Модел. и анализ информ. систем. Т. **19**, № **6** (2012) 34–44

УДК 519.681

Дедуктивная верификация телекоммуникационных систем, представленных на языке Си

Ануреев И.С.[1]

Институт систем информатики имени А.П. Ершова СО РАН

e-mail: anureev@iis.nsk.su

получена 22 июля 2012

Ключевые слова: верификация, спецификация, операционная семантика, аксиоматическая семантика, трансформационная семантика, телекоммуникационные системы, телекоммуникационные протоколы

Предложен дедуктивный подход к верификации телекоммуникационных систем, представленных на языке С. Подход основан на расширении языка С декларативными операторами и сведении верификации параллельных взаимодействующих компонент телекоммуникационных систем к раздельной верификации компонент, представленных на расширенном языке. Рассмотрен пример верификации протокола передачи данных.

1. Введение

В настоящее время верификация телекоммуникационных систем, представленных на языке С, базируется, в основном, на методе проверки моделей. Этот подход позволяет проводить верификацию в автоматическом режиме, но имеет известную проблему взрыва числа состояний, что ограничивает его применение к большим системам.

Наоборот, дедуктивная верификация, основанная на применении логических исчислений, как правило, проводится в полуавтоматическом режиме, что связано с поиском промежуточных утверждений (в частности, инвариантов контрольных точек программы), используемых для построения дерева логического вывода. В то же время использование средств логического доказательства (в частности, индукции) позволяет применять этот подход к большим, потенциально бесконечным системам.

Перечислим наиболее известные проекты в этой области.

В проекте [16] представлен метод дедуктивного поиска ошибок в программах на языке MISRA C, который является индустриальным стандартом для написания безопасных программ, ориентированным, в основном, на встраиваемые системы.

[1]Работа частично поддержана грантом РФФИ 11-07-90412-Ukr_f_a.

Входная С-программа моделируется в виде типизированной системы переходов на языке спецификаций автоматического доказателя теорем PVS.

В рамках проекта Why [17] разработан инструментарий Frama-C [13], предназначенный для анализа Си-программ, комбинирующего статический анализ для полного языка Си и дедуктивную верификацию для ограниченного подмножества.

Примером узкоспециализированного проекта по дедуктивной верификации является проект VERISOFT [9], ориентированный, в основном, на верификацию компонент конкретной операционной системы.

Еще одним проектом, использующим дедуктивную верификацию для проверки параллельных программ, написанных на языке C, является проект VCC [12], выполняемый в Microsoft. Основной целью этого проекта являлась верификация гипервизора Microsoft Hyper-V [14].

Особенностью нашего подхода к дедуктивной верификации телекоммуникационных систем, представленных на языке C, по сравнению с подходами, на которых базируются вышеприведенные проекты, является использование методологии, описывающей пошаговую процедуру сведения параллельных взаимодействующих компонент, составляющих телекоммуникационную систему, к изолированным последовательным компонентам. Это позволяет комбинировать подход с дедуктивными средствами верификации последовательных Си-программ.

2. Язык C-dest

Предлагаемый дедуктивный подход к верификации телекоммуникационных систем, представленных на языке C, использует расширение C-dest этого языка, в которое транслируются С-функции, описывающие процессы и компоненты телекоммуникационной системы. Язык C-dest (The C language with declarative statements) расширяет язык C за счет декларативных операторов и средств явной параллелизации.

Декларативные операторы определяют, какие объекты программы и в каких пределах меняются, но не определяют, как это происходит. Они включают формулы некоторого логического языка, который будем называть языком аннотаций. В дальнейшем в качестве такого языка используется язык логики предикатов первого порядка. Введение декларативных операторов позволяет описать для каждого процесса обобщенное действие всех процессов, выполняющихся одновременно с ним, и, за счет использования интерливинговой семантики, изолировать параллельные процессы друг от друга.

Перечислим операторы, добавленные в язык C-dest. Пусть `P` — формула языка аннотаций, `x = (`x_1`, ..., `x_n`)` — список программных переменных.

Оператор `assume P;` эквивалентен пустому оператору, если `P` истинна. Если `P` ложна, выполнение функции, содержащей этот оператор, приостанавливается до тех пор, пока `P` не станет истинным.

Операторы `assert P;` и `cut P;`, имеющие одинаковую операционную семантику, эквивалентны пустому оператору, если `P` истинна. Если `P` ложна, программа завершается с ошибкой. Аксиоматическая семантика этих операторов различается. Считается, что `P` является полным инвариантом контрольной точки, предшествую-

щей оператору `cut P;` и частичным инвариантом контрольной точки, предшествующей оператору `assert P;`. Полный инвариант позволяет разрезать программу в соответствующей контрольной точке при дедуктивном выводе.

Оператор `havoc x;` изменяет значения переменных из `x` произвольным образом. Оператор `modify x to P;` изменяет значения переменных из `x` таким образом, что при новых значениях переменных формула `P` становится истинной. Формула `P` может содержать переменные вида $\mathtt{x}_i\mathtt{?}$, которые ссылаются на начальные значения переменных из `x`, т. е. значения, которые эти переменные имели до выполнения оператора `modify`.

C-dest также включает следующие недекларативные операторы. Оператор параллельного выполнения $\mathtt{A}_1$ `||` ... `||` $\mathtt{A}_n$ параллельно исполняет программные фрагменты $\mathtt{A}_1$, ..., $\mathtt{A}_n$. Оператор `atomic {A}` накладывает ограничение на доступ к программным переменным. Разделяемые переменные, которые использует программный фрагмент `A`, недоступны другим параллельным процессам телекоммуникационной системы.

Язык C-dest включает нетипизированные переменные. Оператор `var x;` определяет нетипизированные переменные из списка `x` и присваивает им произвольные начальные значения. Оператор `A := B;` присваивает нетипизированной переменной `A` значение `B`.

3. Аксиоматическая семантика языка C-dest

Правила аксиоматической семантики языка C-dest используются при раздельной дедуктивной верификации компонент телекоммуникационной системы, переписанных на языке C-dest. Определим их.

Конструкции языка C, входящие в C-dest, ограничены подмножеством C-light [5, 15] языка C. Дедуктивная верификация C-light-программ основана на их трансляции в язык C-kernel [6], который имеет аксиоматическую семантику. Язык C-dest использует для C-конструкций те же самые правила аксиоматической семантики, что и C-kernel, но меняет форму их представления. Тройки Хоара `{P}A{Q}` заменяются на последовательности операторов `assume P; A assert Q;`, в которых пред- и постусловия моделируются декларативными операторами `assume` и `assert`, соответственно. Правило $\frac{P_1 \ldots P_n}{Q}$ с посылками $\mathtt{P}_1$, ..., $\mathtt{P}_n$ и следствием `Q` представляется в виде правила переписывания `Q` $\rightarrow$ $\mathtt{P}_1$`|` ... `|`$\mathtt{P}_n$.

Рассмотрим специфические правила языка C-dest. Пусть `E` — оператор языка C-dest.

Рассмотрим сначала правила для последовательности операторов, содержащей один элемент. Если `E` — простой оператор и `E` не является оператором `assert` или `cut`, то `E` $\rightarrow$ `true`. В случае `assert` или `cut`, `assert P;` $\rightarrow$ `P` и `cut P;` $\rightarrow$ `P`. Составной оператор сводится к вложенным операторам. Так, правило для условного оператора имеет вид: `if P then B else C` $\rightarrow$ `assume P; B|assume (not P); C`. Правило для оператора параллельного выполнения имеет вид: `{B||C}` $\rightarrow$ `B|C`. Правила для других составных операторов определяются аналогичным образом.

Рассмотрим теперь правила для последовательности из более чем одного опе-

ратора. Если `E` — простой оператор и `E` не является оператором `assert` или `cut`, то `A E` → `A`. Если последовательность заканчивается составным оператором, то он, как и в случае выше, сводится к вложенным операторам. Правило для условного оператора имеет вид: `A if P then B else C` → `A assume P; B|A assume (not P); C`. Правило для оператора параллельного выполнения имеет вид: `{A B||C}` → `A B|A C`. Правила для других составных операторов определяются аналогичным образом.

Последовательность, заканчивающаяся оператором `cut`, сводится к последовательности, заканчивающейся оператором `assert`: `A cut P;` → `A assert P;`.

Пусть `A(x ← e)` обозначает одновременную замену $\mathtt{x}_1 \leftarrow \mathtt{e}_1, \ldots, \mathtt{x}_n \leftarrow \mathtt{e}_n$ переменных $\mathtt{x}_i$ в `A` на $\mathtt{e}_i$, $\mathtt{xx}_i$ — новые переменные. Рассмотрим правила для последовательностей, заканчивающихся оператором `assert`.

```
A assume P; assert Q; → A assert P => Q;
A assert P; assert Q; → A assert P;|A assert Q;
A cut P; assert Q; → A assert P;|P => Q
A havoc x; assert Q; → A assert Q(x ← xx);
A modify x to P; assert Q; → A assert P(x ← xx, x? ← x) => Q(x ← xx);
A atomic {B} assert Q; → A B assert Q;
A {B||C} assert Q; → A B assert Q;|A C assert Q;
A var x; assert Q; → A assert Q(x ← xx);
A x := e; assert Q; → A assert Q(x ← e);
```

В языке C-dest допускаются циклы без инвариантов. Например, процессы телекоммуникационной системы могут моделироваться бесконечными циклами, которые не имеют инвариантов. Правило для цикла без инварианта имеет вид: `A while(P) B assert Q;` → `A assume P; B|A assume not(P); assert Q;|B assume P; B|B assume not(P) assert Q;`. Правило для частного случая бесконечного цикла без инварианта имеет упрощенную форму: `A while (true) B assert Q;` → `A B|BB`.

4. Неограниченный симплексный протокол передачи данных

В этом разделе мы опишем пример телекоммуникационной системы, который будет использоваться в качестве сквозного примера при иллюстрации шагов нашего подхода в разделе 5. В качестве примера мы рассмотрим один из самых простых протоколов передачи данных из [7] — неограниченный симплексный протокол. Данные передаются только в одном направлении. Сетевой уровень на передающей и приемной сторонах находится в состоянии постоянной готовности. Временем обработки можно пренебречь. Размер буфера неограничен, а канал связи между уровнями передачи данных никогда не теряет и не искажает кадры.

Протокол состоит из двух функций, `sender` (отправитель) и `receiver` (получатель). Функция sender работает па уровне передачи данных посылающей машины, а функция receiver — на уровне передачи данных принимающей машины.

Функция `sender` представляет собой бесконечный цикл while. Тело цикла состоит из трех действий: получения пакета с сетевого уровня, формирования исходящего

пакета с помощью переменной `s` и отсылки пакета адресату. Из служебных полей кадра данный протокол использует только поле `info`, содержащее передаваемый пакет.

Функция `receiver` принимающей стороны также представляет собой бесконечный цикл. Вначале она ожидает, пока что-нибудь произойдет, причем единственно возможным событием в данном протоколе может быть получение неповрежденного пакета. Когда пакет появляется, процедура `wait_for_event` завершается, при этом переменной `event` присваивается значение `frame_arrival`. Обращение к процедуре `from_physical_layer` удаляет вновь прибывший кадр из аппаратного буфера и помещает его в переменную `r`. Наконец, порция данных передается сетевому уровню, а уровень передачи данных отправляется ждать следующий кадр.

Протокол использует библиотечные типы и функции из `protocol.h` , чтобы специфицировать взаимодействие (коммуникацию) процессов:

```
#include "protocol.h"

void sender(void) {
 frame s; // буфер для исходящего кадра
 packet buffer; // буфер для исходящего пакета
 while(true) {
  from_network_layer(&buffer); // получить у сетевого уровня пакет для передачи
  s.info = buffer; // скопировать его в кадр s для передачи
  to_physical_layer(&s); // послать кадр s
 }}

void receiver(void) {
 frame r;
 while(true) {
  wait_for_event(); // ждать ожидания события frame_arrival
  from_physical_layer(&r); // получить прибывший кадр
  to_network_layer(&r.info); // передать данные сетевому уровню
 }}
```

5. Описание дедуктивного подхода

Предлагаемый дедуктивный подход сводит верификацию параллельных взаимодействующих компонент телекоммуникационной системы, представленных на языке C, к дедуктивной верификации последовательных изолированных компонент, представленных на языке C-dest, и представляет собой последовательность шагов, которые описаны ниже. Эти шаги иллюстрируются на примере проверки некоторого свойства безопасности `Prop` для протокола, описанного в предыдущем разделе. Свойство `Prop` заключается в том, что последовательность пакетов, передаваемых процессом `receiver` сетевому уровню, является префиксом последовательности пакетов, получаемых процессом `sender` с сетевого уровня.

Первые два шага состоят в описании (формализации) на языке C-dest данных и процессов, которые не представлены в явном виде в исходной спецификации телекоммуникационной системы на языке C, но требуются в формальной модели, используемой в аксиоматической семантике.

1. Определение неявных разделяемых переменных. Функциональность, обеспечивающая взаимодействие параллельных процессов телекоммуникационной системы, моделируется обращением процессов к разделяемым переменным. С этой целью выделяются и определяются разделяемые переменные.

Пример. Определяются 4 разделяемых переменных. Переменная `IN` хранит последовательность пакетов, переданных уровню данных с сетевого уровня. Переменная `OUT` хранит последовательность пакетов, переданных сетевому уровню с уровня данных. Переменная `CH1` хранит последовательность кадров, которые находятся в канале, но еще не пришли к получателю. Переменная `CH2` хранит последовательность кадров, пришедших к получателю, но еще не считанных им.

Пусть функции `del-last(T)`, `last(T)`, `T + TT`, `[x]` и `rev(T)` языка аннотаций определяют кортеж `T` без последнего элемента, последний элемент кортежа `T`, конкатенацию `T` и `TT`, кортеж из одного элемента `x` и обращение кортежа `T`, соответственно. Пусть `CH = rev(info(CH1 + CH2))`. Функция `info(x)` возвращает кортеж значений полей `info` фреймов из кортежа `x`. Свойство `Prop` определяется в этих терминах следующим образом: `∃X(IN = OUT + CH + X)`.

2. Определение неявных процессов. На этом шаге неявные процессы определяются как функции на языке C-dest.

Пример. Определим два неявных процесса — окружение `env` и главный процесс `main`, инициирующий запуск телекоммуникационного протокола:

```
void env(void) {
 while (true) {
  modify CH1, CH2 to CH1 = del-last(CH1?) & CH2 = [last(CH1?)] + CH2?;}}

void main(void) {
 var IN, CH1, CH2, OUT, sender, receiver, env;
 assume IN = OUT + CH & sender = (0, 0) & receiver = (0, 0);
 {sender(); || receiver(); || env();}}
```

Событийные переменные `sender` и `receiver` рассматриваются ниже в п. 6.

3. Определение правил замены вызовов функций. На этом шаге определяются правила переписывания, которые заменяют вызовы C-функций на декларативные операторы. Пусть `p` — процесс, `f` — функция, специфицирующая `p`. Вид правила зависит от типа процесса, специфицированного функцией. Завершающиеся процессы могут завершаться при наступлении некоторого внутреннего (зависит только от входных данных процесса) или внешнего (зависит от значения разделяемых переменных) условия. Для завершающихся процессов правило имеет вид: `f(y)` → `atomic {assert P; modify x to Q;}`, где `y` — список фактических параметров функции `f`. Формула `P` (предусловие функции `f`) определяет условия, которые накладываются на входные данные функции `f`. Формула `Q` (постусловие функции `f`) описывает ограничения на выходные данные функции `f`. Список `x` содержит разделяемые переменные, которые могут изменяться функцией `f`. В случае `P = true`, оператор `assert P;` может опускаться.

Правило для завершающихся процессов, срабатывающих при наступлении некоторого условия, имеет вид: `f(y)` → `atomic {assume P; modify x to Q;}`. Форму-

ла P определяет внешнее условие (наступление некоторого события), при котором функция f срабатывает.

Правило для незавершающихся процессов имеет вид: f(y) → assert P;. Предусловие P является условием запуска незавершающегося процесса.

Правило для незавершающихся процессов, срабатывающих при наступлении некоторого условия, имеет вид: f(y) → atomic {assume P;}.

Пример. Функция sender, принимающая пакет с сетевого уровня и посылающая его получателю, моделирует незавершающийся процесс и имеет правило sender() → assert IN = OUT + CH & sender = (0, 0);.

Функция wait_for_event, ожидающая наступления события "прибыл новый кадр", моделирует процесс, срабатывающий при наступлении условия, и имеет правило wait_for_event(y) → assume not(CH2 = [ ]); modify *y to *y = frame_arrival;.

Функция from_network_layer, получающая пакет у сетевого уровня, моделирует завершающийся процесс и имеет правило: from_network_layer(y) → modify IN, *y to IN = IN? + [*y];. Правила замены для остальных функций определяются аналогичным образом.

4. Приведение тел функций в соответствии с правилами замены вызовов функций. Замена вызовов функций на декларативные операторы является аппроксимирующей, а не функционально эквивалентной заменой. Поэтому чтобы обеспечить ее корректность, пред- и постусловия, которые входят в эти операторы, должны ограничивать тела соответствующих функций. Пусть B — тело функции. Для завершающихся процессов и процессов, срабатывающих при наступлении некоторого условия, тела переписываются в соответствии с правилом: B → assume P; B assert Q & y = y?;, где y — список разделяемых переменных, значения которых не меняются при выполнении функции f. Для незавершающихся процессов правило переписывания тела функции имеет вид: B → assume P; B.

Пример. Рассмотрим пример тела для функции sender.

```
   void sender(void) {
4   assume IN = OUT + rev(info(conc(CH1, CH2))) and sender = (0, 0);
    frame s; packet buffer;
    while (true) {
     atomic {
9     modify IN, OUT, CH1, CH2, receiver to inv;
5,7   modify IN, buffer to IN = IN? + [buffer];
6     sender := (1, buffer);
9     cut inv and sender = (1, buffer);}
    s.info = buffer;
    atomic {
9    modify IN, OUT, CH1, CH2, receiver to inv;
5,7  modify CH1 to CH1 = [s] + CH1?;
6    sender := (0, 0);
9    cut inv and sender = (0, 0);} }}
```

В этом примере цифры слева определяют, какие операторы добавляются в тело функции на соответствующем шаге. Так, предусловие, добавляемое на этом шаге, помечено цифрой 4.

5. Замена вызовов функций. На этом шаге тела функций переписываются в соответствии с правилами замены вызовов функций.

6. Определение и означивание событийных переменных. Параллельные процессы взаимодействуют через разделяемые переменные. Чтобы описать состояния этих процессов, характеризующие обращения к разделяемым переменным, для каждого такого процесса выделяется набор событий и связанных с ними значений. Эти события нумеруются натуральным отрезком `[1, n]`. Для наблюдения событий и связанных с ними значений для каждого процесса `f` определяется глобальная событийная переменная `f`. Значением переменной `f` является пара `(u, v)`, где `u` — номер последнего события, которое произошло, `v` — связанное с ним значение. Доступ к элементам `u` и `v` этой пары обозначим через `f.1` и `f.2` соответственно. Чтобы обеспечить актуальность значений событийных переменных, в тело каждой функции `f` добавляются присваивания вида `f := (u, v)`.

7. Выделение точек обращения к разделяемым переменным. В теле функции `f` выделяются программные фрагменты, которые выполняют обращение к разделяемым переменным (чтение или запись). Такие фрагменты называются точками обращения к разделяемым переменным.

8. Определение инварианта точек обращения к разделяемым переменным. Инвариант `inv` описывает свойство разделяемых переменных, которое остается истинным при любом обращении к ним.

Если `inv` — инвариант, то доказательство свойства `Prop` сводится к проверке истинности формулы `inv => Prop`.

Будем говорить, что `inv` разложим по базису событийных переменных, если он представим в виде конъюнкции импликаций $\mathtt{f}_1\mathtt{.1 = i}_1\ \mathtt{\&\ ...\ \&\ f}_m\mathtt{.1 = i}_m$ `=>` $\mathtt{inv}_{i_1 \ldots i_m}$, где $1 \le \mathtt{i}_j \le \mathtt{n}_j$, $\mathtt{n}_j$ — число событий в процессе $\mathtt{f}_j$, $\mathtt{inv}_{i_1 \ldots i_m}$ — формулы языка аннотаций.

Пример. Инвариант `inv` имеет вид:

```
(sender.1 = 0 & receiver.1 = 0 => IN = OUT + CH) &
(sender.1 = 1 & receiver.1 = 0 => IN = OUT + CH + [sender.2]) &
(sender.1 = 0 & receiver.1 = 1 => IN = OUT + [receiver.2] + CH) &
(sender.1 = 1 & receiver.1 = 1 => IN = OUT + [receiver.2] + CH + [sender.2])
```

Нетрудно доказать, что формула `inv => Prop` истинна.

9. Инвариантное покрытие точек обращения к разделяемым переменным. Замена каждой точки обращения `A` в телах функций на `atomic {modify x to inv; A cut inv &` $\mathtt{inv}_l$`;}` называется инвариантным покрытием точек обращения к разделяемым переменным. Здесь `x` — список всех разделяемых переменных, за исключением событийной переменной `f`, значение которой может меняться только процессом `f`; формула $\mathtt{inv}_l$, называемая локальным инвариантом, обеспечивает полноту инварианта `inv &` $\mathtt{inv}_l$. Инвариантное покрытие позволяет перейти от верификации параллельных взаимодействующих процессов к верификации изолированных последовательных процессов.

10. Генерация условий корректности. Условия корректности генерируются в соответствии с описанными выше правилами аксиоматической семантики C-dest.

6. Заключение

В статье предложен дедуктивный подход к верификации телекоммуникационных систем, представленных на языке C. В рамках подхода впервые рассмотрены такие концепции, как расстановка событийных переменных, описывающих наблюдаемые состояния процессов телекоммуникационной системы, разложение инварианта разделяемых переменных по базису событийных переменных, инвариантное покрытие точек обращения к разделяемым переменным и замена вызовов функций декларативными операторами. Эти концепции позволяют свести верификацию параллельных взаимодействующих процессов к дедуктивной верификации изолированных последовательных процессов. Представлена аксиоматическая семантика языка C, расширенного декларативными операторами и операторами явного параллелизма.

Применение подхода проиллюстрировано на примере доказательства свойства безопасности неограниченного симплексного протокола передачи данных из [7]. Однако та же самая схема доказательства может быть легко применена к более сложным протоколам передачи данных из [7] (например, протоколу скользящего окна). Вообще, подход непосредственно применим для доказательства любого свойства безопасности, которое является инвариантом разделяемых переменных. Интересной задачей является расширение класса свойств безопасности, к которым применим подход, за счет, например, сохранения истории обращений к разделяемым переменным в событийных переменных. Применимость этого подхода к другим классам свойств реактивных систем также интересная открытая проблема.

Также следует отметить, что фактически язык Си, на котором специфицируются телекоммуникационные протоколы, является параметром подхода. Таким образом, интересной проблемой является применение подхода к спецификациям телекоммуникационных систем, представленных на других языках программирования.

В будущем мы планируем интегрировать этот подход в мультиязыковую систему анализа и верификации программ Спектр [4, 11] и верифицировать с его помощью новые телекоммуникационные протоколы и системы. В частности, мы планируем применить его к динамическим процессам. В этом случае экземпляр процесса будет определяться именем функции, специфицирующей процесс, и уникальным идентификатором для этого экземпляра. Для упрощения условий корректности планируется использовать методы из [1, 2, 3, 8, 10]. Еще одной важной задачей является формальное обоснование корректности подхода.

Список литературы

1. Ануреев И.С. Метод элиминации структур данных, основанный на системах переписывания формул // Программирование. 1999. №4. С. 5–15.

2. Ануреев И.С., Марьясов И.В., Непомнящий В.А. Верификация C-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем. 2010. Т. 17, №3. С. 5–28.

3. Атучин М.М., Ануреев И.С. Атрибутные аннотации и их применение в дедуктивной верификации С-программ // Моделирование и анализ информационных систем. 2011. Т. 18, №4. С. 21–33.

4. Непомнящий В.А., Ануреев И.С., Атучин М.М., Марьясов И.В., Петров А.А., Промский А.В. Верификация С-программ в мультиязыковой системе СПЕКТР // Моделирование и анализ информационных систем. 2010. Т. 17, №4. С. 88–100.

5. Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В. На пути к верификации С программ. Язык C-light и его формальная семантика // Программирование. 2002. №6. С. 1–13.

6. Непомнящий В.А., Ануреев И.С., Промский А.В. На пути к верификации С программ. Аксиоматическая семантика языка C-kernel // Программирование. 2003. №6. С. 5–15.

7. Таненбаум Э. Компьютерные сети. 4-е издание. 2003. 992 с.

8. Шилов Н.В., Ануреев И.С., Бодин Е.В. О генерации условий корректности для императивных программ // Программирование. 2008. №6. С. 1–20.

9. Alkassar E., Hillebrand M.A., Paul W., Petrova E. Automated Verification of a Small Hypervisor // Proc. of VSTTE 2010. Lect. Notes Comput. Sci. 2010. Vol. 6217. P. 40–54.

10. Anureev I.S. A three-stage method of C program verification // Joint NCC&IIS Bulletin, Series Computer Science. 2008. Vol. 28. P. 1–29.

11. Anureev I.S. Integrated approach to analysis and verification of imperative programs // Joint NCC&IIS Bulletin, Series Computer Science. 2011. Vol. 32. P. 1–18.

12. Cohen E., Dahlweid M., Hillebrand M., at el. VCC: A Practical System for Verifying Concurrent C // Proc. of TPHOLs 2009. Lect. Notes Comput. Sci. 2009. Vol. 5674. P. 23–42.

13. Frama-C. http://frama-c.com/

14. Leinenbach D., Santen T. Verifying the Microsoft Hyper-V Hypervisor with VCC // Proc. of FM 2009. Lect. Notes Comput. Sci. 2009. Vol. 5850. P. 806–809.

15. Nepomniaschy V.A., Anureev I.S., Promsky A.V. Verification-oriented language C-light and its structural operational semantics // PSI-2003. Proc. of Conf. Lect. Notes Comput. Sci. 2003. Vol. 2890. P. 1–5.

16. Sharma B., Dhodapkar S.D., Ramesh S. Assertion Checking Environment(ACE) for Formal Verification of C Programs // Proc. of SAFECOMP 2002. Lect. Notes Comput. Sci. 2002. Vol. 2434. P. 284–295.

17. Why3: Where Programs Meet Provers. http://why3.lri.fr/

Deductive Verification of Telecommunication Systems Written in C

Anureev I.S.

Keywords: verification, specification, operational semantics, axiomatic semantics, transformational semantics, telecommunication systems, telecommunication protocols

A deductive approach to verification of telecommunication systems written in C is proposed. The approach is based on the extension of C by declarative statements and on reduction of verification of parallel communicating components of these systems to separate verification of components written in this extension. An example of verification of a data link protocol is considered.

Сведения об авторе:
Ануреев Игорь Сергеевич,
Институт систем информатики имени А.П. Ершова СО РАН,
старший научный сотрудник.