MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 30, NO. 3, 2023
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Logic for reasoning about bugs in loops over data sequences (IFIL)

D. A. Kondratyev[1]

DOI: 10.18255/1818-1015-2023-3-214-233

[1]A.P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences, 6, Acad. Lavrentjev pr., Novosibirsk 630090, Russia.

MSC2020: 68Q60
Research article
Full text in English

Received May 29, 2023
After revision June 16, 2023
Accepted June 20, 2023

Classic deductive verification is not focused on reasoning about program incorrectness. Reasoning about program incorrectness using formal methods is an important problem nowadays. Special logics such as Incorrectness Logic, Adversarial Logic, Local Completeness Logic, Exact Separation Logic and Outcome Logic have recently been proposed to address it. However, these logics have two disadvantages. One is that they are based on under-approximation approaches, while classic deductive verification is based on the over-approximation approach. One the other hand, the use of the classic approach requires defining loop invariants in a general case. The second disadvantage is that the use of generalized inference rules from these logics results in having to prove too complex formulas in simple cases. Our contribution is a new logic for solving these problems in the case of loops over data sequences. These loops are referred to as finite iterations. We call the proposed logic the Incorrectness Finite Iteration Logic (IFIL). We avoid defining invariants of finite iterations using a symbolic replacement of these loops with recursive functions. Our logic is based on special inference rules for finite iterations. These rules allow generating formulas with recursive functions corresponding to finite iterations. The validity of these formulas may indicate the presence of bugs in the finite iterations. This logic has been implemented in a new version of the C-lightVer system for deductive verification of C programs.

Keywords: deductive verification; Hoare logic; bug localization; program incorrectness; loop invariant; finite iteration; C-lightVer; ACL2

INFORMATION ABOUT THE AUTHORS

Dmitry A. Kondratyev
corresponding author
orcid.org/0000-0002-9387-6735. E-mail: apple-66@mail.ru
researcher, PhD in Computer Science.

For citation: D. A. Kondratyev, "Logic for reasoning about bugs in loops over data sequences (IFIL)", *Modeling and analysis of information systems*, vol. 30, no. 3, pp. 214-233, 2023.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 30, № 3, 2023
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Логика для суждений об ошибках в циклах над последовательностями данных (IFIL)

Д. А. Кондратьев[1]

DOI: 10.18255/1818-1015-2023-3-214-233

[1]Институт систем информатики им. А.П. Ершова Сибирского отделения Российской академии наук, 630090, Российская Федерация, г. Новосибирск, проспект Академика Лаврентьева, 6.

УДК 004.052.42
Научная статья
Полный текст на английском языке

Получена 29 мая 2023 г.
После доработки 16 июня 2023 г.
Принята к публикации 20 июня 2023 г.

Классическая дедуктивная верификация не ориентирована на доказательство некорректности программ. Доказательство некорректности программ с помощью формальных методов является актуальной задачей в настоящее время. Специальные логики, такие как Incorrectness Logic, Adversarial Logic, Local Completeness Logic, Exact Separation Logic и Outcome Logic, были недавно предложены для решения данной задачи. Но у данных логик имеется два недостатка. Во-первых, в данных логиках используются подходы, основанные на нижней аппроксимации, тогда как в классической дедуктивной верификации используется подход, основанный на верхней аппроксимации. С другой стороны, использование классического подхода требует в общем случае задания инвариантов циклов. Во-вторых, использование правил вывода для программных конструкций в их самом общем виде приводит к необходимости доказательства сложных формул в простых ситуациях. Нашим результатом, представленным в данной статье, является новая логика для решения данных проблем в случае циклов над последовательностями данных. Такая циклы мы называем финитными итерациями. Предложенную логику мы называем логикой для суждений о некорректности финитных итераций (IFIL). Мы избегаем задания инвариантов финитных итераций с помощью символической замены в условиях корректности переменных таких циклов применениями рекурсивных функций. Наша логика основана на специальных правилах вывода для финитных итераций. Эти правила позволяют выводить формулы с применениями рекурсивных функций, соответствующих финитным итерациям. Истинность этих формул может означать наличие ошибок в финитных итерациях. Данная логика была реализована в новой версии программной системы C-lightVer для дедуктивной верификации программ на языке C.

Ключевые слова: дедуктивная верификация; логика Хоара; локализация ошибок; некорректность программ; инвариант цикла; финитная итерация; C-lightVer; ACL2

ИНФОРМАЦИЯ ОБ АВТОРАХ

Дмитрий Александрович Кондратьев
автор для корреспонденции

orcid.org/0000-0002-9387-6735. E-mail: apple-66@mail.ru
научный сотрудник, кандидат физ.-мат. наук.

Для цитирования: D. A. Kondratyev, “Logic for reasoning about bugs in loops over data sequences (IFIL)”, *Modeling and analysis of information systems*, vol. 30, no. 3, pp. 214-233, 2023.

Introduction

Deductive verification allows reasoning about program correctness [1]. Classic deductive verification is based on Hoare Logic (HL) [2–4]. Hoare Logic for a particular programming language contains a set of correct inference rules and axioms for all programming constructs. This set is referred to as the axiomatic semantics of programming language. Verification conditions (VC) are the result of the application of inference rules to an annotated program. The validity of the verification conditions means the correctness of the annotated program.

Classic deductive verification is not focused on reasoning about program incorrectness. Reasoning about incorrectness using formal methods is an important task nowadays [5, 6]. Special logics such as Incorrectness Logic (IL) [6–8], Adversarial Logic (AL) [9], Local Completeness Logic (LCL) [10, 11], Exact Separation Logic (ESL) [12], Outcome Logic (OL) [13] and Hyper Hoare Logic (HHL) [14] have recently been proposed to address it. However, these logics have two disadvantages. One is that they are based on the under-approximation approach, while classic deductive verification is based on the over-approximation approach [12]. The disadvantage of the under-approximation approach is the following inference method: for correctness reasoning, you have to forget information as you go along a path, but you must remember all the paths; for incorrectness reasoning, you must remember information as you go along a path, but you have to forget some of the paths [7]. One the other hand, the use of the classic approach requires defining loop invariants in a general case. Let us note that loop invariant problem can be solved in the cases of certain kinds of loops [15–17]. The second disadvantage is that the use of generalized inference rules from these logics results in having to prove too complex formulas in simple cases [6].

We have proposed a new logic to address these problems in the case of loops over data sequences. The development and implementation of our logic is based on the use of the following existing methods and tools:

- **The core of Hoare logic** [2–4]. The inference rules in our logic correspond to the classic form of inference rules proposed in Hoare logic. Our logic may be considered as a special version of Hoare logic for reasoning about the incorrectness of finite iterations.
- **The symbolic method of verification of finite iterations** [17]. This method is applied to a special kind of loop, finite iterations. The core of this method is a symbolic replacement of finite iterations with special recursive functions. This method allows us to avoid defining invariants in the case of finite iterations.
- **The memory model of two subsets of C programming language, C-light and C-kernel** [18, 19]. This semantics uses the memory model based on the *MeM* and *MD* functions. *MeM* maps an object's name to its address, and *MD* maps an object's address to its value. This memory model is insufficient for reasoning about low-level memory operations, but allows proving properties of simple programs with pointers. However, the other part of C-light and C-kernel semantics is excessive for us; for example, so are special functions for modelling types or the inference rule for `goto` statement.
- **The mixed axiomatic semantics method** [20]. The goal of this method is to simplify verification conditions. This method is based on context-based inference rules. For example, many C variables are Pascal-like variables, i. e., the address-of and dereference operators are not applied to them. For the "Pascal" context, semantics based on a simpler schema including only one map of variable names to values is used.
- **The C-lightVer tool** [21, 22]. This is a system for C program deductive verification. The main advantage of this system is the implementation of the symbolic method of verification of finite iterations.
- **The ACL2 theorem prover** [23]. The ACL2 tool is used as a theorem prover in the C-lightVer system. Applicative Common Lisp (ACL) is the input language of the ACL2 system. Thus, C-lightVer system generates verification conditions written in ACL. The advantage of ACL2 system is a special logic based

on computable recursive functions. It allows ACL2 to automatize proving verification conditions with the use of recursive functions corresponding to finite iterations.

- **The algorithm of the generation of recursive functions corresponding to loops** [21, 22, 24]. This algorithm is based on a translation of the loop body to the definition of a recursive function written in the Applicative Common Lisp language. This algorithm was implemented in the C-lightVer system.
- **Strategies for proving properties that may indicate possible errors** [21, 22]. Our logic has been inspired by these strategies. These strategies are checking the validity of certain properties of finite iterations. The validity of these properties may indicate the presence of bugs in the input annotated program. The properties of the loops are generated as formulas with recursive functions corresponding to finite iterations. The proven formulas are added to the underlying theory as lemmas about the loops. Some lemmas may indicate the presence of bugs in the loops. These lemmas may be considered as unsafe properties of loops. The following strategies have been suggested:
  1. Try to prove property that checks whether `break` is always executed at the first loop iteration.
  2. Try to prove the property that checks whether assignments to array elements in the loop body exist and array elements after the loop execution are identical to the array elements before the loop execution. These assignment statements may be never be used in this case.

  These strategies generate implications where the premises are the loop preconditions and the conclusions are the properties of the recursive functions corresponding to finite iterations. However, this approach has two disadvantages:
  1. The user of the verification system should define a loop precondition such that it is sufficient to prove the properties of the finite iterations.
  2. Only two loop properties are checked using these strategies.

  Our logic is focused on the solution of these problems. First, our logic allows us to obtain loop preconditions using inference rules. Second, our logic allows us to define more general properties of the loops.
- **Strongest postcondition calculus** [2]. This approach is used to transform precondition of a given program to generate the strongest postcondition of this program. This approach allows us to obtain the loop precondition in our logic.

Our contribution is a logic for reasoning about bugs in loops over data sequences. This logic was implemented in the new version of the C-lightVer system.

This paper has the following structure. Preliminary information is provided in Section 1. The contribution of this paper is described in Section 2. The experiment demonstrating the application of our logic is described in Section 3.

Related works. The idea of Partial Incorrectness Logic (PIL) has been presented in the paper [25]. Let us note that Partial Incorrectness Logic is based on the same inference method (strongest postcondition calculus) as our logic. However, Partial Incorrectness Logic is applied to nondeterministic programs whereas our logic is applied to deterministic programs. Besides the aforementioned Incorrectness Logic (IL) [6–8], Adversarial Logic (AL) [9], Local Completeness Logic (LCL) [10, 11], Exact Separation Logic (ESL) [12], Outcome Logic (OL) [13] and Hyper Hoare Logic (HHL) [14], there are more practical approach to finding bugs using formal methods. The use of a counterexample generated by an SMT solver for error localization was described in [26]. However, analysis of the counterexample can be fairly complicated, which was demonstrated in [27]. Constrained Horn Clauses (CHCs) [28] allow reasoning about program properties; however, this approach requires defining special proving strategies in the case of real-world programs. Bounded verification described in [29] is based on loop unrolling without using loop invariants. But efficiency of this strategy depends on how many iterations are chosen to be unrolled. The approach [30] based on the deductive verification of the program with a mutation in the conditions of the `if` statements and `while` loops was

implemented in the Frama-C tool [31]. However, this approach requires that loop invariants to be defined. Model checking based on k-induction has been implemented in the ESBMC tool [32]. However, the efficiency of this approach depends on finding inductive invariants. Model checking based on counterexample-guided abstraction refinement (CEGAR) has been implemented in the CPAchecker tool [33]. However, the efficiency of this approach depends on the efficiency of a reachability analysis. The approach based on symbolic execution has been implemented, for example, in the CPA-SymExec tool [34] and in the KLEE tool [35]. However, this approach depends on the efficiency of constraint solvers in reasoning about path feasibility.

1. Existing methods and tools that we use to develop and implement our logic

To develop and implement our logic, we used the following methods and tools: the core of Hoare logic [2–4], the symbolic method of verification of finite iterations [17], the memory model of two subsets of C programming language (C-light and C-kernel) [18, 19], the mixed axiomatic semantics method [20], the C-lightVer tool [21, 22], the ACL2 theorem prover [23], the algorithm of generation of recursive functions corresponding to loops [21, 22], strategies for proving properties that may indicate possible errors [21, 22] and the strongest postcondition calculus [2].

1.1. The core of Hoare logic

Deductive program verification is applied to the Hoare triple. The Hoare triple has the following form:

$$\{P\}\ S\ \{Q\},$$

where

- P is the precondition (logical formula);
- S is the program (sequence of program statements);
- Q is the postcondition (logical formula).

Deductive program verification is an automatic derivation of valid (partially correct) Hoare triples. The partial correctness [2–4] of the Hoare triple means that if the precondition is true before the execution of a program fragment and if its execution terminates, then the postcondition is true upon its completion.

The inference rule has the following structure:

$$\frac{\psi_1, \ldots, \psi_n}{\varphi},$$

where

- ψ_1,…,ψ_n are premises (Hoare triples and logical formulas);
- φ is the conclusion (Hoare triple).

This notation means that φ is derived from ψ_1,…,ψ_n. As an example, let us consider the classic inference rule for the *while* loop:

$$\frac{\{P\}\ \textbf{prog};\ \{I\},\ \{I \wedge B\}\ \textbf{S}\ \{I\},\ I \wedge \neg B \rightarrow Q}{\{P\}\ \textbf{prog};\ \textbf{while B inv I do S}\ \{Q\}},$$

where I is the loop invariant.

It is necessary to use induction to derive the Hoare triple for the *while* loop. The induction statement in this case is called the loop invariant: this statement is true before the loop execution, true after each loop iteration, and ensures the correctness of loop exit. If the loop has a general form, it is necessary to define the loop invariant.

The syntax-driven axiomatic system (i. e. the one that contains inference rules for all syntax constructs of the programming language) is called Hoare logic or axiomatic semantics.

1.2. The symbolic method of verification of finite iterations

Given that $memb(S)$ denotes the multiset of elements of a data sequence S and $empty(S) = true$ if $|memb(S)| = 0$, let us define two functions:

1. $choo(S)$ returns an arbitrary element of $memb(S)$, if $\neg empty(S)$.
2. $rest(S) = S'$, where $memb(S') = memb(S) \setminus \{choo(S)\}$, if $\neg empty(S)$.

A finite iteration corresponds to the form:

$$\textbf{for x in S do v} := \textbf{body(v, x) end},$$

where

- S is the data sequence;
- x is the variable of type "element of S";
- v is the tuple of the loop variables excluding x;
- $body$ represents the loop body which does not alter x and terminates for every $x \in S$.

Let v_0 denote the initial values of variables from v. Let us define replacement operation $rep(v, S, body)$ for this loop:

1. if $empty(S)$, then $rep(v_0, S, body) = v_0$.
2. if $\neg empty(S)$, then

$$rep(v_0, S, body) = body(rep(v_0, rest(S), body), choo(S)).$$

The following inference rule has been suggested for a finite iteration:

$$\frac{\{P\}\ \textbf{prog};\ \{Q(v \leftarrow rep(v, S, body))\}}{\{P\}\ \textbf{prog};\ \textbf{for x in S do v} := \textbf{body(v, x) end}\ \{Q\}},$$

where $\leftarrow$ denotes a simultaneous substitution.

This method allows us to avoid defining invariants in the case of loops corresponding to finite iterations [17].

1.3. The memory model of two subsets of C programming language, C-light and C-kernel

The C-light language [18] is a representative subset of C. The operational semantics was developed for the C-light language. The memory model based on the *MeM* and *MD* functions is used in this semantics. *MeM* maps an object's name to its address, and *MD* maps an object's address to its value.

The *upd* operation allows us to create a new *MD* map when the memory state changes. Let us define the value of the expression $upd(MD,\ addr,\ val)$, where *MD* is an $address \to value$ map, *addr* is an address and *val* is a value. If *MD* contains an $(adr\ val')$ pair, where val' is some value, then $upd(MD,\ addr,\ val)$ differs from *MD* in that ih has $(adr\ val)$ instead of $(adr\ val')$. If *addr* is not in the range of *MD*, then $upd(MD,\ addr,\ val)$ differs from *MD* in that an $(adr\ val)$ pair is added to it.

Let us consider axioms about *MeM* and *MD*:

1) $MD(NULL) = void$;
2) $MeM(obj) \neq NULL$;
3) $upd(MD,\ NULL,\ val) = MD$;
4) $upd(MeM,\ obj,\ NULL) = MeM$;
5) $delete(MD, NULL) = MD$;
6) $(upd(MD,\ addr,\ val))(addr) = val$;
7) $(upd(MD, adr_1, val))(adr_2) = MD(adr_2)$
   $if\ adr_1 \neq adr_2$;
8) $upd(MD,\ MeM(obj),\ MD(MeM(obj))) = MD$;
9) $upd(MeM,\ obj\ MeM(obj)) = MeM$;

10) $(upd(MeM, obj, addr))(obj) = addr$;
11) $(upd(MeM, obj_1, adr))(obj_2) = MeM(obj_2)$
    $if\ obj_1 \neq obj_2$;
12) $(delete(MD, addr))(addr) = void$;
13) $(delete(MD, adr_1))(adr_2) = MD(adr_2)$
    $if\ adr_1 \neq adr_2$;
14) $(delete(MeM, obj))(obj) = void$;
15) $(delete(MeM, obj_1))(obj_2) = MeM(obj_2)$
    $if\ obj_1 \neq obj_2$;
16) $delete(upd(MD, addr, val), addr) = MD$;
17) $delete(upd(MeM, obj, addr), obj) = MeM$.

Incidentally, because the operational semantics of C-light has an unstructured memory model, this language does not support machine word level operations.

Since the C-kernel language [19] is a subset of the C-light language, its operational semantics is the same as the C-light semantics. Thus, the memory model of C-kernel language is equal to memory of C-light language.

1.4. The C-lightVer tool

The C-lightVer system is based on the classic deductive verification method [21, 22]. C-light is an input language of this system. C-kernel is an intermediate verification language of this tool. The axiomatic semantics has been defined for the C-kernel language.

At the first stage, C-light is translated into an intermediate language, C-kernel. This stage is necessary for elimination of constructs that are complicated for axiomatic semantics. A set of formal rules is used for this translation. For example, increment operators are eliminated by translation into pieces of code with assignments and addition.

At the second stage, verification conditions are generated for the intermediate C-kernel program. This process is based on the axiomatic semantics of C-kernel. Once generated, the verification conditions are passed to the theorem prover.

1.5. The ACL2 theorem prover

ACL2 [23] is used in the C-lightVer system for proving verification conditions with the use of recursive functions corresponding to finite iterations. There are two main advantages of the ACL2 theorem prover:

- If the formula to be proved contains a recursive function, ACL2 can automatically run proving by induction using the definition of this function as the induction schema.
- If the underlying theory contains a theorem that can be considered as a rewriting rule, then ACL2 can automatically apply this rule for rewriting the formula to be proved.

These features allows automatizing proving formulas with *rep* functions in a lot of cases.

1.6. The mixed axiomatic semantics method

The method of mixed axiomatic semantics allows using particular versions of inference rules for particular versions of program constructs [20]. Let us note that there are variables in C programs that are used without referencing and dereferencing operators. A large number of such variables may be used in C programs. A simpler memory model may be used for such variables than the one based on *MeM* and *MD*. Therefore, simpler inference rules may be applied to program constructs with such variables. This allows verification conditions to be simplified.

1.7. The algorithm of the generation of recursive functions corresponding to loops

Let us consider a finite iteration over one-dimensional array:

$$\mathbf{for}\ (\mathbf{i} = \mathbf{i_0};\ \mathbf{i} < \mathbf{n};\ \mathbf{i}++)\ \mathbf{v} := \mathbf{body}(\mathbf{v}, \mathbf{i})\ end,$$

where

- v is the tuple of modifiable variables;
- S ia an one-dimensional array of n elements;
- $S \in v$;
- *body* is the admissible construct.

The admissible construct is one of the following C-kernel operators:

1. An empty operator, including an empty block.
2. The **break**; operator ending the loop.
3. The assignment operator $\mathbf{a} = \mathbf{b};$, where a is a simple type variable or a variable $S[i]$, and b is an expression in C-kernel.
4. The conditional statement **if** $(\mathbf{a})$ **b**, where a is an expression in C-kernel and b is an admissible construct.
5. The conditional statement **if** $(\mathbf{a})$ **b else c**, where a is an expression in C-kernel, and b and c are admissible constructs.
6. The block $\{\mathbf{a_1}\ \mathbf{a_2}\ \ldots\ \mathbf{a_{k-1}}\ \mathbf{a_k}\}$, where a_r is the admissible construct for each r: $1 \le r \le k$.
7. The nested finite iteration
   for $(\mathbf{j} = \mathbf{j_0};\ \mathbf{j} < \mathbf{m};\ \mathbf{j}{+}{+})\ \mathbf{u} := \mathbf{body}(\mathbf{u},\ \mathbf{j})$ **end**.

Since n is not constant in the general case, we can not apply full loop unrolling in this case. We apply the symbolic method of verification of finite iterations in this case. The tuple v includes S and simple type variables that may be changed in the loop body. Let v_0 denote the initial values of variables from v. Let us consider the *rep* definition in this case:

- $rep(v_0, S, body, 0) = v_0$,
- $rep(v_0, S, body, i) = body(rep(v_0, S, body, i-1), S_{i-1})$
  for each $i = 1, 2, \ldots, n$.

If a finite iteration has a `break` statement, we suggest the following solution: when the execution of the loop is terminated by this statement, we assume that the loop iterations continue, but the v values remain unchanged. If `break` was executed at iteration i $(0 < i \le n)$, then for each j $(i \le j \le n)$:

$$rep(v_0, S, body, j) = rep(v_0, S, body, i),$$

The inference rule in this case has the following form:

$$\frac{\{P\}\ \mathbf{prog};\ \{Q(v \leftarrow rep(v, S, body, n))\}}{\{P\}\ \mathbf{prog};\ \mathbf{for}\ (\mathbf{i} = \mathbf{i_0};\ \mathbf{i} < \mathbf{n};\ \mathbf{i}{+}{+})\ \mathbf{v} := \mathbf{body}(\mathbf{v},\ \mathbf{i})\ \mathbf{end}\ \{Q\}},$$

The method for generating the *rep* function body is based on the algorithm that translates loop body constructs to Applicative Common Lisp (the ACL2 language). Let us consider this algorithm [21, 22, 24].

The structure type *frame* is generated. The fields of such structure correspond to loop variables, the *rep* function returns object of type *frame*. All objects of the type *frame* are referred to as *fr*. Each loop instruction can be represented as a creation of new *fr* object with the fields in it appearing as the updated fields of previous *fr* object.

The sequential execution of the statements is translated to $b*$ construct:

$$(b*\ (\ldots(var\ expr)\ldots)\ result),$$

where $(var\ expr)$ means binding *var* to the value of *expr* which may depend on previously bound variables. We use *fr* as such *var* and we use the updates of the *fr* fields as *expr*. To simulate the loop exit, we use the Boolean field *loop-break* of the *frame* object. This field is true only after `break` has been executed.

The following definition of *gen_rep* is the implementation of translation of admissible constructs to Applicative Common Lisp:

- *gen_rep*(**empty statement**) = $(fr\ fr)$
- *gen_rep*(**break**;) = $((when\ t)\ fr)$
- *gen_rep*(**c = b**;) = $(fr\ (change\text{-}frame\ fr\ :c\ b))$
- *gen_rep*(**a[i] = b**;) = $(fr\ (change\text{-}frame\ fr\ :a\ (update\text{-}nth\ i\ b\ fr.a)))$
- *gen_rep*(**if (c) b else d**) =
  $(fr\ (if\ c$
  $(b*\ (gen\_rep(b))\ fr)\ (b*\ (gen\_rep(d))\ fr)))$
  $((when\ fr.loop\text{-}break)\ fr)$
- *gen_rep*($\{\mathbf{a_1}\ \mathbf{a_2}\ \ldots\ \mathbf{a_{k-1}}\ \mathbf{a_k}\}$) =
  $(fr\ (b^*(gen\_rep(a_1)\ \ldots\ gen\_rep(a_k))\ fr))$
  $((when\ fr.loop\text{-}break)\ fr)$

The replacement operation obtained returns not only the tuple v, but also a structure with the Boolean field (*loop-break*). The default value of *loop-break* is false. The *rep* function obtained contains the break condition. Once the `break` statement has been executed, this *rep* will return a structure with the true value of the *loop-break* field. Additionally, this *rep* checks the value of the *loop-break* field from the result of the recursive call. If this value is true, than this *rep* returns the same result as this recursive call. Thus, the values of the loop variables do not change after the execution of the `break` statement in this implementation.

1.8. Strategies for proving properties that may indicate possible errors

Given the user-defined precondition P of the loop, two strategies for proving properties that may indicate possible errors have been suggested [21, 22].

1.8.1. A strategy for finding loops with unused assignments to array elements

Let a loop implementing a finite iteration over an array contain assignments to elements of this array, and let the values of the array elements after the loop execution be equal to the values of these elements before the loop execution.

The strategy for finding such loops checks each loop over the array containing assignments to the array elements [21]. Let this loop be the i-th in the program code. The strategy is based on generating the lemma $P \rightarrow (a = rep_i(a, args).a)$, where

- P is the precondition;
- a is the array over which the finite iteration is performed;
- rep_i is the replacement operation for the finite iteration;
- $args$ are the arguments of rep_i;
- $rep_i(a, args).a$ is the array a after the loop execution,

and checking the validity of this lemma.

If this lemma was proved, than these assignments can appear to be unused statements, which may indicate the presence of an error.

1.8.2. A strategy for checking the execution of the `break` statement at the first loop iteration

Suppose that the loop implementing a finite iteration over an array contains a `break` statement that is always executed at the first loop iteration.

The strategy checks each loop over an array containing a `break` statement [21]. Let the loop be the i-th in the program code. The strategy is based on generating the lemma

$$P \rightarrow ((j_0 = rep_i(a, args).j) \wedge (rep_i(a, args).loop\text{-}break)),$$

where

- P is the precondition;
- a is the array over which the finite iteration is performed;

- j is the counter of the *for* loop which implements the finite iteration;
- *args* are the arguments of rep_i; the value j_0 of the loop variable j before the loop execution;
- $rep_i(a, args).j$ is the value of j after the loop execution;
- *loop-break* is a special field in the returned data structure; its value is true if and only if the `break` statement was executed during the loop execution,

and checking the validity of this lemma.

Thus, the first clause in the lemma conclusion is the assertion that the loop execution did not change the value of the loop counter. The second clause in the conclusion is the assertion that the `break` statement in the loop was executed.

If this lemma was proved, then this case may indicate the presence of an error.

Both strategies have been integrated in our logic.

1.9. Strongest postcondition calculus

The strongest postcondition [2] inference is applied to a program and its precondition. The formula $sp(S, P)$ is the strongest postcondition of a program S with a precondition P iff

- the triple $\{P\}\ S\ \{sp(S, P)\}$ is correct and
- if Q is a formula then validity of the formula $sp(S, P) \rightarrow Q$ implies the correctness of the triple $\{P\}\ S\ \{Q\}$.

This calculus allows defining axiomatic semantics using the following approach: if *stmt* is a program statement, then the inference rule for this statement may be defined as

$$\frac{\{sp(stmt, P)\}\ \textbf{prog};\ \{Q\}}{\{P\}\ \textbf{stmt};\ \textbf{prog}\ \{Q\}}.$$

This approach can be considered as forward tracking: moving from the beginning of the program to its end and eliminating the leftmost statement (at the top level) by applying the corresponding inference rule. Our logic was developed using this inference style.

2. Logic for reasoning about bugs in loops over data sequences

The set of inference rules from our logic can be partitioned into the following two subsets: base inference rules and main inference rules. The goal of the base inference rules is to obtain the loop precondition. The goal of the main inference rules is to obtain the properties that may indicate the presence of bugs in the loops.

2.1. Base of logic for reasoning about bugs in loops over data sequences

Let us consider the inference rules for the kernel of the C-kernel-like subset of C language.

The inference rule for an empty program is:

$$\frac{P \rightarrow Q}{\{P\}\ \textbf{emptyProgram}\ \{Q\}}. \tag{1}$$

The strongest postcondition of the empty program has the following form:

$$sp(\textbf{emptyProgram},\ P) = P.$$

The inference rule for the variable declaration is:

$$\frac{\{\exists MeM' P(MeM \leftarrow MeM') \wedge MeM = upd(MeM', var, addr)\}\ \textbf{prog};\ \{Q\}}{\{P\}\ \textbf{type var};\ \textbf{prog}\ \{Q\}}, \tag{2}$$

where *addr* is the new address ($addr \notin Dom(MD)$). The strongest postcondition of variable declaration has the following form:

$$sp(\textbf{type var},\ P) = \\ \exists MeM' P(MeM \leftarrow MeM') \wedge MeM = upd(MeM', var, addr).$$

The inference rule for variable assignment is:

$$\frac{\{\exists MD' P(MD \leftarrow MD') \wedge MD = upd(MD', MeM(var), rval)\}\ \textbf{prog};\ \{Q\}}{\{P\}\ \textbf{var} = \textbf{rval};\ \textbf{prog}\ \{Q\}}. \tag{3}$$

The strongest postcondition of variable assignment is:

$$sp(\textbf{var} = \textbf{rval},\ P) = \\ \exists MD' P(MD \leftarrow MD') \wedge MD = upd(MD', MeM(var), rval).$$

The mixed axiomatic semantics method allows us to define the following version of this inference rule when neither referencing nor dereferencing operators are used on *var*:

$$\frac{\{\exists var' P(var \leftarrow var')\ \wedge var = rval(var \leftarrow var')\}\ \textbf{prog};\ \{Q\}}{\{P\}\ \textbf{var} = \textbf{rval};\ \textbf{prog}\ \{Q\}}. \tag{4}$$

The strongest postcondition of variable assignment in this case is:

$$sp(\textbf{var} = \textbf{rval},\ P) = \\ \exists var' P(var \leftarrow var')\ \wedge var = rval(var \leftarrow var').$$

The inference rule for assignment to an array element has the following form:

$$\frac{\{\exists MD' P(MD \leftarrow MD') \wedge MD = upd(MD', MeM(a, i), rval)\}\ \textbf{prog};\ \{Q\}}{\{P\}\ \textbf{a[i]} = \textbf{rval};\ \textbf{prog}\ \{Q\}}. \tag{5}$$

Strongest postcondition of assignment to an array element has the following form:

$$sp(\textbf{a[i]} = \textbf{rval},\ P) = \\ \exists MD' P(MD \leftarrow MD') \wedge MD = upd(MD', MeM(a, i), rval).$$

The mixed axiomatic semantics method allows us to define the following version of this inference rule when neither referencing nor dereferencing operators are used on the array element:

$$\frac{\{\exists a' P(a \leftarrow a') \wedge a = update(a', i, rval)\}\ \textbf{prog};\ \{Q\}}{\{P\}\ \textbf{a[i]} = \textbf{rval};\ \textbf{prog}\ \{Q\}}, \tag{6}$$

where *update* is the array update operation with the following axioms:

1) $(update(a,\ i,\ val))[i] = val$;
2) $(update(a, i_1, val))[i_2] = a(i_2)$
   $if\ i_1 \neq i_2$;
3) $update(a,\ i,\ a[i]) = a$.

The strongest postcondition of assignment to an array element has the following form in this case:

$$sp(\textbf{a[i]} = \textbf{rval},\ P) = \\ \exists a' P(a \leftarrow a') \wedge a = update(a', i, rval).$$

The inference rule for the `if` statement has the following form:

$$\frac{\{P \wedge B\}\ \mathbf{S_1};\ \mathbf{prog}\ \{Q\},\ \{P \wedge \neg B\}\ \mathbf{S_2};\ \mathbf{prog}\ \{Q\}}{\{P\}\ \mathbf{if\ B\ then\ S_1\ else\ S_2};\ \mathbf{prog}\ \{Q\}}. \tag{7}$$

The strongest postcondition of the `if` statement has the following form:

$$sp(\mathbf{if\ B\ then\ S_1\ else\ S_2},\ P) = \\ sp(\mathbf{S_1}, P \wedge B) \vee sp(\mathbf{S_2}, P \wedge \neg B).$$

These inference rules allow obtaining the finite iteration precondition that is used in the main inference rules.

2.2. Main inference rules of the logic for reasoning about bugs in loops over data sequences

Let us consider inference rules for the inference of formulas whose validity may indicate the presence of bugs.

The following inference rule corresponds to the strategy for finding loops with unused assignments to array elements:

$$\frac{P \rightarrow (S = rep(v, S, n).S)}{\{P\}\ \mathbf{for}\ (\mathbf{i}\ =\ \mathbf{i_0};\ \mathbf{i}\ <\ \mathbf{n};\ \mathbf{i}++)\ \mathbf{v} := \mathbf{body(v,\ i)}\ \mathbf{end};\ \mathbf{prog}\ \{Q\}}, \tag{8}$$

where

- P is the precondition;
- S is the array over which the finite iteration is performed;
- rep is the replacement operation for the finite iteration;
- $rep(v, S, n).S$ is the array S after the loop execution.

The following inference rule corresponds to the strategy for checking the execution of the `break` statement at the first loop iteration:

$$\frac{P \rightarrow ((i_0 = rep(v, S, n).i) \wedge (rep(v, S, n).loop{-}break))}{\{P\}\ \mathbf{for}\ (\mathbf{i}\ =\ \mathbf{i_0};\ \mathbf{i} < \mathbf{n};\ \mathbf{i}++)\ \mathbf{v} := \mathbf{body(v,\ i)}\ \mathbf{end};\ \mathbf{prog}\ \{Q\}}, \tag{9}$$

where

- P is the precondition;
- S is the array over which the finite iteration is performed;
- rep is the replacement operation for the finite iteration;
- $rep(v, S, n).i$ is the value of the loop counter i after the loop execution;
- $rep(v, S, n).S$ is the array a after the loop execution.

The next inference rules are based on modifying the definition of the rep function. Let us note that the next inference rules are not strategies from [21, 22] encoded in our logic, they represent a completely new approach. If the rep function contains an `if` statement, than we add to two new fields to the *frame* structure: $if{-}true_k$ and $if{-}false_k$, where k is the number of `if` statement in the finite iteration.

The $if{-}true_k$ field contains a conjunction of the values of the condition of the k-th `if` statement on each iteration. Thus, the true value of this field means that the condition of the k-th `if` statement is true on each iteration. This information may be indicative of an error.

The $if{-}false_k$ field contain a conjunction of negations of the values of the condition of the k-th `if` statement on each iteration. Thus, the true value of this field means that the condition of the k-th `if` statement is false on each iteration. This information may be indicative of an error.

For each k (for each `if` statement in the finite iteration) the following inference rules are applied:

$$\frac{P \rightarrow rep(v, S, n).if{-}true_k}{\{P\}\ \textbf{for}\ (\mathbf{i}\ =\ \mathbf{i_0};\ \mathbf{i}\ <\ \mathbf{n};\ \mathbf{i}++)\ \mathbf{v} := \textbf{body}(\mathbf{v},\ \mathbf{i})\ \textbf{end};\ \textbf{prog}\ \{Q\}}, \tag{10}$$

where

- P is the precondition;
- S is the array over which the finite iteration is performed;
- rep is the replacement operation for the finite iteration;
- $rep(v, S, n).if{-}true_k$ is the value of the $if{-}true_k$ field after the loop execution,

and

$$\frac{P \rightarrow rep(v, S, n).if{-}false_k}{\{P\}\ \textbf{for}\ (\mathbf{i}\ =\ \mathbf{i_0};\ \mathbf{i}\ <\ \mathbf{n};\ \mathbf{i}++)\ \mathbf{v} := \textbf{body}(\mathbf{v},\ \mathbf{i})\ \textbf{end};\ \textbf{prog}\ \{Q\}}, \tag{11}$$

where

- P is the precondition;
- S is the array over which the finite iteration is performed;
- rep is the replacement operation for the finite iteration;
- $rep(v, S, n).if{-}false_k$ is the value of the $if{-}false_k$ field after the loop execution.

We can modify the definition of the *rep* function to calculate the values of arbitrary formulas over finite iteration variables. It allows reasoning about the properties of finite iterations. It is possible to extend our logic with new inference rules for error localization.

Let us note that we may apply several inference rules to a particular finite iteration to reason about several types of bugs. We call the proposed logic the Incorrectness Finite Iteration Logic (IFIL).

2.3. Termination of inference based on the proposed logic

Let us consider the following theorem:

Theorem 1. *If the inference rules from IFIL are applied to an annotated sequence of program statements that can contain only variable declarations, assignments to variables, assignments to elements of arrays, `if` statements and finite iterations then this inference will be terminated.*

Proof. Given

- *statements* is the name of the sequence of the program statements considered in the statement of the theorem,
- *count_of_statements* is the function that returns the count of statements in the sequence of the program statements (including nested statements, for example, statements from branches of *if statements*),
- $count\_of\_statements(statements) = n$,
- *top_level_head* is the function that returns the first element at the top level of the sequence of the program statements (for example, we consider each *if statement* as one statement at the top level of the sequence of the program statements),
- *top_level_tail* is the function that returns the sequence of the program statements without the first element at the top level of the sequence of the program statements (for example, we consider each *if statement* as one statement at the top level of the sequence of the program statements),

let us prove this theorem by induction on n. Thus, the proof consists of two cases:

1. **Induction base.** This case corresponds to $n = 0$. The *statements* has the following form

$$emptyProgram$$

in this case. Thus, $top\_level\_head(statements)$ is equal to

$$emptyProgram$$

and $top\_level\_tail(statements)$ is equal to $emptyProgram$. Let us note that

$$count\_of\_statements(top\_level\_head(statements)) = count\_of\_statements(emptyProgram) = 0$$

Thus,

$$count\_of\_statements(top\_level\_tail(statements)) = \\ n - count\_of\_statements(top\_level\_head(statements)) = 0$$

We should apply inference rule 1 from Section 2.1 in this case. Since the application of inference rule 1 from Section 2.1 will be terminated, the inference process will be terminated, too.

2. **Induction step.** Let us consider all possible values of $top\_level\_head(statements)$:
   (a) *Variable declaration.* The *statements* has the following form

   $$type\ \ var;\ \ prog$$

   in this case. Thus, $top\_level\_head(statements)$ is equal to

   $$type\ \ var$$

   and $top\_level\_tail(statements)$ is equal to *prog*. Let us note that

   $$count\_of\_statements(top\_level\_head(statements)) = count\_of\_statements(type\ var) = 1$$

   Thus,

   $$count\_of\_statements(top\_level\_tail(statements)) = \\ n - count\_of\_statements(top\_level\_head(statements)) = n - 1$$

   We should apply inference rule 2 from Section 2.1 in this case. Since the application of inference rule 2 will be terminated and

   $$count\_of\_statements(prog) = count\_of\_statements(top\_level\_tail(statements)) = n - 1$$

   in this case, we can apply the induction hypothesis. The induction hypothesis means that the application of the inference process to annotated program code *prog* will be terminated in this case. Thus, the inference process will be terminated.
   (b) *Variable assignment.* The *statements* has the following form

   $$var = rval;\ \ prog$$

   in this case. Thus, $top\_level\_head(statements)$ is equal to

   $$var = rval$$

   and $top\_level\_tail(statements)$ is equal to *prog*. Let us note that

   $$count\_of\_statements(top\_level\_head(statements)) = count\_of\_statements(var = rval) = 1$$

   Thus,

   $$count\_of\_statements(top\_level\_tail(statements)) = \\ n - count\_of\_statements(top\_level\_head(statements)) = n - 1$$

   We should apply either inference rule 3 from Section 2.1 or inference rule 4 from Section 2.1 in this case. Since the application of either of the inference rules will be terminated and

   $$count\_of\_statements(prog) = count\_of\_statements(top\_level\_tail(statements)) = n - 1$$

in both cases, we can apply the induction hypothesis. The induction hypothesis means that application of inference process to annotated program code *prog* will be terminated in this case. Thus, the inference process will be terminated.

(c) *Assignment to an array element.* The *statements* has the following form

$$a[i] = rval;\ prog$$

in this case. Thus, $top\_level\_head(statements)$ is equal to

$$a[i] = rval$$

and $top\_level\_tail(statements)$ is equal to *prog*. Let us note that

$$count\_of\_statements(top\_level\_head(statements)) = count\_of\_statements(a[i] = rval) = 1$$

Thus,

$$count\_of\_statements(top\_level\_tail(statements)) = \\ n - count\_of\_statements(top\_level\_head(statements)) = n - 1$$

We should apply either inference rule 5 from Section 2.1 or inference rule 6 from Section 2.1 in this case. Since the application of either of the inference rules will be terminated and

$$count\_of\_statements(prog) = count\_of\_statements(top\_level\_tail(statements)) = n - 1$$

in both cases, we can apply the induction hypothesis. The induction hypothesis means that the application of the inference process to annotated program code *prog* will be terminated in this case. Thus, the inference process will be terminated.

(d) *if statement.* The *statements* has the following form

$$if\ B\ then\ S_1\ else\ S_2;\ prog$$

in this case. Thus, $top\_level\_head(statements)$ is equal to

$$if\ B\ then\ S_1\ else\ S_2$$

and $top\_level\_tail(statements)$ is equal to *prog*. Let us note that

$$count\_of\_statements(top\_level\_head(statements)) = \\ count\_of\_statements(if\ B\ then\ S_1\ else\ S_2) = \\ 1 + count\_of\_statements(S_1) + count\_of\_statements(S_2)$$

Thus,

$$count\_of\_statements(top\_level\_tail(statements)) = \\ n - count\_of\_statements(top\_level\_head(statements)) = \\ n - 1 - count\_of\_statements(S_1) - count\_of\_statements(S_2)$$

Let us note that

$$count\_of\_statements(prog) = count\_of\_statements(top\_level\_tail(statements)) = \\ n - 1 - count\_of\_statements(S_1) - count\_of\_statements(S_2)$$

Consequently,

$$count\_of\_statements(S_1;\ prog) = count\_of\_statements(S_1) + count\_of\_statements(prog) = \\ count\_of\_statements(S_1) + n - 1 - count\_of\_statements(S_1) - count\_of\_statements(S_2) = \\ n - 1 - count\_of\_statements(S_2)$$

and

$$count\_of\_statements(S_2;\ prog) = count\_of\_statements(S_2) + count\_of\_statements(prog) = \\ count\_of\_statements(S_2) + n - 1 - count\_of\_statements(S_1) - count\_of\_statements(S_2) = \\ n - 1 - count\_of\_statements(S_1)$$

Since $count\_of\_statements(S_2) \geq 0$,

$$count\_of\_statements(S_1;\ prog) = n - 1 - count\_of\_statements(S_2) \leq n - 1$$

Since $count\_of\_statements(S_1) \geq 0$,

$$count\_of\_statements(S_2;\ prog) = n - 1 - count\_of\_statements(S_1) \leq n - 1$$

We should apply inference rule 7 from Section 2.1 in this case. Since the application of inference rule 7 will be terminated and

$$count\_of\_statements(S1;\ prog) \leq n - 1$$

and

$$count\_of\_statements(S2;\ prog) \leq n - 1$$

we can apply the induction hypothesis. The induction hypothesis means that the application of the inference process to annotated program code $S_1;\ prog$ will be terminated and the application of the inference process to annotated program code $S_2;\ prog$ will be terminated. Thus, the inference process will be terminated.

(e) *Finite iteration.* The *statements* has the following form

$$for\ (i\ =\ i_0;\ i\ <\ n;\ i++)\ v := body(v,\ i)\ end;\ prog$$

in this case. Thus, $top\_level\_head(statements)$ is equal to

$$for\ (i\ =\ i_0;\ i\ <\ n;\ i++)\ v := body(v,\ i)\ end$$

and $top\_level\_tail(statements)$ is equal to *prog*. We should apply one the following rule:

- inference rule 8 from Section 2.2,
- inference rule 9 from Section 2.2,
- inference rule 10 from Section 2.2,
- inference rule 11 from Section 2.2

in this case. Since the application of all mentioned inference rules from Section 2.2 will be terminated, the inference process will be terminated.

□

Let us note that if the input sequence of the program statements does not contain finite iteration, we infer only the strongest postcondition instead of the formula describing the property of a finite iteration. Proving other properties of our logic is a difficult problem due to the heuristic nature of our logic. In any case, we are planning to discovery and to prove other properties of our logic in a future work.

3. Experiment

Our logic was implemented in the C-lighVer system as an alternative semantics for the C-kernel language. We performed a bug localization experiment on the `negate_first` program from the well-known verification challenge [36]:

```
void negate_first(int n, int* a) {
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] < 0) {a[i] = -a[i]; break;}}}
```

The precondition of this program is

$$(a_0 = a) \wedge (0 < n) \wedge (n \leq length(a_0))$$

The postcondition of this program is

$$\begin{gathered}(\neg found\_negative(n,\ a_0) \rightarrow \\ a = a_0) \\ \wedge \\ (found\_negative(n,\ a_0) \rightarrow \\ a = update(a_0, count\_index(n,\ a_0), -a_0[count\_index(n,\ a_0)]))\end{gathered}$$

where

- *found-negative* predicate checks if there is a negative element in the array;
- *count-index* function calculates the index of the first negative element of the array if the array contains such an element. The value of this function is undefined in other cases.

The main problem of reasoning about this program is the `break` statement in the loop.

Let us consider the `negate_first` program with an introduced error:

```
1. /*@ requires (a0 = a) && (0 < n) && (n <= length(a0));
2. ensures (!found_negative(n, a0) ==> a == a0) &&
3.         (found_negative(n, a0) ==>
              a == update(a0, count_index(n, a0), -a0[count_index(n, a0)]))
4. */
5. void negate_first(int n, int* a) {
6.   int i;
7.   for (i = 0; i < n; i++) \{
8.        if (a[i] < a[i]) {a[i] = -a[i]; break;}}}
```

The specifications of this function were defined using the ACSL language. The error is using `a[i]` instead of `0` in the `if` condition at line 8.

The following formulas obtained have been proved automatically using the ACL2 system:

```
(implies
    (and (integer-listp a) (integer-listp a_0) (equal a a_0)
         (integerp n) (< 0 n) (<= n (length a_0)))
    (equal
        a
        (rep
            (frame-init
                0
                a
                nil
            )
            (envir-init
                n
            )
        ).a
    )
)
```

and

```
(implies
     (and (integer-listp a) (integer-listp a_0) (equal a a_0)
          (integerp n) (< 0 n) (<= n (length a_0)))
     (equal
         t
         (rep
             (frame-init
                 0
                 a
                 nil
             )
             (envir-init
                 n
             )
         ).iffalse1
    )
)
```

where

- `frame-init` is the constructor of the *frame* structure;
- `envir-init` is the constructor of the *envir* structure which stores values that have not been modified by the finite iteration.

The mixed axiomatic semantics method allows using simple inference rules without *MeM* or *MD* to obtain these formulas.

The validity of these formulas means the following:

- assignment in the loop body is never used;
- the condition of `if` statement is always false.

Thus, our logic can help localize bug without using loop invariants.

Conclusion

The new result presented in this paper is a logic for reasoning about bugs in loops over data sequences. This logic is based on special inference rules for finite iterations. These rules allow generating properties that may indicate errors in finite iterations. These properties are generated as formulas with recursive functions corresponding to finite iterations. This logic has been implemented in a new version of the C-lightVer system for a deductive verification of C programs. We have performed reasoning about the incorrectness of an illustrative example to demonstrate how our logic works.

There are three advantages of our approach:

1. Our logic does not require defining invariants of finite iterations.
2. Our logic is based on an over-approximation approach similar to classic deductive verification. It simplifies the development of a unified approach to reasoning about correctness and incorrectness.
3. Our logic is based on the use of special inference rules for finite iterations. These rules allow generating simple formulas to be proved in the case of finite iterations.

We believe that our approach has promise, because we can extend our logic with new special inference rules for finite iterations. Thus, we are planning to extend our logic with new inference rules to handle more types of bugs. For example, we will soon be applying our approach to finite iterations over lists, trees and other dynamic data structures.

References

[1] R. Hähnle and M. Huisman, "Deductive software verification: From pen-and-paper proofs to industrial tools", in *Computing and Software Science*, vol. 10000, Springer, 2019, pp. 345–373.

[2] K. R. Apt and E.-R. Olderog, "Fifty years of Hoare's logic", *Formal Aspects of Computing*, vol. 31, no. 6, pp. 751–807, 2019.

[3] K. R. Apt and E.-R. Olderog, "Assessing the success and impact of Hoare's logic", in *Theories of Programming: The Life and Works of Tony Hoare*, 2021, pp. 41–76.

[4] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[5] B. Möller, P. O'Hearn, and T. Hoare, "On algebra of program correctness and incorrectness", in *Relational and Algebraic Methods in Computer Science*, vol. 13027, Springer, 2021, pp. 325–343.

[6] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic", *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–27, 2022.

[7] P. W. O'Hearn, "Incorrectness logic", *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019.

[8] A. Raad, J. Berdine, H.-H. Dang, D. Dreyer, P. O'Hearn, and J. Villard, "Local reasoning about the presence of bugs: Incorrectness separation logic", in *Computer Aided Verification*, vol. 12225, Springer, 2020, pp. 225–252.

[9] J. Vanegue, "Adversarial logic", in *Static Analysis*, vol. 13790, Springer, 2022, pp. 422–448.

[10] M. Milanese and F. Ranzato, "Local completeness logic on Kleene algebra with tests", in *Static Analysis*, vol. 13790, Springer, 2022, pp. 350–371.

[11] B. Bruni, R. Giacobazzi, R. Gori, and F. Ranzato, "A correctness and incorrectness program logic", *Journal of the ACM*, vol. 70, no. 2, pp. 1–45, 2023.

[12] P. Maksimović, C. Cronjäger, A. Lööw, J. Sutherland, and P. Gardner, "Exact separation logic: Towards bridging the gap between verification and bug-finding", in *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, vol. 263, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 19:1–19:27.

[13] N. Zilberstein, D. Dreyer, and A. Silva, "Outcome logic: A unifying foundation of correctness and incorrectness reasoning", *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 522–550, 2023.

[14] T. Dardinier and P. Müller, *Hyper hoare logic: (Dis-)Proving program hyperproperties (extended version)*, 2023. arXiv: 2301.10037 [cs.LO].

[15] A. Humenberger, M. Jaroschek, and L. Kovács, "Invariant generation for multi-path loops with polynomial assignments", in *Verification, Model Checking, and Abstract Interpretation*, vol. 10747, Springer, 2018, pp. 226–246.

[16] S. Chakraborty, A. Gupta, and D. Unadkat, "Full-program induction: Verifying array programs sans loop invariants", *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 5, pp. 843–888, 2022.

[17] V. A. Nepomniaschy, "Symbolic method of verification of definite iterations over altered data structures", *Programming and Computer Software*, vol. 31, no. 1, pp. 1–9, 2005.

[18] V. A. Nepomniaschy, I. S. Anureev, I. N. Mikhailov, and A. V. Promskii, "Towards verification of C programs. C-light language and its formal semantics", *Programming and Computer Software*, vol. 28, no. 6, pp. 314–323, 2002.

[19] V. A. Nepomniaschy, I. S. Anureev, and A. V. Promskii, "Towards verification of C programs: Axiomatic semantics of the C-kernel language", *Programming and Computer Software*, vol. 29, no. 6, pp. 338–350, 2003.

[20] I. V. Maryasov, V. A. Nepomniaschy, A. V. Promsky, and D. A. Kondratyev, "Automatic C program verification based on mixed axiomatic semantics", *Automatic Control and Computer Sciences*, vol. 48, no. 7, pp. 407–414, 2014.

[21] D. A. Kondratyev and V. A. Nepomniaschy, "Automation of C program deductive verification without using loop invariants", *Programming and Computer Software*, vol. 48, no. 5, pp. 331–346, 2022.

[22] D. A. Kondratyev and A. V. Promsky, "The complex approach of the C-lightVer system to the automated error localization in C-programs", *Automatic Control and Computer Sciences*, vol. 54, no. 7, pp. 728–739, 2020.

[23] J. S. Moore, "Milestones from the pure lisp theorem prover to ACL2", *Formal Aspects of Computing*, vol. 31, no. 6, pp. 699–732, 2019.

[24] D. A. Kondratyev, I. V. Maryasov, and V. A. Nepomniaschy, "The automation of C program verification by the symbolic method of loop invariant elimination", *Automatic Control and Computer Sciences*, vol. 53, no. 7, pp. 653–662, 2019.

[25] L. Zhang and B. L. Kaminski, "Quantitative strongest post: A calculus for reasoning about the flow of quantitative information", *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–29, 2022.

[26] S. Dailler, D. Hauzar, C. Marché, and Y. Moy, "Instrumenting a weakest precondition calculus for counterexample generation", *Journal of Logical and Algebraic Methods in Programming*, vol. 99, pp. 97–113, 2018.

[27] B. Becker, C. B. Lourenço, and C. Marché, "Explaining counterexamples with giant-step assertion checking", in *Proceedings of the 6th Workshop on Formal Integrated Development Environment*, vol. 338, 2021, pp. 82–88.

[28] Q. L. Le, J. Sun, L. H. Pham, and S. Qin, *S2TD: A separation logic verifier that supports reasoning of the absence and presence of bugs*, 2022. arXiv: 2209.09327 [cs.PL].

[29] T. Dardinier, G. Parthasarathy, and P. Müller, "Verification-preserving inlining in automatic separation logic verifiers", *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 789–818, 2023.

[30] R. Könighofer, R. Toegl, and R. Bloem, "Automatic error localization for software using deductive verification", in *Hardware and Software: Verification and Testing*, vol. 8855, Springer, 2014, pp. 92–98.

[31] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams, "The dogged pursuit of bug-free C programs: The Frama-C software analysis platform", *Communications of the ACM*, vol. 64, no. 8, pp. 56–68, 2021.

[32] M. R. Gadelha, F. Monteiro, L. Cordeiro, and D. Nicole, "ESBMC v6.0: Verifying C programs using *k*-induction and invariant inference", in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 11429, Springer, 2019, pp. 209–213.

[33] S. Löwe, "CPAchecker with explicit-value analysis based on CEGAR and interpolation", in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 7795, Springer, 2013, pp. 610–612.

[34] D. Beyer and T. Lemberger, "CPA-SymExec: Efficient symbolic execution in CPAchecker", in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 900–903.

[35] C. Cadar and M. Nowack, "KLEE symbolic execution engine in 2019", *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 6, pp. 867–870, 2021.

[36] B. Jacobs, J. Kiniry, and M. Warnier, "Java program verification challenges", in *Formal Methods for Components and Objects*, vol. 2852, Springer, 2003, pp. 202–219.