \in p'.s'.st'.stop \land p.state \neq stop$	$false$
$\exists p', s', st' \in passed.\ p \in p'.s'.st'.error \land p.state \neq error$	$false$
$\exists st' \in passed.\ p'.s'.st'.reset = true \land st \equiv \{timeout \ldots\}$	$false$
Otherwise	$true$

Table 1: Table of *checkComp* rules.

can inadvertently exacerbate this problem. For instance, Leino et al. [7] propose a method that involves splitting a verification condition into multiple separate conditions, where the conjunction of these conditions remains equivalent to the original. While such techniques may offer potential benefits for program verification, they can complicate the situation for Reflex programs. This complexity arises from the finite state machine abstraction inherent in the Reflex language, which generates a control flow graph characterized by high coupling and low cohesion. Consequently, splitting a single condition can lead to the fragmentation of numerous conditional paths, necessitating the automatic discarding of irrelevant conditions at generation time, which relies on various forms of static analysis.

Couchot [2] introduces a graph-based technique for reducing verification conditions, utilizing two types of graphs: the constant dependence graph and the predicate dependence graph. These graphs facilitate the checking of VC satisfiability and the discharge of unsatisfiable conditions. However, this approach has notable drawbacks, including the requirement to construct a complete VC prior to checking and its reliance on SAT solvers. Such dependencies can significantly extend the VC generation process, making program development less efficient. In contrast, our approach focuses solely on syntax analysis, which is simpler and more time-efficient.

Another perspective on the challenges we address involves identifying unreachable states. In this context, a combination of process state values and timeout invocations is treated as a single state, with transitions between these combinations representing state transitions through iterations of the control loop. Several methods have been proposed to tackle this issue [1, 9]. They are based on three steps: 1)reachable state space over-approximation; 2) debugging; and 3) spurious solution detection. Changing in the last step allows us to vary the accuracy of methods to discard more unreachable states. The advantage of these methods is that they may consider discrete time, which in our case is equivalent to loop iterations. However, these methods require explicit state formulations, which can be memory-intensive and rely on slow model-checking techniques. Modification for improvement in analysis accuracy also significantly slows it.

Additionally, we can frame our problem as a context-free language (CFL) reachability problem and explore reachability analysis techniques [3, 8]. Given that the reachability of each process state may be constrained by various statements from earlier parts of the program, this approach resembles a variant of the all-pairs S-path problem. While applicable in scenarios where unnecessary processing of *stop* and *error* states can be eliminated, this method lacks completeness, as it does not adequately account for more complex program behaviors.

5. Conclusion

This paper presents our approach to reducing the amount of verification conditions corresponding to the unreachable paths. We created a set of attributes associated with path elements and a set of heuristics that detect incompatible combinations of attribute values on the paths.

In the future, we plan to expand this analysis with more attributes and heuristics to consider more unreachable paths.

References

1. Berryhill Ryan, Veneris Andreas G. Methodologies for Diagnosis of Unreachable States via Property Directed Reachability // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. — 2018. — Vol. 37. — P. 1298–1311. — URL: https://api.semanticscholar.org/CorpusID:13169698.
2. Couchot Jean-François, Giorgetti Alain, Stouls Nicolas. Graph Based Reduction of Program Verification Conditions // arXiv preprint arXiv:0907.1357. — 2009.
3. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis / Qirun Zhang, Michael R Lyu, Hao Yuan, Zhendong Su // Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. — 2013. — P. 435–446.
4. Hähnle Reiner, Huisman Marieke. 24 challenges in deductive software verification // 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements / EasyChair. — 2017. — P. 37–41.

5. Hähnle Reiner, Huisman Marieke. Deductive software verification: from pen-and-paper proofs to industrial tools // Computing and Software Science: State of the Art and Perspectives. — 2019. — P. 345–373. — URL: https://doi.org/10.1007/978-3-319-91908-9_18.
6. Ishchenko Artyom D., Anureev Igor S. Verification Condition Generator for Process-Oriented Programs in Reflex Language Using Isabelle/HOL // 2024 IEEE 25th International Conference of Young Professionals in Electron Devices and Materials (EDM). — 2024. — P. 1820–1825.
7. Leino K Rustan M, Moskal Michał, Schulte Wolfram. Verification condition splitting // Submitted manuscript, September. — 2008.
8. Reps Thomas. Program analysis via graph reachability // Information and software technology. — 1998. — Vol. 40, no. 11-12. — P. 701–726.
9. Suda Martin. Property directed reachability for automated planning // Journal of Artificial Intelligence Research. — 2014. — Vol. 50. — P. 265–319.
10. Zyubin Vladimir Evgenievich, Liakh Tatiana Viktorovna, Rozov Andrei Sergeevich. Reflex language: a practical notation for cyber-physical systems // System Informatics. — 2018. — no. 12. — P. 85–104. — URL: https://doi.org/10.31144/si.2307-6410.2018.n12.p85-104.