DOI: 10.15514/ISPRAS-2026-38(6)-23



# Применение статического анализа для сокращения количества порождаемых условий корректности для программ на языке Reflex

*А.Д. Ищенко, ORCID: 0009-0005-6176-5917 <iartem2000@gmail.com>*
*И.С. Ануреев, ORCID: 0000-0001-9574-128X <anureev@gmail.com>*
*Институт Автоматики и Электрометрии СО РАН,*
*Россия, 630090, г. Новосибирск, пр. Академика Коптюга, 1.*

**Аннотация.** Системы управления широко применяются в критически важных областях, что предъявляет повышенные требования к надежности программного обеспечения для таких систем и к обоснованию его корректности. Для выполнения этих требований используют формальные методы, в частности дедуктивную верификацию, которая предполагает генерацию условий корректности на основе аксиоматической семантики программы и ее спецификаций. Однако автоматическая генерация условий корректности нередко приводит к появлению избыточных и/или чрезмерно сложных формул, что затрудняет их последующую проверку и снижает эффективность автоматических доказателей. Частично эту проблему решают предметно-ориентированные языки, такие как Reflex, который использует процесс-ориентированную парадигму, упрощающую структуру программ. Вместе с тем особенности семантики конструкций Reflex могут вызывать дополнительный рост числа условий корректности за счет нерелевантных условий корректности, соответствующих недостижимым путям в программе. В работе предлагается метод статического анализа, направленный на сокращение числа генерируемых условий корректности для программ на языке Reflex. Метод основан на аннотировании операторов программы набором атрибутов, отражающих их семантические свойства. Сначала атрибуты вычисляются для элементарных операторов, а затем агрегируются для составных конструкций. В процессе генерации условий корректности атрибуты накапливаются вдоль пройденного пути выполнения, после чего проверяется их совместность. Несовместимые комбинации атрибутов позволяют выявлять и отбрасывать нерелевантные условия корректности. Экспериментальная оценка на наборе тестовых программ показала, что предложенный подход позволяет существенно сократить количество условий корректности (до 97% в отдельных случаях), причем эффективность метода растет с увеличением числа процессов и состояний в программе.

**Ключевые слова:** язык Reflex; процесс-ориентированное программирование; условия корректности; статический анализ; атрибуты

**Для цитирования:** Ищенко А.Д., Ануреев И.С. Применение статического анализа для сокращения количества порождаемых условий корректности для программ на языке Reflex. Труды ИСП РАН, 2026, том 38, вып. 6, часть 2, с. 21–40. DOI: 10.15514/ISPRAS-2026-38(6)-23.

**Благодарности:** Работа выполнена при поддержке Министерства образования и науки Российской Федерации, проект 125022803031-1.





# Using static analysis to reduce the number of verification conditions generated for Reflex programs

*Ishchenko A.D., ORCID: 0009-0005-6176-5917 <iartem2000@gmail.com>*
*Anureev I.S., ORCID: 0000-0001-9574-128X <anureev@gmail.com>*
*Institute of Automation and Electrometry SB RAS,*
*1, Akademika Koptyuga ave., Novosibirsk, 630090, Russia.*

**Abstract.** Control systems are widely used in safety-critical areas, which places high demands on software reliability and on proving that the software works correctly. To meet these demands, formal methods come into play – deductive verification in particular. That approach generates verification conditions based on the program's axiomatic semantics and its specifications. The trouble is, automatic generation often produces formulas that are redundant or overly complex, making them harder to verify and reducing the effectiveness of automated theorem provers. Domain-specific languages like Reflex partially address this issue. Reflex uses a process-oriented paradigm that simplifies program structure. But the semantics of Reflex constructs can sometimes generate even more verification conditions, especially irrelevant ones that correspond to unreachable paths in the program. In this work we propose a static analysis method aimed at reducing the number of verification conditions generated for Reflex programs. The method annotates program statements with a set of attributes that capture their semantic properties. First, attributes are computed for elementary statements, then aggregated for composite constructs. During verification condition generation, attributes accumulate along the execution path taken, and their consistency is checked. Incompatible attribute combinations reveal irrelevant verification conditions, which can then be discarded. Experimental evaluation on a set of test programs shows that our approach cuts down the number of verification conditions significantly – up to 97% in some cases – and the improvement grows as the number of processes and states in the program increases.

**Keywords:** Reflex; process-oriented programming; verification conditions; static analysis; attributes.

**For citation:** Ishchenko A.D., Anureev I.S. Using static analysis to reduce the number of verification conditions generated for Reflex programs. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 26, part 2, pp. 21–40. (in Russian). DOI: 10.15514/ISPRAS-2026-38(6)-23

**Acknowledgments:** This work was supported by the Russian Ministry of Education and Science, project 125022803031-1.

## *1. Введение*

Системы управления находят широкое применение в самых разных областях – от интернета вещей до промышленных программируемых контроллеров. Это предъявляет повышенные требования к их надежности и, в частности, к доказательству корректности используемого программного обеспечения.

Сегодня самым распространенным способом обеспечения надежности остается тестирование: системы прогоняют на цифровых моделях или на реальных объектах управления. Такой подход популярен благодаря своей простоте и относительной дешевизне. Однако он не может быть полным – невозможно охватить все мыслимые сценарии работы. В итоге даже широко используемые устройства, которые кажутся абсолютно безопасными, могут содержать редкие, но опасные ошибки.

Для верификации критически важного программного обеспечения (с точки зрения безопасности) применяют формальные методы. Они опираются на формальную семантику программы и ее требований, переводя проблему в область математических задач и тем самым упрощая ее решение. Один из таких методов – дедуктивная верификация [1]. Здесь требуется задать аксиоматическую семантику языка программирования в рамках некоторой системы логического вывода и формализовать требования. Затем программа вместе с предъявленными к ней требованиями преобразуется в набор логических формул. Этот этап называют генерацией условий корректности (УК). Для автоматизации процесса разрабатываются специальные генераторы, которые перебирают все возможные пути потока

управления и строят для них УК. Однако наивная реализация такого генератора может породить множество избыточных условий корректности – из-за путей, которые на самом деле недостижимы. Доказательство таких УК отнимает лишнее время, поэтому важно уметь сокращать их количество.

Для упрощения создания и верификации управляющих программ часто используют предметно-ориентированные языки. Один из них – язык Reflex [2], разработанный специально для написания программ управления. Он выполнен в стиле процесс-ориентированной парадигмы: программа представляет собой список процессов, каждый из которых последовательно выполняется раз в заданный период (интервал активации), формируя тем самым цикл управления. Процесс состоит из нескольких активных состояний, которые задает программист, а также двух неактивных – *stop* и *error*. В каждом состоянии содержится список инструкций: это могут быть обычные императивные операторы (например, условные) и специальные процесс-ориентированные операторы, служащие для взаимодействия между процессами. Использование Reflex меняет стиль программирования управляющих систем: в частности, условия корректности формулируются в терминах управляющих процессов и их состояний, что делает их концептуально проще. С другой стороны, из-за switch-case-подобной структуры процессов и состояний Reflex-программ число УК возрастает, причем некоторые из них оказываются ложноположительными – они соответствуют недостижимым в реальном выполнении путям.

В данной статье предлагается система статического анализа для программ на Reflex, которая выявляет такие условия корректности и тем самым позволяет оптимизировать процесс генерации УК. Она является развитием идей, предложенных в [3], вводит новые проверяемые атрибуты и правила их верификации, а также устраняет обнаруженные неточности.

## 2. Язык Reflex

Программа на языке Reflex состоит из процессов, их состояний, процесс-ориентированных операторов и обычных Си-подобных конструкций. Поскольку выполнение программы происходит в цикле управления, в котором активные процессы и их текущие состояния могут меняться, у каждого процесса имеется скрытый параметр – локальное время, отсчитывающее длительность пребывания в текущем активном состоянии. Значение этого параметра автоматически увеличивается в конце каждой итерации цикла управления на величину, равную интервалу активации. Прочитать его можно с помощью специального оператора *timeout t* {…}. Такой оператор может быть только последней инструкцией в активном состоянии. Он сравнивает значение параметра *t* с локальным временем процесса. Если локальное время оказалось больше или равно *t*, то выполняется список инструкций …, связанный с оператором.

Для взаимодействия между процессами и их состояниями предусмотрены следующие инструкции:

- *restart*, *start p* – переводят текущий процесс (или процесс *p*) в его первое состояние;
- *stop*, *stop p* – переводят текущий процесс (или процесс *p*) в состояние останова;
- *error*, *error p* – переводят текущий процесс (или процесс *p*) в состояние ошибки;
- *set state s*, *set next state* – переводят текущий процесс в состояние *s* или в состояние, следующее за текущим и обнуляет локальное время;
- *reset timer* – обнуляет локальное время текущего процесса;
- *process p in state k* – проверяет, находится ли процесс *p* в состоянии вида $k \in \{active, inactive, stop, error\}$.

Кроме того, в языке Reflex используются следующие конструкции в синтаксисе языка Си:

- арифметические и логические выражения;
- условные операторы if-else и переключатели switch-case.





В листинге 1 приведен пример программы для турникета, написанной на Reflex. Программа состоит из четырех процессов, запускаемых каждые 100 миллисекунд (интервал активации задается параметром *clock*). В ней демонстрируется использование операторов *start*, *stop*, *set state* и *timeout*.

```
program Turnstile{
    clock 100;
    bool passed;
    bool PdOut = inp[0];
    bool paid = inp[1];
    bool opened = inp[2];
    bool open  = out[0];
    bool rst = out[1];
    bool enter = out[2];
    process Init {
        state init {
            start Controller;
            start EntranceController;
            stop;}}
    process Controller{
        state isClosed{
            if (paid){
                open = true;
                passed = false;
                set next state;}}
        state minimalOpened{
            if (PdOut){
                     passed = true;}
            timeout 0t1s{
                if (passed) {
                    open = false;
                    set state isClosed;}
                else {set next state;}}}
        state isOpened{
            if (PdOut){
                open = false;
                set state isClosed;}
            timeout 0t9s{
                open = false;
                set state isClosed;}}}
    process EntranceController{
        state isClosed{
            if (opened){
                enter = true;
                set next state;}}
        state isOpened{
            if (!opened){
                enter = false;
                rst = true;
                start Unlocker;
                set state isClosed;}}}
    process Unlocker{
        state unlock{
            timeout 0t1s{
                rst = false;
                stop;}}}}
```

*Листинг 1. Пример Reflex программы.*
*Listing 1. Reflex program example.*

Программа принимает три входных сигнала: факт прохода пользователя (*PdOut*), оплата прохода (*paid*), состояние «турникет открыт» (*opened*). Выходных сигналов также три: открыть турникет (*open*), перезапустить платежное устройство (*rst*), подать сигнал разрешения прохода (*enter*). Выполнение начинается с запуска процесса *Init*, который, в свою очередь, запускает процессы *Controller* и *EntranceController*.

Процесс *Controller* управляет створками турникета и может находиться в трех состояниях: закрыт, кратковременно открыт и открыт. Из состояния закрыт в кратковременно открыт турникет переходит после получения сигнала об оплате, включает сигнал открытия створок и сбрасывает переменную *passed*. Во втором состоянии проверяется, прошел ли пользователь через турникет. Спустя секунду после перехода в это состояние, если пользователь прошел, турникет закрывается, а если нет – переходит в третье состояние. В третьем состоянии также проверяется факт прохода, но через 9 секунд турникет закрывается в любом случае.

Процесс *EntranceController* отвечает за сигнал прохода и перезапуск платежного устройства. В первом состоянии он ждет, пока створки турникета откроются, чтобы подать соответствующий сигнал и перейти во второе состояние. Во втором состоянии процесс ожидает закрытия двери: когда это происходит, он отключает сигнал, включает перезапуск платежного устройства, запускает процесс *Unlocker* и возвращается в первое состояние. Процесс *Unlocker* имеет единственное состояние, в котором по таймауту отключается перезапуск платежного устройства.

## 3. Статический анализ

Увеличение числа УК происходит при обработке трех категорий инструкций: циклов, условных конструкций и логических выражений. Циклы дробят условие на несколько частей из-за введения дополнительного инварианта, и сократить их количество можно лишь в том случае, если доказано, что на данном пути не будет выполнено ни одной итерации цикла. Ветвление в логических выражениях возникает из-за особенностей программной реализации логических операций: при определенных значениях левых операндов обработка правых операндов отменяется. Указанные случаи характерны для большинства языков программирования, и сокращение УК здесь возможно только при доказательстве узкоспециализированного для данной программы логического условия, что обычно требует применения SAT- и SMT-решателей и довольно затратно по времени.К условным конструкциям в языке Reflex относятся:

- *process* {…} – объявление процесса с набором состояний. Ветвление возникает на этапе выбора конкретного состояния, в котором находится текущий процесс;
- *timeout t* … – проверка двух случаев: когда таймаут *t* превышен и когда нет;
- *if* (*expr*) {…} *else* {…} – проверка истинности или ложности выражения `expr`;
- *switch* (*expr*) {*case* …} – проверка соответствия выражения `expr` разным меткам.

Третий и четвертый операторы – обычные Си-операторы, и для них в общем случае сокращение УК также возможно только путем доказательства того, что условное выражение на данном пути принимает строго определенное значение. Поэтому в данной работе мы сосредоточимся на статическом анализе первых двух типов. Для оператора *timeout* мы дополнительно рассматриваем только те случаи, когда параметр таймаута *t* является константой: в противном случае потребовалось бы доказывать, что переменная, задающая таймаут, никогда не опускается ниже периода цикла управления, что опять же требует SMT-решателей.

В листинге 2 показано одно из условий корректности для недостижимого пути, представленное в синтаксисе системы Isabelle/HOL, которое можно отбросить с помощью статического анализа. Это УК было получено при применении к программе из листинга 1 генератора условий корректности, представленного в [4].





Данное УК состоит из нескольких смысловых частей. Во-первых, в строке 2 указано, что в начале итерации цикла управления (состояние *st0*) выполнен инвариант. Затем в строках 3-5 описывается присваивание входным переменным некоторых неопределенных значений. Далее утверждается, что процесс *Init* находится в состоянии *init*, и идет обработка операторов этого состояния. Процесс *Controller* устанавливается в состояние *isClosed* (что соответствует его запуску), процесс *EntranceController* – в состояние *isClosed*, процесс *Init* – в состояние *stop*. После этого обрабатывается процесс *Controller*, причем утверждается, что он находится в состоянии *isClosed*. В этом состоянии переменная $inp_1$ имеет значение *false*. Далее утверждается, что процессы *EntranceController* и *Unlocker* находятся в состоянии *stop*. В конце применяется конструктор *toEnv*, который обозначает завершение текущей итерации цикла управления и переименование конечного состояния в $st_{final}$. Условие корректности требует доказать, что в конечном состоянии инвариант выполняется.

Путь, описываемый приведенным условием, недостижим, потому что при обработке процесса *Init* в строке 8 запускается процесс *EntranceController* (то есть устанавливается в свое первое состояние – *isClosed*), однако далее в строке 12 утверждается, что этот процесс находится в состоянии *stop*. Строки 8 и 12 противоречат друг другу.

```
lemma
  assumes base_inv:"(inv st0)"
  and st1:"(st1=(setVarBool st0 ''inp_2'' inp_2))"
  and st2:"(st2=(setVarBool st1 ''inp_1'' inp_1))"
  and st3:"(st3=(setVarBool st2 ''inp_0'' inp_0))"
  and st3_Init_state:"(getPstate st3 ''Init'')=''init''"
  and st4:"(st4=(setPstate st3 ''Controller'' ''isClosed''))"
  and st5:"(st5=(setPstate st4 ''EntranceController'' ''isClosed''))"
  and st6:"(st6=(setPstate st5 ''Init'' ''stop''))"
  and st6_Controller_state:"(getPstate st6 ''Controller'')=''isClosed''"
  and st6_if1:"(getVarBool st6 ''inp_1'')=False"
  and st6_EntranceController_state:"(getPstate st6
      ''EntranceController'')=''stop''"
  and st6_Unlocker_state:"(getPstate st6 ''Unlocker'')=''stop''"
  and st7:"(st7=(toEnv st6))"
  and st_final:"(st_final=st7)"
shows "(inv st_final)"
```

*Листинг 2. Пример недостижимого условия корректности.*
*Listing 2. Unreachable verification condition example.*

Перед анализом программа приводится к канонической форме, в которой краткие формы операторов заменяются на полные:

- *restart* → *start* **p**, где **p** – текущий процесс;
- *stop* → *stop* **p**, где **p** – текущий процесс;
- *error* → *error* **p**, где **p** – текущий процесс;
- *set next state* → *set state* **s**, где **s** – следующее состояние текущего процесса.

Результат статического анализа – набор атрибутов. Они используются при генерации условий корректности, чтобы не создавать УК для недостижимых путей. Если на некотором пути выполняется хотя бы одно из условий несовместности значений атрибутов (эти условия описаны в разделе 4), что означает недостижимость данного пути, то процесс порождения УК прекращается для всех путей, для которых этот путь является префиксом. Атрибуты определяются для объявлений процессов, состояний процессов и операторов, входящих в тела объявлений состояний. Обозначим **p**.A – значение атрибута A в объявлении процесса **p**,

**p**.**s**.A – значение атрибута A в объявлении состояния **s** процесса **p**, а **p**.**s**.**st**.*A* – значение атрибута *A* после выполнения оператора **st** в состоянии **s** процесса **p**.

Пусть *P* – множество всех процессов программы, $\Theta$ = {*start*, *stop*, *error*}, *bool* = {*true*,*false*}, $\Phi$ = {*active*,*inactive*,*stop*,*nonstop*,*error*,*nonerror*}.

Атрибуты делятся на базовые и производные. К базовым относятся:

1. *p.s.st.reset* ∈ *bool* – был ли сброшен таймер (по умолчанию false);
2. *p.s.st.stateChanged* ∈ *bool* – произошло ли изменение состояния (по умолчанию false);
3. *p.s.st.reachTo* := *S*, где *S* – множество состояний процесса *p*, в которые возможен переход из текущего состояния (по умолчанию ∅);
4. *p.s.st.procChange*:=$f \in P \to_p \Theta$, где *f*(*p′*)=*start*/*stop*/*error* означает, что выполнение оператора *st* запускает, останавливает или останавливает с ошибкой процесс *p′*. Функция *f* является частичной (по умолчанию ⊥);
5. *p.s.st.potProcChange*:=$f \in (P \times \Theta) \to$bool, где *f*(*p′*, *start*/*stop*/*error*) = *true* означает, что выполнение оператора *st* может запустить, остановить или остановить с ошибкой процесс *p′* (по умолчанию ⊥);
6. *p.s.st.procStatus* := $f \in P \to_p \Phi$, где *f*(*p′*)=*inactive* означает, что процесс *p′* не находится в активном состоянии (по умолчанию ⊥).

Эти атрибуты определяются при первом проходе по программе. На их основе затем вычисляются производные атрибуты:

1. *p.reachE* ∈ *bool* – достигает ли процесс *p* когда-либо состояния ошибки (по умолчанию false);
2. *p.reachS* ∈ *bool* – достигает ли процесс *p* когда-либо состояния останова (по умолчанию false);
3. *p.startS* ∈ *bool* – может ли процесс *p* начать выполнение в состоянии останова (по умолчанию true для всех процессов, кроме первого);
4. *p.s.reachFrom* := *S*, где *S* – множество состояний процесса *p*, из которых возможен переход в состояние *s* (по умолчанию ∅);
5. *p.group* := *i*, где *i* – номер группы, к которой отнесен процесс *p*. Два процесса принадлежат одной группе, если они всегда запускаются, останавливаются или останавливаются с ошибкой в рамках одной итерации цикла управления (по умолчанию *i* равен номеру процесса).

Общая схема статического анализа показана на рис. 1.

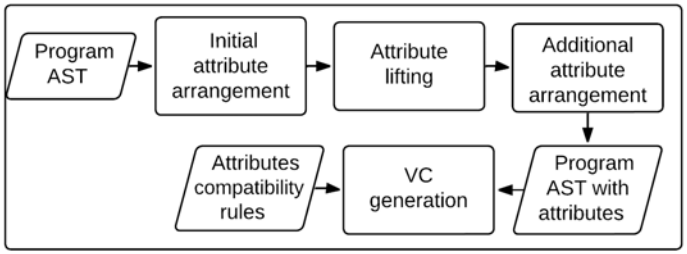



*Рис. 1. Схема статического анализа.*
*Fig. 1. Static analysis scheme.*





Алгоритм получает абстрактное синтаксическое дерево (AST) анализируемой программы и устанавливает начальные значения атрибутов (раздел 3.1) для соответствующих операторов. Затем применяется lifting-алгоритм (раздел 3.2), который переносит значения атрибутов *potProcChange*, *procChange*, *reset*, *stateChanged* и *reachFrom* на конструкции более высокого уровня. После этого определяются значения атрибутов *reachE*, *reachS*, *group* и *startS* (раздел 3.3). В заключение, алгоритму генерации УК передается AST с уже присвоенными атрибутами и правилами несовместности, что позволяет исключить порождение условий корректности для недостижимых путей.

## 3.1 Начальная расстановка атрибутов

Сначала определяются атрибуты для операторов *start*, *stop*, *error*, *reset timer* и *set state*. Для этого выполняется обход AST-дерева программы и устанавливаются значения атрибутов согласно алгоритму 1.

```
procedure AttributeInit(st)
   if st ≡ start p′ then
      st.procChange.add(p′,start); st.potProcChange.add(p′,start,true)
      if p ≡ p′ then
         st.reset := true
         st.reachTo = {s};
         if !s.isFirst() then st.stateChanged := true
      end if
   end if
   if st ≡ stop p′ then
      st.procChange.add(p′,stop); st.potProcChange.add(p′,stop,true)
      if p ≡ p′ then
         st.reset := true; st.reachTo = {stop}; st.stateChanged := true
      end if
   end if
   if st ≡ error p′ then
      st.procChange.add(p′,error); st.potProcChange.add(p′,error,true)
      if p ≡ p′ then
         st.reset := true; st.reachTo = {error}; st.stateChanged := true
      end if
   end if
   if st ≡ reset timer then st.reset := true
   if st ≡ set state s′ then
      st.reset := true; st.reachTo = {s′}
      if !(s≡s′) then st.stateChanged := true
   end if
end procedure
```

*Алгоритм 1. Алгоритм первичной расстановки атрибутов.*
*Algorithm 1. Initial attribute arrangement algorithm.*

Здесь *s* и *p* обозначают текущее состояние и текущий процесс соответственно, ≡ означает синтаксическое совпадение, *add* добавляет значение функции для указанных операндов, а *isFirst*() возвращает *true*, если состояние *s* является первым в своем процессе.

Атрибут *procStatus* вычисляется на уровне выражений. В языке Reflex имеется логическое выражение *process* **p** *in state* **s**, где **s** ∈ {*active*,*inactive*,*stop*,*error*}. На этапе обработки AST и построения графа потока управления создаются специальные узлы, соответствующие различным комбинациям значений таких логических выражений. Для этих узлов устанавливается значение атрибута *procStatus* в соответствии с выбранными комбинациями. В настоящее время реализованы два алгоритма генерации условий корректности: с учетом статуса процесса *procStatus* и без учета.

## 3.2 Lifting-алгоритм

Lifting-алгоритм «просачивает» значения атрибутов наверх, к операторам более высокого уровня. В общем случае подъем атрибута выполняется в двух ситуациях: либо атрибут со значением не по умолчанию принадлежит утверждению, входящему в линейную последовательность утверждений, либо атрибут присутствует на всех ветвях оператора ветвления (*if-else*, *switch-case*) с несколькими путями.

Для первого случая (последовательность операторов *st*) применяется алгоритм 2.

```
procedure LiftAttributes(st)
   pc := ⊥; ppc := ⊥
   for all st' in statements(st) do
      if st'.reset then st.reset := true
      if st'.stateChanged then st.stateChanged := true; st.reachTo :=st'.reachTo
      else st.reachTo := st.reachTo ∪ st'.reachTo
      for all p' in defined(pc) ∩ (defined(st'.potProcChange) \
      defined(st'.procChange)) do
         potStart := st'.potProcChange(p', start)
         potStop := st'.potProcChange(p', stop)
         potError := st'.potProcChange(p', error)
         if pc(p') = start ∧ (potStop ∨ potError) then pc.remove(p')
         if pc(p') = stop ∧ (potStart ∨ potError) then pc.remove(p')
         if pc(p') = error ∧ (potStart ∨ potStop) then pc.remove(p')
      end for
      pc := pc ∪ st'.procChange
      for all p' in defined(st'.procChange) do
         ppc := ppc.remove(p')
      end for
      ppc := ppc ∪ st'.potProcChange
   end for
   st.procChange := pc; st.potProcChange := ppc
end procedure
```

*Алгоритм 2. Алгоритм подъема атрибутов для последовательных операторов.*
*Algorithm 2. The algorithm for lifting attributes of consecutive operators.*

Здесь *statements*(*st*) возвращает последовательность операторов составного оператора *st*, *defined*(*f*) – множество процессов, для которых определена функция *f*, а *remove*(*p*) удаляет определение функции для процесса *p*. При объединении двух булевых троек итоговым значением в каждой позиции служит дизъюнкция соответствующих значений аргументов.

Для ветвящегося оператора *st* используется алгоритм 3.

Функция *first*(*l*) возвращает первый элемент списка *l*.

Для атрибутов *reachS*, *reachE* и *procStatus* lifting-алгоритм не применяется.

## 3.3 Означивание производных атрибутов

Атрибуты *reachE* и *reachS* показывают, может ли процесс когда-либо оказаться в состояниях *stop* и *error* после того, как его переведут туда другие процессы. Их начальные значения вычисляются по алгоритму 4.

Здесь *r* – анализируемая программа, *processes*(*r*) возвращает список ее процессов, *states*(*p*) – список состояний процесса *p*, а *body*(*s*) – список всех операторов в состоянии *s*.

Атрибут *startS* зависит от значения этого же атрибута у предыдущих процессов и от того, запускается ли текущий процесс в их первом состоянии. Он задается алгоритмом 5.

Поле *id* содержит порядковый номер процесса в том виде, в котором он объявлен в программе.





```
procedure LiftAttributes(st)
   procChange := (first(statements(st))).procChange
   ppc := ⊥; st.stateChanged := true
   for all st' in statements(st) do
      if !st'.reset then st.reset := false
      if !st'.stateChanged then st.stateChanged := false
      st.reachTo := st.reachTo ∪ st'.reachTo
      for all s' in defined(procChange) do
         if s' ∉ defined(st'.procChange) ∨ st'.procChange(s') ≠ procChange(s') then
            procChange := procChange.remove(s')
         end if
      end for
      ppc := ppc ∪ st'.potProcChange
   end for
   for all s' in defined(procChange) do
      ppc := ppc.remove(s')
      ppc := ppc.add(s', procChange(s'), true)
   end for
   st.procChange := procChange; st.potProcChange := ppc
end procedure
```

*Алгоритм 3. Алгоритм подъема атрибутов для параллельных операторов.*
*Algorithm 3. The algorithm for lifting attributes of parallel operators.*

```
procedure UpdateReachability(p)
   if ∃ s | stop ∈ p.s.reachTo ∨ ∃ p' ∈ processes(r) | p'.potProcChange(p, stop) =
   true then
      p.reachS := true
   end if
   if ∃ s | error ∈ p.s.reachTo ∨ ∃ p' ∈ processes(r) | p'.potProcChange(p, error) =
   true then
      p.reachE := true
   end if
end procedure
```

*Алгоритм 4. Алгоритм определения достижимости состояний останова и ошибки.*
*Algorithm 4. Reachability of stop and error states.*

```
procedure ResolveStartStates(r)
   for all p in processes(r) do
      if ∃ p' ∈ processes(r) | p'.id < p.id ∧ p'.startS = false ∧
      first(states(p')).procChange(p) = start ∧ ∄ p'' ∈ processes(r) | p'.id < p''.id <
      p.id ∧ ((p, stop) ∈ first(states(p'')).potProcChange ∨ (p, error) ∈
      first(states(p'')).potProcChange) then
         p.startS := false
      end if
   end for
end procedure
```

*Алгоритм 5. Алгоритм разрешения начального состояния процессов.*
*Algorithm 5. Resolution of starting process states.*

Поясним сказанное на примере (рис. 2). В первом состоянии процесса **P1** запускаются процессы **P2** и **P4**, а в первом состоянии **P4** запускается процесс **P3**. Тогда для процессов **P2** и **P4** атрибут *startS* получает значение *false*. Процесс **P3** остается со значением *true*, поскольку он объявлен раньше **P4**, и оператор *start P3* из первого состояния **P4** сможет запустить его только на следующей итерации цикла управления.

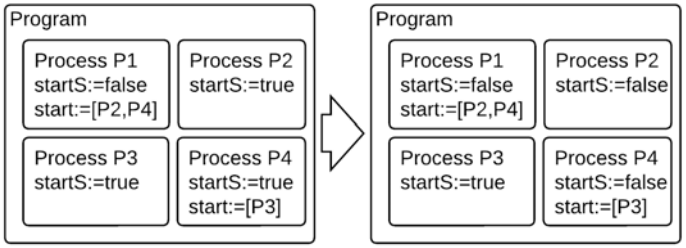



*Рис. 2. Изменение атрибута startS.*
*Fig. 2. Changing of startS attribute.*

Атрибут *group* у процессов – это упрощенный способ отслеживания соответствия между некоторыми состояниями разных процессов. Если у двух процессов значения *group* совпадают, значит они всегда одновременно запускаются, останавливаются и переходят в состояние ошибки. Для вычисления *group* применяются функции, описанные в алгоритме 6.

Здесь *lines*(*st*) возвращает список операторов верхнего уровня внутри составного оператора *st*: например, для *if* (*A*) *B else C* результатом будет последовательность из *B* и *C*. Функция *enumerate*(*s*) выдает пары (элемент *s*, номер от 1 до |*s*|). Глобальные переменные $p_{cur}$ и $s_{cur}$ обозначают текущий процесс и текущее состояние соответственно.

## 4. Правила несовместимости

После того как все атрибуты установлены, начинается генерация условий корректности. Исходный алгоритм генерации УК описан в [4]. Он выполняет обход графа потока управления в глубину, постепенно выстраивая список предусловий *acc*. Когда формирование очередного УК завершается, оно выводится, алгоритм выполняет откат к предыдущему незавершенному ветвлению, сбрасывает лишние предусловия и продолжает генерацию с нового места.

Использование атрибутов модифицирует описанный алгоритм. Дополнительно к *acc* заполняется список *passed*. Этот список описывает пройденный путь как последовательность языковых конструкций, что позволяет исключить некоторые невозможные продолжения этого пути на основе проверки совместности значений атрибутов конструкций. Если текущий путь признается недостижимым, выполняется откат (как при завершении УК), но само УК не выводится.

Модификация функции генерации УК для оператора *st* состояния *s* процесса *p* с проверкой достижимости и заполнением списка *passed* происходит по алгоритму 7.

Функция *add*(*passed*,*st*) добавляет структуру *st* в список *passed*, *updatePrec*(*acc*,*st*) расширяет сгенерированные предусловия *acc* согласно алгоритму генерации из работы [4], а *fallback*(*passed*,*acc*) выполняет откат, сокращая списки *passed* и *acc* до состояния на момент последнего незавершенного ветвления. Функция *checkComp*(*passed*,*p*,*s*,*st*) проверяет совместность атрибутов *st* с атрибутами конструкций, уже находящихся в пути *passed*, и возвращает *false*, если выполнено хотя бы одно из приведенных ниже правил несовместимости атрибутов. Также определим предикат *unchanged(q,K,x,y)* проверяющий, что между операторами *x* и *y* не существует оператора который бы переводил процесс *q* в одно из состояний множества *K*:

$$unchanged(q,K,x,y) \equiv \nexists\ p_m.s_m.st_m \mid x < p_m.s_m.st_m < y \wedge p_m.s_m.st_m.procChange(q) \in K$$

Разработанные правила несовместимости атрибутов можно разделить на несколько групп.





```
procedure SetsInter(setsSet, set)
   buffSet := ∅
   for all s' in setsSet do
      inter := s' ∩ set
      buffSet.add(inter)
      buffSet.add(s' \ inter)
   end for
   return buffSet
end procedure
procedure SetsDiv(setsSet, hPC, st)
   if st ≡ state … then s_cur := st
   nhPC := st.procChange ∪ hPC
   startPpred := {p' | nhPC(p') = start ∧ p'.id ≤ p_cur.id}
   setsSet := SetsInter(setsSet, startPpred)
   startPsucc := {p' | nhPC(p') = start ∧ p'.id > p_cur.id}
   setsSet := SetsInter(setsSet, startPsucc)
   stopPpred := {p' | nhPC(p') = stop ∧ p'.id ≤ p_cur.id}
   setsSet := SetsInter(setsSet, stopPpred)
   stopPsucc := {p' | nhPC(p') = stop ∧ p'.id > p_cur.id}
   setsSet := SetsInter(setsSet, stopPsucc)
   errorPpred := {p' | nhPC(p') = error ∧ p'.id ≤ p_cur.id}
   setsSet := SetsInter(setsSet, errorPpred)
   errorPsucc := {p' | nhPC(p') = error ∧ p'.id > p_cur.id}
   setsSet := SetsInter(setsSet, errorPsucc)
   for all nst in lines(st) do
      setsSet := SetsDiv(setsSet, nhPC, nst)
   end for
   return setsSet
end procedure
procedure BuildGroups(r)
   s1 := {p' | p'.startS = false}
   s2 := {p' | p'.startS = true}
   s := {s1, s2}
   for all p in processes(r) do
      p_cur := p
      s := SetsDiv(s, ⊥, p)
   end for
   for all (s_i, i) in enumerate(s) do
      for all p in s_i do
         p.group := i
      end for
   end for
end procedure
```

*Алгоритм 6. Алгоритм разбиения процессов на группы.*
*Algorithm 6. Process grouping algorithm.*

```
procedure CheckAndProcess(passed, p, s, st, acc)
   if checkComp(passed, p, s, st) then
      add(passed, st)
      updatePrec(acc, st)
   else
      fallback(passed, acc)
   end if
end procedure
```

*Алгоритм 7. Алгоритм проверки и обработки состояния.*
*Algorithm 7. State check and handling algorithm.*

## 4.1 Правила лишних состояний

Эти правила позволяют не обрабатывать состояния *stop* и *error* в тех случаях, когда процесс в них никогда не переходит. Проверка выполняется при обработке оператора состояния *s* процесса *p* и формулируется так:

1. $p.reachE = false \wedge p.s = error$;
2. $p.reachS = false \wedge p.startS = false \wedge p.s = stop$.

## 4.2 Правило обнуления времени

Это правило применяется к оператору таймаута, у которого параметр **t** задан константой:

$\exists\ p',s',st' \in passed \mid (p.s.st'.reset = true \vee p'.s'.\ st'.procChange(p)=start) \wedge st \equiv \{timeout\ t\ \ldots\} \wedge t>0$.

## 4.3 Правила комбинаций статусов процессов

Данный набор правил выявляет невозможные комбинации статусов процессов при условии, что между соответствующими точками не было изменений этих процессов.

1. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=active$
   $\wedge (p'.s'.st'.procStatus(p'')=inactive \vee p'.s'.st'.procStatus(p'')=stop \vee$
   $p'.s'.st'.procStatus(p'')=error)$
   $\wedge unchanged(p'', \{\text{start}\}, p'.s'.st', p.s.st)]$

2. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=inactive \wedge p'.s'.st'.procStatus(p'') = active$
   $\wedge unchanged(p'', \{\text{stop_error}\}, p'.s'.st', p.s.st)]$

3. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=stop \wedge$
   $(p'.s'.st'.procStatus(p'')=active \vee$
   $p'.s'.st'.procStatus(p'')=nonstop \vee p'.s'.st'.procStatus(p'')=error)$
   $\wedge unchanged(p'', \{\text{stop}\}, p'.s'.st', p.s.st)]$

4. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=error \wedge (p'.s'.st'.procStatus(p'')=active \vee$
   $p'.s'.st'.procStatus(p'')=nonerror \vee p'.s'.st'.procStatus(p'')=stop)$
   $\wedge unchanged(p'', \{\text{error}\}, p'.s'.st', p.s.st)]$

5. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=nonstop \wedge (p'.s'.st'.procStatus(p'')=stop)$
   $\wedge unchanged(p'', \{\text{start,error}\}, p'.s'.st', p.s.st)]$

6. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=nonerror \wedge (p'.s'.st'.procStatus(p'')=error)$
   $\wedge unchanged(p'', \{\text{start,stop}\}, p'.s'.st', p.s.st)]$

Поясним первое правило. Оно запрещает ситуацию, когда текущий оператор показывает, что некоторый процесс находится в активном состоянии ($p.s.st.procStatus(p'')=active$), но ранее уже был оператор, который указывал на его неактивность, останов или ошибку ($p'.s'.st'.procStatus(p'')=inactive \vee stop \vee error$), причем между ними нет оператора запуска этого процесса ($p_m.s_m.st_m.procChange(p'')= start$. Аналогичным образом интерпретируются остальные правила.





## 4.4 Правила статусов после изменений процессов

Эти правила определяют, какие статусы процессов не могут иметь место сразу после изменения состояния процесса.

1. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=active$
   $\wedge (p'.s'.st'.procChange(p'')= stop \vee p'.s'.st'.procChange(p'')= error)$
   $\wedge unchanged(p'', \{start\}, p'.s'.st', p.s.st)]$

2. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=inactive$
   $\wedge (p'.s'.st'.procChange(p'')= start)$
   $\wedge unchanged(p'', \{stop,error\}, p'.s'.st', p.s.st)]$

3. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=stop$
   $\wedge (p'.s'.st'.procChange(p'')= start \vee p'.s'.st'.procChange(p'')= error)$
   $\wedge unchanged(p'', \{stop\}, p'.s'.st', p.s.st)]$

4. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=error$
   $\wedge (p'.s'.st'.procChange(p'')= start \vee p'.s'.st'.procChange(p'')= stop)$
   $\wedge unchanged(p'', \{error\}, p'.s'.st', p.s.st)]$

5. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=nonstop$
   $\wedge (p'.s'.st'.procChange(p'')= stop)$
   $\wedge unchanged(p'', \{start,error\}, p'.s'.st', p.s.st)]$

6. $\exists\ p', s', st' \in passed \mid$
   $[p.s.st.procStatus(p'')=nonerror$
   $\wedge (p'.s'.st'.procChange(p'')= error)$
   $\wedge unchanged(p'', \{start,stop\}, p'.s'.st', p.s.st)]$

Первое правило, например, утверждает, что если текущий оператор считает процесс активным ($p.s.st.procStatus(p'')=active$), а ранее этот процесс был остановлен или переведен в состояние ошибки ($p'.s'.st'.procChange(p'')= stop \vee error$), и после этого его ни разу не запускали, то такая ситуация невозможна.

## 4.5 Правила текущего состояния процесса после изменения

Эта группа описывает, какие состояния процесса несовместимы с тем, как процесс изменялся ранее.

1. $p.s \neq first(p) \wedge \exists\ p',s',st' \in passed \mid p \in p'.s'.st'.procChange(p)=start$
   $\wedge unchaged(p,\{stop,error\}, p'.s'.st', \text{p.s})$;
2. $p.s \neq \text{stop} \wedge \exists\ p',s',st' \in passed \mid p \in p'.s'.st'.procChange(p)=stop$
   $\wedge unchaged(p,\{start,error\}, p'.s'.st', \text{p.s})$;
3. $p.s \neq \text{error} \wedge \exists\ p',s',st' \in passed \mid p \in p'.s'.st'.procChange(p)=errpr$
   $\wedge unchaged(p,\{start,stop\}, p'.s'.st', \text{p.s})$.

По сути, эти правила аналогичны правилам из раздела 4.3.

### 4.6 Правила текущего состояния процесса с учетом статусов

Данная группа описывает, какие состояния процесса невозможны в зависимости от того, какой статус у него был зафиксирован ранее.

1. $\exists\ p', s', st' \in passed \mid$

$[p'.s'.st'.procStatus(p)=active \wedge (p.s=stop \vee p.s=error)$
$\wedge\ unchaged(p,\{stop,error\},\ p'.s'.st',\text{p.s})]$
$\vee\ [p'.s'.st'.procStatus(p)=inactive \wedge p.s\neq stop \wedge p.s \neq error$
$\wedge\ unchaged(p,\{start\},\ p'.s'.st',\text{p.s})]$
$\vee\ [p'.s'.st'.procStatus(p)=stop \wedge p.s\neq stop$
$\wedge\ unchaged(p,\{start,error\},\ p'.s'.st',\text{p.s})]$
$\vee\ [p'.s'.st'.procStatus(p)=error \wedge p.s\neq error$
$\wedge\ unchaged(p,\{start,stop\},\ p'.s'.st',\text{p.s})]$
$\vee\ [p'.s'.st'.procStatus(p)=nonstop \wedge p.s= stop$
$\wedge\ unchaged(p,\{stop\},\ p'.s'.st',\text{p.s})]$
$\vee\ [p'.s'.st'.procStatus(p)=nonerror \wedge p.s=error$
$\wedge\ unchaged(p,\{error\},\ p'.s'.st',\text{p.s})]$

### 4.7 Правила группировки

Эти правила описывают ограничения на процессы, принадлежащие одной группе.

1. $\exists\ p', s' \in passed \mid p.group=p'.group \wedge (p.s = error \oplus p'.s' = error)$;
2. $\exists\ p', s' \in passed \mid p.group=p'.group \wedge (p.s = stop \oplus p'.s' = stop)$;
3. $\exists\ p', s' \in passed \mid p.group=p'.group \wedge first(p).reachFrom=\emptyset \wedge first(p).reachTo=\emptyset$
$\wedge\ first(p) \notin first(p).reachTo \wedge p.s=first(p) \wedge p'.s' \neq first(p')$;
4. $\exists\ p', s' \in passed \mid p.group=p'.group \wedge first(p').reachFrom=\emptyset \wedge first(p').reachTo=\emptyset$
$\wedge\ first(p') \notin first(p').reachTo \wedge p'.s'=first(p') \wedge p.s \neq first(p)$

Первое правило, например, требует, чтобы два процесса из одной группы одновременно либо оба находились в состоянии ошибки, либо оба не находились.

Данный алгоритм реализован как часть генератора условий корректности, размещенного на GitHub (сноска): https://github.com/bearhug15/ReflexVCG

## *5. Тестирование разработанной системы*

Разработанная система статического анализа была реализована и применена для генерации условий корректности на нескольких тестовых программах. В таблице 1 приведены результаты тестирования генерации УК для 20 программ по 4 тестам:

1. Генерация без статического анализа с учетом статуса процесса.
2. Генерация с использованием статического анализа с учетом статуса процесса.
3. Генерация без статического анализа и без учета статуса процесса.
4. Генерация с использованием статического анализа и без учета статуса процесса.

Для каждой программы и теста приведено среднее время по 10 запускам и порождаемое количество условий корректности. Полученные данные показывают, что наилучший результат достигается на программах с большим числом процессов, позволяя сокращать количество условий корректности до 26 раз (SimpleMicrogrid) и уменьшить время порождения до 5 раз (GasSeparator). Это объясняется тем, что многие правила несовместности зависят именно от взаимодействия между процессами. Таким образом, сравнение тестов 1 и 2 (аналогично 3 и 4) показывает, что при масштабировании программы





(увеличении количества процессов) статический анализ дает ощутимый выигрыш во времени при генерации УК.

Также из сравнения этих тестов видно, что при масштабировании программы количество порождаемых нерелевантных УК значительно возрастает. Соответственно, возрастают накладные расходы на их доказательство.

*Табл. 1. Результаты тестирования.*
*Table 1. Testing results.*

| Программа | Число проц. | Тест 1 | | Тест 2 | | Тест 3 | | Тест 4 | |
|---|---|---|---|---|---|---|---|---|---|
| | | Время (с) | УК. | Время (с) | УК. | Время (с) | УК. | Время (с) | УК. |
| HandDryer | 1 | 0.384 | 9 | 0.383 | 6 | 0.377 | 9 | 0.375 | 6 |
| TrainBarrier | 1 | 0.374 | 13 | 0.389 | 11 | 0.393 | 13 | 0.396 | 11 |
| TrafficLights | 1 | 0.408 | 14 | 0.413 | 12 | 0.396 | 14 | 0.392 | 12 |
| RevolvingDoor | 1 | 0.4 | 16 | 0.395 | 11 | 0.394 | 16 | 0.395 | 11 |
| BaggageCarousel | 1 | 0.405 | 19 | 0.412 | 12 | 0.397 | 13 | 0.394 | 8 |
| Escalator | 1 | 0.444 | 43 | 0.44 | 27 | 0.437 | 43 | 0.425 | 27 |
| Barrier | 2 | 0.439 | 73 | 0.423 | 33 | 0.416 | 73 | 0.408 | 33 |
| WashingMachine | 2 | 0.497 | 161 | 0.475 | 111 | 0.498 | 161 | 0.471 | 111 |
| Thermopot | 3 | 0.497 | 181 | 0.427 | 41 | 0.495 | 181 | 0.432 | 41 |
| WaterHeater | 3 | 0.569 | 361 | 0.454 | 72 | 0.565 | 361 | 0.448 | 72 |
| SmartLighting | 3 | 0.654 | 487 | 0.495 | 120 | 0.61 | 487 | 0.485 | 120 |
| Fridge | 4 | 0.672 | 751 | 0.463 | 82 | 0.657 | 751 | 0.476 | 82 |
| Turnstile | 4 | 0.717 | 1009 | 0.476 | 123 | 0.703 | 1009 | 0.477 | 123 |
| SimpleMicrogrid | 4 | 0.906 | 1513 | 0.498 | 56 | 1.223 | 1513 | 0.478 | 58 |
| Tram | 4 | 1.104 | 2017 | 0.652 | 342 | 0.974 | 2017 | 0.614 | 412 |
| BottleFilling | 4 | 1.095 | 2593 | 0.726 | 705 | 1.068 | 2593 | 0.716 | 875 |
| MassageChair | 4 | 1.911 | 6157 | 0.97 | 2063 | 1.703 | 6157 | 0.90 | 2063 |
| GasSeparator | 4 | 14.602 | 43473 | 2.93 | 6602 | 14.209 | 43473 | 2.705 | 6946 |
| Engine | 5 | 2.853 | 8321 | 0.727 | 633 | 2.469 | 8321 | 0.722 | 633 |
| VendingMachine | 5 | 4.484 | 17425 | 1.131 | 1866 | 3.601 | 16336 | 1.412 | 2206 |

## *6. Связанные работы*

Рост сложности программ ведет к значительному увеличению числа условий корректности, что становится серьезной проблемой для дедуктивной верификации. Как отмечают авторы работы [5], верификация киберфизических систем, включая программируемые логические контроллеры, сама по себе является одним из современных вызовов в области дедуктивной верификации, и в рассматриваемом нами случае эта проблема проявляется особенно остро. Более того, различные методы, направленные на упрощение УК, могут непреднамеренно усугубить ситуацию. Например, в работе [6] предлагается разбивать условия корректности на несколько отдельных условий, связка которых остается эквивалентной исходному. Хотя такие методы могут дать определенные преимущества для верификации программ, в случае Reflex они могут усложнить задачу. Сложность возникает из-за присущей языку Reflex абстракции конечных автоматов, порождающей граф потока управления с высокой связностью процессов и низкой сцепленностью состояний внутри этих процессов. Следовательно, расщепление одного условия может привести к фрагментации множества

условных путей, что требует автоматического отбрасывания нерелевантных условий уже на этапе генерации с опорой на различные виды статического анализа.

Кушо [7] предлагает технику сокращения числа условий корректности на основе графов двух типов: графа константной зависимости и графа предикатной зависимости. Эти графы позволяют проверять релевантность УК и удалять нерелевантные. Однако у подхода есть существенные недостатки: необходимость построения полного УК до проверки и зависимость от SAT-решателей. Такие зависимости могут значительно замедлить процесс генерации УК. В отличие от этого, наш подход опирается исключительно на синтаксический анализ, что проще и экономичнее по времени.

Другой взгляд на рассматриваемую проблему связан с определением недостижимых состояний. В этом контексте комбинация состояния процесса и значения таймера трактуется как одно состояние, а переходы между такими комбинациями – как переходы между состояниями на итерациях цикла управления. Для решения этой задачи предложено несколько методов [8-9]. Они основаны на трех шагах: 1) аппроксимация пространства достижимых состояний; 2) отладка; 3) обнаружение ложных решений. Изменение последнего шага позволяет варьировать точность методов, отбрасывая большее число недостижимых состояний. Преимущество этих методов – возможность учитывать дискретное время, которое в нашем случае эквивалентно итерациям цикла. Однако они требуют явного перечисления состояний, что может занимать много памяти, и опираются на медленные методы проверки моделей. Модификации, повышающие точность анализа, также существенно замедляют его.

Кроме того, данную проблему можно сформулировать как задачу достижимости для контекстно-свободного языка (CFL) и применить соответствующие методы анализа достижимости [10-11]. Учитывая, что достижимость каждого состояния процесса может ограничиваться различными операторами из предыдущих частей программы, этот подход напоминает вариант задачи о кратчайшем пути между всеми парами вершин (all-pairs S-path problem). Хотя этот метод применим в сценариях, где можно избавиться от ненужной обработки состояний *stop* и *error*, ему не хватает полноты, поскольку он не учитывает более сложное поведение программы.

Проблема, решаемая в данной работе, имеет некоторое сходство с широко известной проблемой экспоненциального взрыва размера условий корректности. Во многих работах [12-13] эта проблема решается за счет введения пассивной формы и выделения многократно повторяющихся частей УК, что позволяет только уменьшить размер условий корректности, но не их количество. В нашем же подходе мы рассматриваем совместность фрагментов путей программы, что дает возможность исключать УК с противоречивыми предусловиями. Кроме того, указанный подход предполагает конструирование единого условия корректности из конъюнкции УК, соответствующих разным путям. С учетом большого числа УК, порождаемых для Reflex-программ, такой подход неприменим.

Также сходство с нашей задачей имеет существующая в области символьного исполнения (symbolic execution) проблема недостижимых путей (infeasible paths). Ее решение обычно предполагает динамическое символьное исполнение [14], требующее применения SMT-решателей в каждой точке ветвления. Однако такой подход страдает от проблемы взрыва путей и является затратным по времени. Существуют различные оптимизации [15-16], направленные на ускорение проверок и оптимизацию использования ресурсов, но они не представляются применимыми для нашей задачи. Есть и другие подходы [17], решающие проблему экспоненциального роста числа путей при символьном исполнении, однако они ориентированы на эффективный поиск отдельного пути с заданными свойствами. В нашем случае эти техники не подходят, поскольку нам нужны все достижимые пути, а использование указанных методов для поиска недостижимых путей было бы неэффективным: в Reflex-программах, состоящих из большого числа процессов и состояний, недостижимые пути составляют большую их часть.





## *7. Заключение*

В статье предложен подход к сокращению числа условий корректности, соответствующих недостижимым путям, основанный на анализе атрибутов элементов пути. Разработан формальный набор атрибутов, характеризующих свойства отдельных участков пути выполнения, а также предложен набор эвристик, позволяющих выявлять несовместимые комбинации значений этих атрибутов. Это позволяет на ранних этапах анализа исключать недостижимые пути. В дальнейшем планируется расширение предложенного анализа за счет введения дополнительных атрибутов, а также разработка новых эвристик для выявления противоречивых комбинаций.

Также в настоящий момент разрабатывается операционная семантика языка Reflex, которая в дальнейшем позволит нам доказать непротиворечивость аксиоматической семантики и корректность статического анализа.

Отдельным направлением развития является углубленный анализ выражений в условиях операторов ветвления, таких как *if* и *switch*, с целью выявления логически невозможных комбинаций значений и путей выполнения. В частности, предполагается учитывать взаимосвязи между условиями, их взаимную исключаемость и влияние на пространство состояний программы.

Перспективным представляется также исследование возможностей автоматического вывода атрибутов и эвристик на основе анализа программных шаблонов, а также их адаптации к различным классам программ. В совокупности это может привести к созданию более универсального и масштабируемого инструментария для сокращения пространства путей при формальной верификации.

## Список литературы / References


[1]. Hähnle R., Huisman M. Deductive software verification: from pen-and-paper proofs to industrial tools. Computing and Software Science: State of the Art and Perspectives, 2019, pp. 345-373. DOI: 10.1007/978-3-319-91908-9_18.

[2]. Zyubin V.E., Liakh T.V., Rozov A.S. Reflex language: a practical notation for cyber-physical systems. System Informatics, 2018, no. 12, pp. 85-104. DOI: 10.31144/si.2307-6410.2018.n12.p85-104.

[3]. Ishchenko A. Enhancing Verification Condition Generation for Reflex Programs Through Simple Static Analysis. System Informatics, 2024, vol. 25, pp. 1-10.

[4]. Ishchenko A.D., Anureev I.S. Verification Condition Generator for Revised Reflex Language Using Isabelle/HOL. 2025 IEEE 26th International Conference of Young Professionals in Electron Devices and Materials (EDM). IEEE, 2025, pp. 1440-1445.

[5]. Hähnle R., Huisman M. 24 challenges in deductive software verification. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements, 2017, pp. 37-41.

[6]. Leino K.R.M., Moskal M., Schulte W. Verification condition splitting. Submitted manuscript, September 2008.

[7]. Couchot J.-F., Giorgetti A., Stouls N. Graph Based Reduction of Program Verification Conditions. arXiv preprint, 2009. DOI: 10.48550/arXiv.0907.1357.

[8]. Berryhill R., Veneris A.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, pp. 1298-1311.

[9]. Suda M. Property Directed Reachability for Automated Planning. Journal of Artificial Intelligence Research, 2014, vol. 50, pp. 265-319.

[10]. Reps T. Program Analysis via Graph Reachability. Information and Software Technology, 1998, vol. 40, no. 11-12, pp. 701-726.

[11]. Zhang Q., Lyu M.R., Yuan H., Su Z. Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis. Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, 2013, pp. 435-446.

[12]. Flanagan C., Saxe J.B. Avoiding Exponential Explosion: Generating Compact Verification Conditions. Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, 2001, pp. 193-205.

[13]. Grigore R., Charles J., Fairmichael F., Kiniry J. Strongest Postcondition of Unstructured Programs. Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs, 2009, pp. 1-7.
[14]. Khurshid S., Păsăreanu C.S., Visser W. Generalized Symbolic Execution for Model Checking and Testing. International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2003, pp. 553-568.
[15]. Delahaye M., Botella B., Gotlieb A. Infeasible Path Generalization in Dynamic Symbolic Execution. Information and Software Technology, 2015, vol. 58, pp. 403-418. DOI: 10.1016/j.infsof.2014.07.012.
[16]. Saumya C., Gangaraju R., Sundararajah K., Kulkarni M. Targeted Control-flow Transformations for Mitigating Path Explosion in Dynamic Symbolic Execution. arXiv preprint, 2023. DOI: 10.48550/arXiv.2308.01554 [cs.SE].
[17]. Xiao X., Zhang X.-S., Li X.-D. New Approach to Path Explosion Problem of Symbolic Execution. 2010 First International Conference on Pervasive Computing, Signal Processing and Applications. IEEE, 2010, pp. 301-304.

## *Информация об авторах / Information about authors*

Артем Дмитриевич ИЩЕНКО – инженер-исследователь Института Автоматики и Электрометрии с 2025 года, аспирант Новосибирского Государственного Университета. Сфера научных интересов: статический анализ, дедуктивная верификация.

Artem Dmitrievich ISHCHENKO – research engineer at the Institute of Automation and Electrometry since 2025, post-graduate student at Novosibirsk State University. Research interests: static analysis, deductive verification.

Игорь Сергеевич АНУРЕЕВ – старший научный сотрудник Института Автоматики и Электрометрии. Сфера научных интересов: семантика программных систем и компьютерных языков, спецификация и верификация программных систем и программных моделей, автоматизированное доказательство теорем.

Igor Sergeevich ANUREEV – senior researcher at the Institute of Automation and Electrometry. Research interests: semantics of program systems and computer languages, specification and verification of program systems and program models, automated theorem proving.