Моделирование и анализ информационных систем. Т. 26, № 4 (2019), с. 475–487
Modeling and Analysis of Information Systems. Vol. 26, No 4 (2019), pp. 475–487

Computing methodologies and applications

DOI: 10.18255/1818-1015-2019-4-475-487

УДК 04.423.42, 004.434, 681.51

Операционная семантика аннотированных Reflex программ

Ануреев И. С.

Поступила в редакцию 12 сентября 2019
После доработки 15 ноября 2019
Принята к публикации 27 ноября 2019

Аннотация. Reflex — процесс-ориентированный язык, который обеспечивает разработку простого в обслуживании управляющего программного обеспечения для программируемых логических контроллеров. Язык был успешно использован в нескольких системах управления с повышенными требованиями к надежности, например, в системе управления печью для выращивания монокристаллов кремния и в комплексе контроля радиоэлектронной аппаратуры. В настоящее время основной целью языкового проекта Reflex является разработка методов формальной верификации для Reflex программ для того, чтобы гарантировать повышенную надежность создаваемого на его основе программного обеспечения. В статье представлена формальная операционная семантика Reflex программ, расширенных аннотациями, описывающими формальную спецификацию программных требований, как необходимый базис для применения таких методов. Дан краткий обзор языка Reflex и приведен простой пример его использования — управляющая программа для сушилки рук. Определены понятия окружения и переменных, разделяемых с окружением, позволяющие абстрагироваться от конкретных портов ввода/вывода. Определены типы аннотаций, задающие ограничения на значения переменных при запуске программы, ограничения на окружение (в частности, на объект управления), инварианты цикла управления, пред- и постусловия внешних функций, используемых в Reflex программах. Аннотированный Reflex также использует стандартные аннотации `assume`, `assert` и `havoc`. Операционная семантика аннотированных Reflex программ использует глобальные часы и локальные часы отдельных процессов, время которых измеряется в количестве итераций цикла управления, для моделирования временных ограничений на исполнение процессов в определенных состояниях. Она хранит полную историю изменений значений разделяемых переменных для более полного описания временных свойств программы и ее окружения. Семантика учитывает бесконечность цикла выполнения программы, логику управления переходами процессов из состояния в состояние и взаимодействие процессов между собой и с окружением. Расширение формальной операционной семантики языка Reflex на аннотации упрощает доказательство корректности разрабатываемого авторами трансформационного подхода к дедуктивной верификации Reflex программ, трансформирующего аннотированную Reflex программу к аннотированной программе на сильно ограниченном подмножестве языка C, за счет сведения сложного доказательства сохранения истинности требований к программе при трансформации к более простому доказательству эквивалентности исходной и результирующей аннотированных программ относительно их операционных семантик.

Ключевые слова: операционная семантика, язык Reflex, система управления, управляющее программное обеспечение, программируемый логический контроллер, аннотация, аннотированная программа

Для цитирования: Ануреев И. С., "Операционная семантика аннотированных Reflex программ", *Моделирование и анализ информационных систем*, **26**:4 (2019), 475–487.

Об авторах:
Ануреев Игорь Сергеевич, orcid.org/0000-0001-9574-128X, канд. физ.-мат. наук, с.н.с.,
Институт систем информатики им. А.П. Ершова СО РАН, Институт автоматики и электрометрии СО РАН,
пр. Акад. Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: anureev@iis.nsk.su

Благодарности:
Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 17-07-01600, а также в рамках темы госзадания ИАиЭ СО РАН (№ АААА-А17-11706061006-6).

Введение

Многие системы управления, в частности, в области промышленной автоматизации, основаны на промышленных программируемых логических контроллерах (ПЛК), обладающих следующими особенностями: они по своей природе являются открытыми (то есть взаимодействуют с внешней средой), реактивными (имеют управляемое событиями поведение) и параллельными (должны обрабатывать многочисленные асинхронные события). Эти особенности привели к использованию специальных языков при разработке управляющего программного обеспечения, например, языков стандарта IEC 61131-3 [1], которые являются наиболее популярными в области программирования ПЛК. Однако поскольку сложность управляющего программного обеспечения возрастает, а качество становится более приоритетным, 35-летняя технология, основанная на подходе IEC 61131-3, не в состоянии удовлетворить современные требования [2].

Reflex – это предметно-ориентированный язык с C-подобным синтаксисом для области управляющего программного обеспечения, основанный на концепциях процесса и состояния процесса как программного кода и созданный как альтернатива языкам стандарта IEC 61131-3. Программа на языке Reflex описывается как множество взаимодействующих процессов. Язык имеет специализированные инструкции для управления процессами и их состояниями и инструкции для работы с временными интервалами. Эти средства поддерживают парадигму событийно-ориентированного программирования, в которой выполнение программы определяется событиями (в том числе, временными событиями). Он также имеет инструкции, связывающие переменные программ на этом языке с физическими портами ввода/вывода. Reflex предполагает выполнение программы на основе цикла управления с фиксированным временем итерации и строгую инкапсуляцию зависимых от платформы подпрограмм ввода-вывода в библиотеку, что является широко применяемой техникой в системах управления, созданной на основе IEC 6113-3. Для обеспечения простоты поддержки и межплатформенной переносимости генерация исполняемого кода осуществляется в два этапа: транслятор для языка Reflex генерирует C-код, а затем C-компилятор создает исполняемый код для целевой платформы.

В настоящее время проект Reflex сфокусирован на средствах разработки программного обеспечения для систем управления, имеющих повышенные требования к безопасности. Благодаря платформенной независимости язык Reflex легко интегрируется с LabVIEW [3]. Это позволяет разрабатывать программное обеспечение, сочетающее событийную ориентированность с развитым графическим пользовательским интерфейсом, удаленными датчиками и актуаторами, устройствами

с поддержкой LabVIEW и т. д. Используя гибкость LabVIEW был разработан набор симуляторов объектов управления для целей обучения [4]. Основанные на LabVIEW симуляторы включают в себя 2D-анимацию, инструменты для отладки и языковую поддержку для обучения разработке управляющего программного обеспечения. Одним из результатов, полученным в этом направлении, является набор инструментов динамической верификации на основе LabVIEW для Reflex программ. Динамическая проверка рассматривает программное обеспечение как черный ящик и проверяет его соответствие требованиям, наблюдая за поведением программного обеспечения во время выполнения на наборе тестовых случаев. Хотя такая процедура может помочь обнаружить наличие ошибок в программном обеспечении, она не может гарантировать их отсутствие [5].

В отличие от методов тестирования и динамической верификации, методы формальной верификации являются единственным способом обеспечить требуемые свойства программного обеспечения. Предложенный в [6] трансформационный подход к дедуктивной верификации Reflex программ базируется на алгоритме трансформации [7], сводящем дедуктивную верификацию аннотированных Reflex программ к дедуктивной верификации очень ограниченного подмножества аннотированных C программ, и алгоритме генерации условий корректности для этого подмножества C программ, основанном на исчислении сильнейшего постусловия. Доказательство корректности этих алгоритмов требует, чтобы исходная и результирующая программы имели формальную семантику. В этой статье мы представляем формальную операционную семантику аннотированных Reflex программ.

1. Введение в язык Reflex

Синтаксис языка Reflex демонстрируется здесь на простом примере программы, управляющей сушилкой для рук (Листинг 1). Программа использует вход от ИК-датчика, указывающего на присутствие рук под сушилкой, и управляет вентилятором и обогревателем с помощью совместного выходного сигнала. Формальное определение синтаксиса языка Reflex в EBNF можно найти [8].

```
PROGR HandDryerController {

 INIT <formula 1>;

 ENVIRONMENT <formula 2>;

 INVARIANT <formula 3>;

 TACT 100;
 CONST ON 1;
 CONST OFF 0;
 /*============================*/
 /* I/O ports specification    */
 /* direction, name, address,  */
 /* offset, size of the port   */
 /*============================*/
 INPUT  SENSOR_PORT  0 0 8;
 OUTPUT ACTUATOR_PORT 1 0 8;

 /*============================*/
 /* processes definition       */
 /*============================*/
 PROC Controller {
  /*===== VARIABLES =============*/
  BOOL HANDS = {SENSOR_PORT[1]} FOR ALL;
  BOOL DRYER = {ACTUATOR_PORT[1]} FOR ALL;
```

```
  /*===== STATES ================*/
  STATE Waiting {
   IF (HANDS == ON) {
    O_DRYER = ON;
    SET NEXT;
   } ELSE DRYER = OFF;
  }
  STATE Drying {
   IF (HANDS == ON) RESET TIMEOUT;
   TIMEOUT 10;
   SET STATE Waiting;
  }
 } /* PROC */
}  /* PROGR */
```

Листинг 1. Программа управления сушилкой для рук
Listing 1. Hand Dryer Control Program

На языке Reflex программа представляется как множество взаимодействующих процессов. Определение процесса `PROC <process name> <process body>` начинается с ключевого слова `PROC`, за которым следуют имя и тело процесса.

Выполнение программы суть повторяющиеся итерации цикла управления, где каждая итерация заключается в последовательном выполнении процессов программы (в порядке их определения в программе). Каждая итерация имеет фиксированное время выполнения в миллисекундах (период выполнения), задаваемое инструкцией `TACT`.

Тело процесса состоит из объявлений переменных и определений состояний. Определение состояния процесса `STATE <state name> <state body>` состоит из имени и тела состояния. Последнее специфицирует действие, выполняемое процессом в этом состоянии. Для каждого процесса неявно определены два дополнительных пассивных состояния `STOP` и `ERROR`. Эти состояния называются пассивными, так как никаких действий в них не выполняется. Состояние `STOP` соответствует остановке или приостановке выполнения процесса. Состояние `ERROR` соответствует ошибочному выполнению процесса. Все остальные состояния процесса являются активными.

Тело состояния определяется как блок (оператор последовательной композиции), включающий операторы присваивания, условный оператор if, вложенные блоки, операторы управления состояниями процессов и операторы таймаута.

Состояние, определяемое первым в теле процесса, называется стартовым состоянием этого процесса. Первый процесс, определенный в тексте программы, является единственным активным процессом при ее запуске. Он устанавливается в свое стартовое состояние. Остальные процессы устанавливаются при запуске программы в состояние `STOP`.

Синтаксис и семантика Reflex выражений идентичны синтаксису и семантике C выражений, за исключением фиксированного порядка вычисления аргументов операций и функций (слева направо) и специфических для Reflex логических операций над состояниями, называемых предикатами активности. Операторы присваивания, условные операторы и блоки имеют синтаксис и семантику, как в языке C (несущественные отличия заключаются в том, что присваивание рассматривается в качестве оператора в языке Reflex, т.е.не допускаются вложенные присваивания, и все переменные Reflex программы глобальны). Поэтому ниже мы опишем только те инструкции, которые специфичны для Reflex.

Каждый процесс имеет локальные часы, которые отсчитывают дискретное вре-

мя, измеряемое в количестве итераций цикла управления. Один тик локальных часов соответствует одной итерации цикла управления.

Оператор `SET STATE` s; устанавливает текущий процесс в состояние s для следующей итерации цикла управления. Этот и другие операторы, изменяющие состояние процесса (даже на то же самое состояние), сбрасывают время локальных часов этого процесса в ноль. Оператор `SET NEXT`; устанавливает текущий процесс в состояние, которое следует за текущим состоянием в теле процесса. Его применение к текущему состоянию, которое определяется последним в теле процесса, является синтаксической ошибкой.

Операторы `START PROC` p; и `STOP PROC` p; запускают и останавливают процесс p. Они устанавливают p в его стартовое состояние и в состояние `STOP` соответственно.

Операторы `RESTART`;, `STOP`; и `ERROR`; запускают и останавливают (нормальным образом и с ошибкой) текущий процесс. Они устанавливают этот процесс в его стартовое состояние, состояние `STOP` и состояние `ERROR` соответственно.

Процессы могут проверять, находятся ли другие процессы в активных или пассивных состояниях, используя условные операторы совместно с предикатами активности, например:

```
IF (PROC <process name> IN STATE ACTIVE) { ... }
```

Операторы таймаута осуществляют контроль времени выполнения процесса в определенном состоянии. Оператор `RESET TIMEOUT`; обнуляет локальные часы текущего процесса. Оператор `TIMEOUT <clocks num> <statement>` обнуляет локальные часы текущего процесса и выполняет оператор `<statement>` в том случае, если время на локальных часах достигло значения `<clocks num>` и ничего не делает в противном случае. Этот оператор может использоваться только один раз в теле состояния и должен быть последним оператором в теле состояния.

Тело процесса может содержать определения переменных

```
<type> <variable name> = {<port name>[<bit number>]} <scope>;
```

задающие такие характеристики переменной как тип, имя, порт (с которым она связывается), номер бита в порту (соответствующего значению переменной) и область видимости.

Поддерживаемыми типами являются `BOOL` для булевских значений, а также целочисленные и вещественные типы `INT`, `SHORT`, `LONG`, `FLOAT`, `DOUBLE`, определяемые как в языке C. Область действия `FOR ALL` указывает на то, что эта переменная может использоваться любыми процессами в программе. Привязка к порту позволяет считывать значение из входного порта в переменную и записывать значение переменной в выходной порт.

Декларации портов, размещаемые перед определениями процессов, задают характеристики портов, используемых в программе,

```
<direction> <port name> <base address> <offset> <size in bits>;
```

такие как направление (входной и выходной порты), имя, базовый адрес, смещение и размер в битах.

Важной особенностью переменных, связанных с портами, является то, что все операции чтения и записи для этих переменных имеют двойную буферизацию. Значения портов ввода/вывода считываются один раз за итерацию цикла управления, и каждое значение сохраняется в двух экземплярах — один для операций чтения

и один для операций записи. Новые значения для выходных портов устанавливаются и отправляются на внешние устройства в конце итерации. Таким образом, все процессы читают одни и те же значения портов, даже если они изменяются в итерации.

Объявления констант `CONST <constant name> <constant value>;` определяют имена и значения констант.

Формальная спецификация требований к Reflex программе задается специальными инструкциями, называемыми аннотациями. Пусть `<formula>` — формула на языке многосортной логики первого порядка.

Аннотация `INIT <formula>;` описывает ограничения на значения переменных при запуске программы.

Аннотация `ENVIRONMENT <formula>;` описывает ограничения на значения, которые окружение (например, объект управления) может передавать в программу (записывать в переменные программы) через входные порты.

Аннотация `INVARIANT <formula>;` описывает инварианты цикла управления, т. е. свойства, которые должны выполняться на каждой итерации цикла управления. В частности, эта аннотация описывает связи между значениями, считываемыми с входных портов, и значениями, записываемыми в выходные порты после очередной итерации.

Примеры этих трех видов аннотаций для программы управления сушилкой для рук можно найти в [6]. Программа, включающая аннотации, называется аннотированной программой.

Reflex программа может использовать в выражениях вызовы внешних C функций `<function name>(<expression 1>, ..., <expression n>)`. Для каждой такой функции в программе должен быть описан ее прототип,

```
<function type> <function name>
 (<parameter type 1> <parameter name 1>, ...,
  <parameter type n> <parameter name n>) {
 REQUIRES <formula 1>;
 ENSURES <formula 2>;
};
```

задающий ее имя, тип, параметры и их типы, пред- и постусловия.

Аннотация `REQUIRES <formula 1>;` описывает ограничения на параметры функции (предусловие функции).

Аннотация `ENSURES <formula 2>;` описывает ограничения на значение, возвращаемое функцией (постусловие функции). Формула `<formula 2>` содержит переменную `<function name>`, которая ссылается на значение, возвращаемое функцией.

Reflex программа может также включать следующие виды аннотаций.

Аннотация `ASSUME <formula>;` завершает программу со значением `IGNORE`, если формула `<formula>` ложна и ничего не делает, в противном случае. Значение `IGNORE` на некотором пути выполнения программы означает, что этот путь не учитывается при доказательстве корректности программы.

Аннотация `ASSERT <formula>;` завершает программу со значением `FAIL`, если формула `<formula>` ложна и ничего не делает, в противном случае. Значение `FAIL` на некотором пути выполнения программы означает, что программа выполняется некорректно на этом пути.

Аннотация `HAVOC <variable name>;` присваивает произвольное значение переменной `<variable name>` в соответствие с ее типом. Аннотация

```
HAVOC <variable name 1>, ..., <variable name n>;
```

сводится к последовательному выполнению аннотаций

```
HAVOC <variable name 1>; ...; HAVOC <variable name n>;
```

Определим операционную семантику Reflex программ.

2. Операционная семантика аннотированных Reflex программ

Пусть N — множество натуральных чисел (включая 0). Пусть U^* — множество всех последовательностей из элементов множества U, $|u|$ и $u[i]$ обозначают длину и i-й элемент последовательности $u \in U^*$ соответственно. Пусть $< u_1, \ldots, u_m >$ обозначает последовательность элементов u_1, ..., u_m и $con(w_1, \ldots, w_n)$ — конкатенацию последовательностей w_1, ..., w_n.

Для множества A всех Reflex программ определим: множества T, H и E типов, операторов и выражений; множество $val(t)$ всех значений типа $t \in T$; значение `OK` как нормальное завершение Reflex инструкции, множество Γ всех значений такое, что $\cup_{t \in T} val(T) \cup \{\texttt{OK}, \texttt{IGNORE}, \texttt{FAIL}\} \subseteq \Gamma$.

Для каждой программы $\alpha \in A$ определим:

- ее окружение π;
- множество $P = \{p_1, \ldots, p_n\}$ процессов ($P \subseteq \Gamma$);
- множество Θ состояний процессов ($\Theta \subseteq \Gamma$);
- множество F функций ($F \subseteq \Gamma$);
- множество $V = V_s \cup V_l$ переменных ($V \subseteq \Gamma$);
- множество $V_s = V_i \cup V_o$ разделяемых переменных (они разделяются с π);
- множество V_l локальных переменных ($V_s \cap V_l = \emptyset$);
- множество V_i входных переменных (окружение π может только писать в эти переменные);
- множество V_o выходных переменных (окружение π может только читать из этих переменных);
- $V_i \cap V_o = \emptyset$;
- функцию $vt \in V \to T$, связывающую переменные с их типами;
- функцию $pso \in P \to \Theta^*$, связывающую процессы с упорядоченными последовательностями их состояний (состояния в $pso(p)$ перечисляются в порядке, в котором они определяются в p);
- упорядоченную последовательность $po \in P^n$ процессов (процессы в po перечисляются в порядке, в котором они определяются в α);

- функцию $psv \in \Theta \to H$, связывающую состояния процессов с их телами;
- функцию $ftype \in F \to T$, связывающую внешние функции с их типами;
- функцию $fpar \in F \to V_l^*$, связывающую внешние функции Reflex программы с их параметрами;
- функцию $ftpar \in F \to T^*$, связывающую внешние функции Reflex программы с типами их параметров;
- функции $fpre, fpost \in F \to H$, связывающую функции с их пред- и постусловиями;
- формулы ini, env и inv в аннотациях `INIT`, `ENVIRONMENT` и `INVARIANT` соответственно.

Для каждой программы α мы считаем, что следующие ограничения выполняются при определении операционной семантики:

1. Программа α корректна определена.
2. Информация о портах программы α и сопоставление переменных с портами не принимается в расчет, так как она относится к взаимодействию с физическими устройствами. Вместо этого программа α взаимодействует с абстрактным окружением через входные и выходные переменные.
3. Области действия переменных в α не принимаются в расчет, так как они определяют только корректный доступ к переменным, который гарантируется для корректно-определенных программ.
4. Отсутствует перегрузка имен переменных, состояний процессов, внешних функций и параметров внешних функций. Это ограничение достигается переименованием перегруженных имен. Поэтому мы считаем, что для каждой внешней функции `<function name>` параметры этой функции, задаваемые в ее прототипе, и переменная `<function name>` являются локальными переменными программы α. Типы этих переменных определяются прототипом функции.
5. Программа α не содержит деклараций констант и декларации такта. Это ограничение достигается выполнением C-подобных макроподстановок.

Операционная семантика языка Reflex определяется системой переходов $(S, S_i, \to_R, \to_C)$, где S — множество состояний, $S_i \subseteq S$ — множество начальных состояний, $\to_R$ и $\to_C$ — отношения переходов для Reflex инструкций и внешних C функций соответственно. Имеются несколько решений для описания операционной семантики языка C [9–13]. Поэтому мы фокусируемся на операционной семантике Reflex инструкций (отношении переходов $\to_R$), считая, что отношение $\to_C$ определяется с помощью одного из этих подходов (наш подход наиболее близок к подходу [12]).

Состояние $s \in S$ определяется как кортеж $(gc, cp, lc, ps, vv, ih, oh)$, который включает:

- глобальные часы $gc \in N$;

- текущий процесс $cp \in P$;
- функцию $lc \in P \to N$, связывающую процессы с их локальными часами, при этом все часы отсчитают время в тиках (один тик соответствует одной итерации по циклу управления);
- функцию $ps \in P \to \Theta$, связывающую процессы с их текущими состояниями;
- функцию $vv \in V \to \Gamma$, связывающую переменные с их значениями;
- входную историю $ih \in V_i \to \Gamma^*$ ($ih(v)[i]$ — значение переменной $v \in V_i$, записанное π во время i-го тика глобальных часов gc);
- выходную историю $oh \in V_o \to \Gamma^*$ ($oh(v)[i]$ — значение переменной $v \in V_o$, прочитанное π во время i-го тика глобальных часов gc).

Пусть $s.gc$, ..., $s.oh$ обозначает доступ к компоненте gc, ..., oh состояния s. Состояние s является начальным, если выполнены следующие свойства:

- $gc = 0$;
- $lc(p) = 0$ для каждого $p \in P$;
- $ps(po.1) = pso(po.1)$;
- $ps(po.i) = stop$ для каждого $1 < i \leq n$;
- $|ih(v)| = 0$ для каждого $v \in V_i$;
- $|oh(v)| = 0$ для каждого $v \in V_o$.

Отношение перехода $\to_R \in \Xi_R \times \Xi_R$ определяется на множестве $\Xi_R = I_R \times S$ конфигураций Reflex программ, где I_R — множество всех Reflex инструкций, обладающее следующим свойством: $T \cup H \cup E \cup \Gamma \subseteq I_R$. Таким образом, инструкциями могут быть типы (из T), операторы (из H) или выражения (из E), встречающиеся в Reflex программах, значения (из Γ), возвращаемые Reflex инструкциями в операционной семантике языка Reflex, а также вспомогательные инструкции, используемые (и определяемые) в правилах операционной семантики языка Reflex (см., например, метаприсваивание ниже).

Операционная семантика Reflex инструкций определяется правилами перехода. Многие из этих правил используют метаприсваивание $u_1.u_2.\ ...\ .u_m\ \texttt{:=}\ u;$.

Метаприсваивание. Пусть $upd(w, u_1.u_2.\ ...\ .u_m, u)$ — функция, заменяющая $w(u_1)(u_2)\ ...\ (u_m)$ на u в w. Например, если $w \in N \to (N \to N)$, то $upd(w, 3.7, 10) = w'$ означает, что $w' \in N \to (N \to N)$, для любых $x \in N$ и $y \in N$ таких, что $x \neq 3$, или $y \neq 7$, верно $w'(x)(y) = w(x)(y)$, и $w'(3)(7) = 10$. В случае операций доступа к элементам последовательностей и кортежей $u(i)$ означает $u[i]$ и $u.i$ соответственно. Пусть $val(u, s)$ обозначает значение выражения или формулы u в состоянии s. Тогда метаприсваивание определяется правилом

$$(u_1.u_2.\ ...\ .u_m\ \texttt{:=}\ u;, s) \to_R (val(u, s), upd(s, u_1.u_2.\ ...\ .u_m, val(u, s))).$$

Итератор. Итераторы используются в правилах перехода как сокращение для повторения инструкций. Пусть $\eta[v \leftarrow u]$ обозначает замену всех вхождений v в инструкции η на u. Тогда итератор определяется правилами

(FOREACH v IN $< u_1, ..., u_m >$ DO η END; $\rightarrow_R \eta[v \leftarrow u_1]$ ... $\eta[v \leftarrow u_m], s$);
Если $< k_1, ..., k_m >$ — перестановка $< 1, ..., m >$,
то (FOREACH v IN $\{u_1, ..., u_m\}$ DO η END; $\rightarrow_R \eta[v \leftarrow u_{k_1}]$ ... $\eta[v \leftarrow u_{k_m}], s$).

Аннотации ASSUME, ASSERT и HAVOC. Эти аннотации определяются правилами

Если $val(f, s) =$ TRUE, то (ASSUME $f;, s$) $\rightarrow_R$ (OK, s);
Если $val(f, s) =$ FALSE, то (ASSUME $f;, s$) $\rightarrow_R$ (IGNORE, s);
Если $val(f, s) =$ TRUE, то (ASSERT $f;, s$) $\rightarrow_R$ (OK, s);
Если $val(f, s) =$ FALSE, то (ASSERT $f;, s$) $\rightarrow_R$ (FAIL, s);
Если $\gamma \in val(vt(v))$, то (HAVOC $v;, s$) $\rightarrow_R$ ($vv.v$:= γ, s);
(HAVOC $v_1, ..., v_n;, s$) $\rightarrow_R$ (FOREACH v IN $< v_1, ..., v_n >$ DO HAVOC v; END;, s);

Программа. Программа α определяется правилами

($\alpha;, s$) $\rightarrow_R$ (ASSUME ini; CONTROL LOOP;, s);
(CONTROL LOOP;, s) $\rightarrow_R$ (INPUT; p_1; ... p_n; OUTPUT; ASSERT inv; TICK; CONTROL LOOP;, s).

Инструкция CONTROL LOOP; определяет повторяющийся цикл управления. Инструкции INPUT; и OUTPUT; взаимодействуют с окружением, записывая значения во входные переменные и читая значения из выходных переменных соответственно. Инструкция p; выполняет процесс p. Инструкция TICK; увеличивает на единицу значения глобальных и локальных часов. Эти инструкции определяются правилами

(INPUT;, s) $\rightarrow_R$
(FOREACH v IN V_i DO HAVOC v; ih := $con(ih, < vv(v) >)$; END; ASSUME env;, s);
(OUTPUT;, s) $\rightarrow_R$ (FOREACH v IN V_o DO oh := $con(oh, < vv(v) >)$; END;, s);
($p;, s$) $\rightarrow$ (cp := p; $psv(ps(p))$, s);
(TICK;, s) $\rightarrow$ (gc := $gc + 1$; $lc.p_1$:= $lc(p_1) + 1$; ...; $lc.p_n$:= $lc(p_n) + 1;, s$).

Предикаты активности. Они определяются правилами:

((PROC p IN STATE ACTIVE), s) $\rightarrow_R$ ($s.ps(p) \neq$ STOP $\wedge$ $s.ps(p) \neq$ ERROR, s);
(PROC IN STATE ACTIVE, s) $\rightarrow_R$ ((PROC $s.cp$ IN STATE ACTIVE), s);
((PROC p IN STATE INACTIVE), s) $\rightarrow_R$ ($s.ps(p) =$ STOP $\vee$ $s.ps(p) =$ ERROR, s);
(PROC IN STATE INACTIVE, s) $\rightarrow_R$ ((PROC $s.cp$ IN STATE INACTIVE), s);
((PROC p IN STATE STOP), s) $\rightarrow_R$ ($s.ps(p) = stop, s$);
(PROC IN STATE STOP, s) $\rightarrow_R$ ((PROC $s.cp$ IN STATE STOP), s);
((PROC p IN STATE ERROR), s) $\rightarrow_R$ ($s.ps(p) =$ ERROR, s);
(PROC IN STATE ERROR, s) $\rightarrow_R$ ((PROC $s.cp$ IN STATE ERROR), s).

Операторы управления состояниями процессов. Они определяются правилами

(STOP PROC $p;, s$) $\rightarrow_R$ ($lc.p$:= 0; $ps.p$:= STOP;, s);
(STOP;, s) $\rightarrow_R$ (STOP PROC $s.cp;, s$);
(ERROR;, s) $\rightarrow_R$ ($lc.(s.cp)$:= 0; $ps.(s.cp)$:= ERROR;, s);
(START PROC $p;, s$) $\rightarrow_R$ ($lc.p$:= 0; $ps.p$:= $pso(p)[1];, s$);
(RESTART;, s) $\rightarrow_R$ (START PROC $s.cp;, s$);
(SET STATE $\theta;, s$) $\rightarrow_R$ ($lc.p$:= 0; $ps.p$:= $\theta;, s$);
Если $pso.p = con(..., < s.ps(s.cp), \theta >, ...)$,
то (SET NEXT, s) $\rightarrow_R$ ($lc.p$:= 0; $ps.p$:= $\theta;, s$).

Операторы таймаута. Они определяются правилами

(RESET TIMEOUT;, s) $\to_R$ ($lc.(s.cp)$:= 0;, s);
Если (e, s) $\to_R$ (γ, s'), и $s.cl(s.cp) \geq \gamma$, то (TIMEOUT e η, s) $\to_R$ (η, s');
Если (e, s) $\to_R$ (γ, s'), и $s.cl(s.cp) < \gamma$, то (TIMEOUT e η, s) $\to_R$ (OK, s').

Условный оператор. Он определяется правилами

Если (e, s) $\to_R$ (TRUE, s'), то (IF e η_1 ELSE η_2, s) $\to_R$ (η_1, s');
Если (e, s) $\to_R$ (FALSE, s'), то (IF e η_1 ELSE η_2, s) $\to_R$ (η_2, s');
Если (e, s) $\to_R$ (TRUE, s'), то (IF e η, s) $\to_R$ (η, s');
Если (e, s) $\to_R$ (FALSE, s'), то (IF e η, s) $\to_R$ (OK, s');

Блоки. Они определяются правилом

({η_1 ... η_m}, s) $\to_R$ (η_1 ... η_m, s).

Оператор присваивания. Он определяется правилами

Если (e, s) $\to_R$ (FAIL, s'), то (v = e, s) $\to_R$ (FAIL, s');
Если (e, s) $\to_R$ (γ, s'), и $\gamma \neq$ FAIL, то (v = e, s) $\to_R$ ($vv.v$:= γ;, s');

Последовательная композиция. Она определяется правилами

Если (η_1, s) $\to_R$ (γ, s'), и $\gamma \notin$ {FAIL, IGNORE},
то (η_1 η_2 ... η_m, s) $\to_R$ (η_2 ...η_m, s');
Если (η_1, s) $\to_R$ (FAIL, s'), то (η_1 η_2 ... η_m, s) $\to_R$ (FAIL, s').
Если (η_1, s) $\to_R$ (IGNORE, s'), то (η_1 η_2 ... η_m, s) $\to_R$ (IGNORE, s').

Вызов внешней функции. Он определяется правилами

Если $fpar(f) =< v_1, \ldots, v_m >$, FAIL $\notin \{\gamma_1, \ldots, \gamma_m\}$, и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_m, s_{m-1}) $\to_R$ (γ_m, s_m),
то (f(\(e_1, \ldots, e_m\);, s) $\to_R$
($vv.v_1$:= γ_1; ...; $vv.v_m$:= γ_m; ASSUME $fpre(f)$;
HAVOC f; ASSUME $fpost(f)$;, s_m);
Если $fpar(f) =< v_1, \ldots, v_m >$, FAIL $\notin \{\gamma_1, \ldots, \gamma_{k-1}\}$, γ_k = FAIL, и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_k, s_{k-1}) $\to_R$ (γ_k, s_k),
то (f(\(e_1, \ldots, e_m\);, s) $\to_R$ (FAIL, s_k).

Переменные в выражениях. Правило для переменной имеет вид

(v, s) $\to_R$ ($vv.v, s$).

Операции в выражениях. Определим над множеством O операций языка Reflex предикат $odef \in O \times \Gamma^+ \to bool$, специфицирующий область определения этих операций, и функцию $oval \in O \times \Gamma^+ \to \Gamma$, специфицирующую значение для аргументов из области определения. Тогда правила для префиксной формы o(\(e_1, \ldots, e_m\); операции o местности m имеют вид:

Если FAIL $\notin \{\gamma_1, \ldots, \gamma_m\}$, $odef(o, \gamma_1, \ldots, \gamma_m)$ = TRUE и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_m, s_{m-1}) $\to_R$ (γ_m, s_m),
то (o(\(e_1, \ldots, e_m\);, s) $\to_R$ ($oval(o, \gamma_1, \ldots, \gamma_m), s_m$);
Если FAIL $\notin \{\gamma_1, \ldots, \gamma_{k-1}\}$, γ_k = FAIL, и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_k, s_{k-1}) $\to_R$ (γ_k, s_k),
то (o(\(e_1, \ldots, e_m\);, s) $\to_R$ (FAIL, s_k).
Если FAIL $\notin \{\gamma_1, \ldots, \gamma_m\}$, $odef(o, \gamma_1, \ldots, \gamma_m)$ = FALSE и
(e_1, s) $\to_R$ (γ_1, s_1), (e_2, s_1) $\to_R$ (γ_2, s_2), ..., (e_m, s_{m-1}) $\to_R$ (γ_m, s_m),
то (o(\(e_1, \ldots, e_m\);, s) $\to_R$ (FAIL, s_m);

Заключение

Reflex имеет очень простой синтаксис. Он не имеет составных типов данных и указателей. Он не имеет операторов итерации и операторов перехода. Он имеет ограниченное количество операций в выражениях и ограниченное число простых типов данных.

Сложность семантики аннотированных Reflex программ обусловлена необходимостью учитывать бесконечность цикла выполнения программы, логику управления переходами процессов из состояния в состояние, взаимодействие процессов между собой и с окружением, ограничения на окружение и запуск программы, временные ограничения на исполнение процессов в определенных состояниях, вызовы внешних функций и инварианты цикла управления.

Предложенная в данной работе операционная семантика аннотированных Reflex программ справляется с этими трудностями, используя понятия глобальных и локальных часов, разделяя переменные на внутренние, входные и выходные, учитывая полные истории изменения значений разделяемых переменных и моделируя ограничения на запуск программы и окружения и инварианты цикла управления через аннотации `ASSUME`, `ASSERT` и `HAVOC`.

Мы планируем использовать эту семантику при доказательстве корректности трансформационного подхода к верификации Reflex программ [6], в частности, для доказательства корректности трансформационной семантики Reflex программ [7].

Список литературы / References

[1] Iec, IEC, “61131-3: Programmable Controllers–Part 3: Programming Languages”, *International Standard, Second Edition, International Electrotechnical Commission, Geneva*, **1** (2003).

[2] Basile F., Chiacchio P., Gerbasio D., “On the Implementation of Industrial Automation Systems Based on PLC”, *IEEE Transactions on Automation Science and Engineering*, **10**:4 (2012), 990–1003.

[3] Travis J., Kring J., *LabVIEW for Everyone: Graphical Programming Made Easy and Fun, Third Edition*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[4] Zyubin V., “Using Process-Oriented Programming in LabVIEW”, *Proceedings of the Second IASTED International Multi-Conference on ”Automation, Control, and Information technology“: Control, Diagnostics, and Automation. Novosibirsk*, 2010, 35–41.

[5] Buxton J. N., Randell B., *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*, NATO Science Committee, Brussels, Scientific Affairs Division, NATO, Rome, Italy, 1970.

[6] Anureev I. S., Garanina N. O., Liakh T. V., Rozov A. S., Zyubin V. E., Gorlatch S., “Two-Step Deductive Verification of Control Software Using Reflex”, *Preliminary Proceedings of A. P. Ershov Informatics Conference (PSI-19). Novosibirsk, Russia, Akademgorodok, Russia, July 2–5*, 2019, 17–30.

[7] Anureev I. S., Garanina N. O., Liakh T. V., Rozov A. S., Schulte H., Zyubin V. E., “Towards Safe Cyber-Physical Systems: the Reflex Language and its Transformational Semantics”, *14th International Siberian Conference on Control and Communications (SIBCON). Tomsk State University of Control Systems and Radioelectronics, Tomsk, April 18-20*, 2019, 1–6.

[8] Zyubin V. E., Liakh T. V., Rozov A. S., “Reflex Language: a Practical Notation for Cyber-Physical Systems”, *System Informatics*, **12** (2018), 85–104.

[9] Norrish M., “C Formalised in HOL”, *Ph.D. thesis. University of Cambridge, Technical Report, UCAM-CL-TR-453*, 1998.

[10] Gurevich Y., Huggins J., “The Semantics of the C Programming Language”, *International Workshop on Computer Science Logic. Lecture Notes in Computer Science*, **702** (1992), 274–308.

[11] Blazy S., Leroy X., “Mechanized Semantics for the Clight Subset of the C Language”, *Journal of Automated Reasoning*, **43**:3 (2009), 263–288.

[12] Nepomniaschy V. A., Anureev I. S., Mikhailov I. N., Promsky A. V., “Towards Verification of C Programs. C-light Language and its Formal Semantics”, *Programming and Computer Science*, **28**:6 (2002), 314–323.

[13] Ellison C., Rosu G., “An Executable Formal Semantics of C with Applications”, *Proc. of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, **47**:1 (2012), 533–544.

Anureev I. S., "Operational Semantics of Annotated Reflex Programs", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 475–487.

DOI: 10.18255/1818-1015-2019-4-475-487

Abstract. Reflex is a process-oriented language that provides a design of easy-to-maintain control software for programmable logic controllers. The language has been successfully used in a several reliability critical control systems, e. g. control software for a silicon single crystal growth furnace and electronic equipment control system. Currently, the main goal of the Reflex language project is to develop formal verification methods for Reflex programs in order to guarantee increased reliability of the software created on its basis. The paper presents the formal operational semantics of Reflex programs extended by annotations describing the formal specification of software requirements as a necessary basis for the application of such methods. A brief overview of the Reflex language is given and a simple example of its use – a control program for a hand dryer – is provided. The concepts of environment and variables shared with the environment are defined that allows to disengage from specific input/output ports. Types of annotations that specify restrictions on the values of the variables at program launch, restrictions on the environment (in particular, on the control object), invariants of the control cycle, pre- and postconditions of external functions used in Reflex programs are defined. Annotated Reflex also uses standard annotations `assume`, `assert` and `havoc`. The operational semantics of the annotated Reflex programs uses the global clock as well as the local clocks of separate processes, the time of which is measured in the number of iterations of the control cycle, to simulate time constraints on the execution of processes at certain states. It stores a complete history of changes of the values of shared variables for a more precise description of the time properties of the program and its environment. Semantics takes into account the infinity of the program execution cycle, the logic of process transition management from state to state and the interaction of processes with each other and with the environment. Extending the formal operational semantics of the Reflex language to annotations simplifies the proof of the correctness of the transformation approach to deductive verification of Reflex programs developed by the authors, transforming an annotated Reflex program to an annotated program in a very limited subset of the C language, by reducing a complex proof of preserving the truth of program requirements during the transformation to a simpler proof of equivalence of the original and the resulting annotated programs with respect to their operational semantics.

Keywords: operational semantics, Reflex language, control system, control software, programmable logic controller, annotation, annotated program

On the authors:
Igor S. Anureev, orcid.org/0000-0001-9574-128X, PhD,
A. P. Ershov Institute of Informatics Systems SB RAS, Institute of Automation and Electrometry SB RAS,
6 Acad. Lavrentjev pr., Novosibirsk 630090, Russia, e-mail: anureev@iis.nsk.su

Acknowledgments:
This work was funded by the RFBR according to the research № 17-07-01600 and Funding State budget of the Russian Federation (IA&E project № AAAA-A17-11706061006-6).