Модел. и анализ информ. систем. Т. **17**, № **4** (2010) 101–110

УДК 004.415.52

Верификация и синтез программ сложения на базе правил корректности операторов

Шелехов В.И.

Институт систем информатики, Новосибирск

e-mail: vshel@iis.nsk.su

получена 18 октября 2010

Ключевые слова: предикатное программирование, дедуктивная верификация, программный синтез, тотальная корректность программы

Дедуктивная верификация и синтез программ двоичного сложения проводятся на базе системы правил доказательства корректности для операторов языка предикатного программирования P. В работе представлены ключевые фрагменты процесса верификации и синтеза программ сумматоров. Условия корректности программ транслировались на язык спецификаций системы автоматического доказательства PVS. Доказательство на PVS оказалось на порядок более трудоемким процессом по сравнению с обычным программированием. Однако в режиме синтеза построение теорий и доказательство на PVS проще и быстрее по сравнению с верификацией.

1. Введение

В настоящей работе на примере программ двоичного сложения иллюстрируются методы программного синтеза и дедуктивной верификации программ на языке предикатного программирования P (Predicate programming language) [1]. Язык P находится на границе между функциональными и логическими языками. Язык P, его логическая и операционная семантика построены расширением цепочки языков $\mathsf{CCP} = \mathsf{P}_0 \subset \mathsf{P}_1 \subset \mathsf{P}_2 \subset \mathsf{P}_3 \subset \mathsf{P}_4 = \mathsf{P}$, где CCP – типизированное исчисление вычислимых предикатов [2]. Базис исчисления содержит: оператор суперпозиции $\mathsf{B(x:z)};\mathsf{C(z:y)}$, параллельный оператор $\mathsf{B(x:y)} \,||\, \mathsf{C(x:z)}$ и условный оператор **if** (b) B(x: y) **else** C(x: y); где B и C – имена предикатов; x, y и z – различающиеся наборы переменных, причем набор x может быть пустым, b – переменная логического типа. Логическая семантика LS для операторов определяется следующим образом:

$$\mathsf{LS}(\mathsf{B(x:z)};\mathsf{C(z:y)}) \;\cong\; \exists \mathsf{z}.\,(\mathsf{B(x,\,z)} \,\&\, \mathsf{C(z,\,y)})$$
$$\mathsf{LS}(\mathsf{B(x:y)} \,||\, \mathsf{C(x:z)}) \;\cong\; \mathsf{B(x,\,y)} \,\&\, \mathsf{C(x,\,z)}$$
$$\mathsf{LS}(\textbf{if}\ (\mathsf{b})\ \mathsf{B(x:y)}\ \textsf{else}\ \mathsf{C(x:y)}) \;\cong\; (\mathsf{b} \Rightarrow \mathsf{B(x,\,y)}) \,\&\, (\mathsf{b} \Rightarrow \mathsf{C(x,\,y)})$$

Предикатная программа состоит из набора определений предикатов вида:

$$\mathsf{A}(\mathsf{x}\colon \mathsf{y}) \equiv \textbf{pre } \mathsf{P}(\mathsf{x}) \ \{\mathsf{S}\} \ \textbf{post } \mathsf{Q}(\mathsf{x}, \mathsf{y}), \tag{1}$$

где A – имя определяемого предиката, x – аргументы, а y – результаты предиката, S – оператор, $\mathsf{P}(\mathsf{x})$ – предусловие, $\mathsf{Q}(\mathsf{x}, \mathsf{y})$ – постусловие. Однозначность программы S, тотальность и однозначность спецификации $[\mathsf{P}(\mathsf{x}), \mathsf{Q}(\mathsf{x}, \mathsf{y})]$ определяются, соответственно, формулами:

$$\mathsf{P}(\mathsf{x}) \,\&\, \mathsf{LS}(\mathsf{S})(\mathsf{x}, \mathsf{y}1) \,\&\, \mathsf{LS}(\mathsf{S})(\mathsf{x}, \mathsf{y}2) \Rightarrow \mathsf{y}1 = \mathsf{y}2$$
$$\mathsf{P}(\mathsf{x}) \Rightarrow \exists \mathsf{y}.\, \mathsf{Q}(\mathsf{x}, \mathsf{y})$$
$$\mathsf{P}(\mathsf{x}) \,\&\, \mathsf{Q}(\mathsf{x}, \mathsf{y}1) \,\&\, \mathsf{Q}(\mathsf{x}, \mathsf{y}2) \Rightarrow \mathsf{y}1 = \mathsf{y}2$$

Корректность[1] определения предиката (1) представляется формулой:

$$\mathsf{P}(\mathsf{x}) \Rightarrow [\mathsf{LS}(\mathsf{S})(\mathsf{x}, \mathsf{y}) \Rightarrow \mathsf{Q}(\mathsf{x}, \mathsf{y})] \,\&\, \exists \mathsf{y}.\, \mathsf{LS}(\mathsf{S})(\mathsf{x}, \mathsf{y}) \tag{2}$$

Доказательство корректности для однозначных программ с однозначной и тотальной спецификацией можно проводить по более простой формуле:

$$\mathsf{P}(\mathsf{x}) \,\&\, \mathsf{Q}(\mathsf{x}, \mathsf{y}) \Rightarrow \mathsf{LS}(\mathsf{S})(\mathsf{x}, \mathsf{y}) \tag{3}$$

Отметим, что доказывать однозначность спецификации и программы не требуется[2], достаточно доказать тотальность спецификации. Формула (3) является достаточным условием формулы корректности (2).

На базе формул (2, 3) построены две серии (R и L) правил доказательства корректности операторов языка P [3]. В разделе 2 определены методы дедуктивной верификации и программного синтеза на базе правил серии L. Они проще, чем соответствующие правила серии R.

Спецификация задачи суммирования дана в разделе 4. В разделах 5–7 рассматриваются три программы двоичного сложения – сумматоры с последовательным и параллельным переносом и сумматор Линь. Представлены лишь ключевые фрагменты процесса верификации и синтеза программ сумматоров. Полный набор условий корректности, дополнительные леммы и все доказательства доступны в формате системы PVS: http://www.iis.nsk.su/persons/vshel/files/adders.zip. Опыт верификации и синтеза и обзор работ представлены в разделах 8 и 9.

2. Дедуктивная верификация и программный синтез

Методы верификации и синтеза определим на примере оператора суперпозиции в правой части определения предиката:

$$\mathsf{A}(\mathsf{x}\colon \mathsf{y}) \equiv \textbf{pre } \mathbf{P}(\mathbf{x}) \ \{\mathsf{B}(\mathsf{x}\colon \mathsf{z});\ \mathsf{C}(\mathsf{z}\colon \mathsf{y})\} \ \textbf{post } \mathsf{Q}(\mathsf{x}, \mathsf{y}) \tag{4}$$

[1]Здесь и далее термин "корректность" используется в смысле тотальной корректности.

[2]Однозначность спецификации следует из ее тотальности и формулы (3). Доказана однозначность программы при условии, что все используемые базисные операции являются однозначными

Предположим, что операторы B и C корректны относительно спецификаций $[P_B(x), Q_B(x,z)]$ и $[P_C(z),\ Q_C(z,\ y)]$ соответственно. Пусть спецификация $[P(x), Q(x,\ y)]$ тотальна. Не трудно доказать, что корректность предиката A гарантируется в случае истинности правил[3]:

Правило LS1. $P(x) \vdash P_B(x)$

Правило LS2. $P(x)\ \&\ Q(x,\ y)\ \&\ Q_B(x,\ z) \vdash P_C(z)\ \&\ Q_C(z,\ y)$

Задача верификации. Пусть имеется определение (4). Операторы B и C корректны относительно своих спецификаций. Для доказательства корректности определения (4) требуется доказать тотальность спецификации $[P(x),\ Q(x,\ y)]$ и истинность правил LS1 и LS2.

Задача синтеза. Требуется построить программу для предиката $A(x{:}\,y)$, представленного спецификацией $[P(x),\ Q(x,\ y)]$. Допустим, доказана тотальность спецификации. Пусть для некоторых предикатов $P_B(x)$, $Q_B(x,\ z)$, $P_C(z)$ и $Q_C(z,\ y)$ удалось доказать истинность правил LS1 и LS2. Тогда для предиката A синтезируется определение (4) с включением в программу двух новых предикатов: $B(x{:}\,z)$ со спецификацией $[P_B(x), Q_B(x,\ z)]$ и $C(z{:}\,y)$ со спецификацией $[P_C(z),\ Q_C(z,\ y)]$. Дальнейшей целью является синтез программ для B и C.

3. Технология предикатного программирования

Технология предикатного программирования определяет спецификацию, разработку (возможно, в режиме синтеза), верификацию и трансформационную реализацию произвольной программы в классе задач дискретной и вычислительной математики. К предикатной программе применяется набор трансформаций с получением эффективной программы на императивном расширении языка P и последующей конвертацией на любой из императивных языков: C, C++, ФОРТРАН и др. Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование структурных объектов низкоуровневыми структурами с использованием массивов и указателей.

К предикатной программе, разработанной для реализации интегральных схем, применяются последовательности, составленные из 1-й и 3-й трансформаций. Для рекурсивно определяемого предиката реализуется его полная *развертка* многократным применением подстановки определения предиката на место рекурсивных вызовов. Полученная программа легко транслируется на языки VHDL и Verilog, ориентированные на проектирование интегральных схем.

[3]Истинность правил определяется как истинность формул, получающихся заменой $\vdash$ на $\Rightarrow$

4. Спецификация сумматора

Существует много алгоритмов двоичного сложения. И в настоящее время появляются новые все более совершенные сумматоры, в частности, для квантовых компьютеров. Простейшим является алгоритм поразрядного сложения с последовательным переносом Ripple carry adder [4], реализующий известную школьную схему сложения столбиком.

Традиционно 64- и 32-битные сумматоры собираются из 16-битных, а те, в свою очередь, – из 4-битных. Простейшим блоком в этой схеме является однобитный сумматор, известный как "полный сумматор", реализующий сложение трех битов x, y и cin, где cin – значение входного переноса от предыдущего одноразрядного блока. Результатом полного сумматора являются бит z и бит cout – выходной перенос, поставляемый следующему блоку. Спецификация полного сумматора представляется следующими описаниями:

$$\textbf{type}\ \mathsf{bit} = \textbf{bool};$$
$$\textbf{formula}\ \mathsf{b2n}(\mathsf{bit\ x}\colon \textbf{nat}) = \mathsf{x}?\,1\colon 0;$$
$$\textbf{formula}\ \mathsf{fullAdder}(\mathsf{bit\ x,\ y,\ cin,\ z,\ cout}) =$$
$$\mathsf{b2n(x)} + \mathsf{b2n(y)} + \mathsf{b2n(cin)} = \mathsf{b2n(z)} + 2 * \mathsf{b2n(cout)}$$

Функция b2n определяет преобразование бита x в значение 0 или 1.

В настоящей работе рассматриваются три программы двоичного сложения – сумматоры RippleCarryAdder, CarryLookaheadAdder и LingAdder [4] с одинаковой спецификацией [**true**, Adder(n, a, b, c0, s, cout)]. Предикат Adder определяет значение суммы a + b + c0, где a и b – n-битные числа, представленные битовыми векторами (с индексами от 0 до n − 1), а бит c0 – входной перенос:

$$\textbf{type}\ \mathsf{below}(\textbf{nat}\ \mathsf{n}) = 0\,..\,\mathsf{n}-1;$$
$$\textbf{type}\ \mathsf{bvec}(\textbf{nat}\ \mathsf{n}) = \textbf{array}(\mathsf{bit}, \mathsf{below}(\mathsf{n}));$$
$$\textbf{formula}\ \mathsf{Adder}(\textbf{nat}\ \mathsf{n},\ \mathsf{bvec(n)\ a},\ \mathsf{b},\ \mathsf{bit\ c0},\ \mathsf{bvec(n)\ s}, \mathsf{bit\ cout}) =$$
$$\mathsf{val(n,a)} + \mathsf{val(n,b)} + \mathsf{b2n(c0)} = \mathsf{val(n,s)} + \mathsf{b2n(cout)} * 2\hat{}\,\mathsf{n};$$

Здесь функция val(n, a) вычисляет натуральное значение n-битного числа a. Результатом сложения является n-битное число s и выходной перенос cout, определяющий бит с индексом n в значении суммы.

5. Сумматор с последовательным переносом

Сумматор RippleCarryAdder реализует школьную схему сложения столбиком. Рассмотрим более общую задачу Ripple1(n, a, b, c0: s, c), в которой вместе со значением суммы s определяются переносы по каждому биту при сложении чисел a и b. Значения переносов фиксируются в битовом векторе c длины n + 1. Задача Ripple1 определяется спецификацией [**true**, Adder_rip(n, a, b, c0, s, c)]:

$$\textbf{formula}\ \mathsf{Adder\_rip}(\textbf{nat}\ \mathsf{n},\ \mathsf{bvec(n)\ a},\ \mathsf{b},\ \mathsf{bit\ c0},\ \mathsf{bvec(n)\ s},\ \mathsf{bvec(n+1)\ c}) =$$
$$\mathsf{c[0]} = \mathsf{c0}\ \&\ \textbf{forall}\ \mathsf{below(n)\ j.\ fullAdder(a[j],\ b[j],\ c[j],\ s[j],\ c[j+1])}$$

Предикат Adder_rip определяет алгоритм, представленный на рис.1.

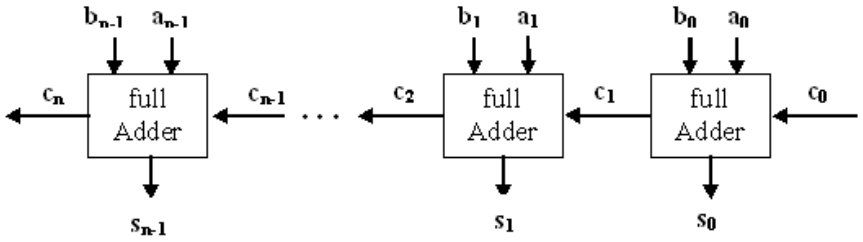

Рис. 1. Сумматор с последовательным переносом

Очевидно, что cout = c[n]. Это определяет следующую программу:

$$\begin{aligned} &\mathsf{RippleCarryAdder}(\mathbf{nat}\ \mathsf{n}, \mathsf{bvec(n)}\ \mathsf{a}, \mathsf{b}, \mathsf{bit\ c0}\colon \mathsf{bvec(n)}\ \mathsf{s}, \mathsf{bi\ tcout}) \equiv \\ &\quad \{\, \mathsf{Ripple1(n,\ a,\ b,\ c0}\colon \mathsf{s,\ bvec(n+1)\ c});\ \mathsf{cout} = \mathsf{c[n]}\,\} \\ &\quad \mathbf{post}\ \mathsf{Adder(n,\ a,\ b,\ c0,\ s,\ cout)}; \end{aligned} \tag{5}$$

Оператор суперпозиции в (5) является композицией более общего вида $\mathsf{B(x}\colon \mathsf{z,\ t)}; \mathsf{C(x,\ z}\colon \mathsf{y})$. При условии корректности предиката $\mathsf{B(x}\colon \mathsf{z,\ t})$ относительно спецификации $[\mathsf{P_B(x)},\ \mathsf{Q_B(x,\ z,\ t)}]$ доказательство корректности такой суперпозиции реализуется правилами:

Правило LS13. $\mathsf{P(x)} \vdash \mathsf{P_B(x)}$

Правило LS14. $\mathsf{P(x)} \,\&\, \mathsf{Q(x,\ z,\ t)} \,\&\, \mathsf{Q_B(x,\ z,\ t1)} \vdash \mathsf{t} = \mathsf{t1}\ \&\ \mathsf{LS(C(x,\ z}\colon \mathsf{y))}$

В соответствии с LS14 корректность (5) гарантируется доказательством леммы:

$$\begin{aligned} &\mathsf{Adder\_rip\_ls14}: \mathbf{lemma} \\ &\quad \mathsf{Adder(n,\ a,\ b,\ c0,\ s,\ cout)} \,\&\, \mathsf{Adder\_rip(n,\ a,\ b,\ c0,\ s1,\ c)} \ \Rightarrow\ \mathsf{s} = \mathsf{s1}\ \&\ \mathsf{cout} = \mathsf{c[n]} \end{aligned}$$

6. Сумматор с параллельным переносом

Переносы – элементы вектора c – вычисляются последовательно в сумматоре RippleCarryAdder. Второй сумматор CarryLookaheadAdder вычисляет переносы параллельно. Сначала создаются битовые векторы *протяжки* p и *генерации* g:

$$\begin{aligned} &\mathbf{formula}\ \mathsf{PG(nat\ n,\ bvec[n]\ a,\ b,\ p,\ g)} = \\ &\qquad \mathbf{forall}\ \mathsf{below(n)\ k.}\ (\mathsf{p[k]} = (\mathsf{a[k]}\ \mathbf{xor}\ \mathsf{b[k]})) \,\&\, (\mathsf{g[k]} = (\mathsf{a[k]} \,\&\, \mathsf{b[k]})); \end{aligned}$$

Вычисление суммы s и переносов c представлено предикатом Adder_look:

$$\begin{aligned} &\mathbf{formula}\ \mathsf{carry\_pg(nat\ n,\ bvec(n)\ p,\ g,\ bit\ c0,\ bvec(n+1)\ c)} = \\ &\quad \mathsf{c[0]} = \mathsf{c0}\ \&\ \mathbf{forall}\ \mathsf{below(n)\ k.\ c[k+1]} = (\mathsf{g[k]} \vee \mathsf{c[k]} \,\&\, \mathsf{p[k]}); \\ &\mathbf{formula}\ \mathsf{sum\_pg(nat\ n,\ bvec(n)\ p,\ bvec(n+1)\ c,\ bvec(n)\ s)} = \\ &\quad \mathbf{forall}\ \mathsf{below(n)\ k.\ s[k]} = (\mathsf{p[k]}\ \mathbf{xor}\ \mathsf{c[k]}); \\ &\mathbf{formula}\ \mathsf{Adder\_look(nat\ n,\ bvec(n)\ a,\ b,\ bit\ c0,\ bvec(n)\ s,\ bvec(n+1)\ c)} = \\ &\quad \mathbf{exists}\ \mathsf{bvec(n)\ p,\ g.\ PG(n,\ a,\ b,\ p,\ g)} \,\&\, \mathsf{carry\_pg(n,\ p,\ g,\ c0,\ c)} \,\&\, \mathsf{sum\_pg(n,\ p,\ c,\ s)}; \end{aligned}$$

Многократно применяя формулу $\mathsf{c}[\mathsf{k}+1] = (\mathsf{g}[\mathsf{k}] \vee \mathsf{c}[\mathsf{k}] \,\&\, \mathsf{p}[\mathsf{k}])$ для вхождений $\mathsf{c}[\mathsf{k}]$ в правой части можно выразить биты переносов $\mathsf{c}[\mathsf{k}+1]$ только через биты протяжки и генерации. В итоге получим алгоритм сумматора с параллельным вычислением переносов; см. рис. 2.

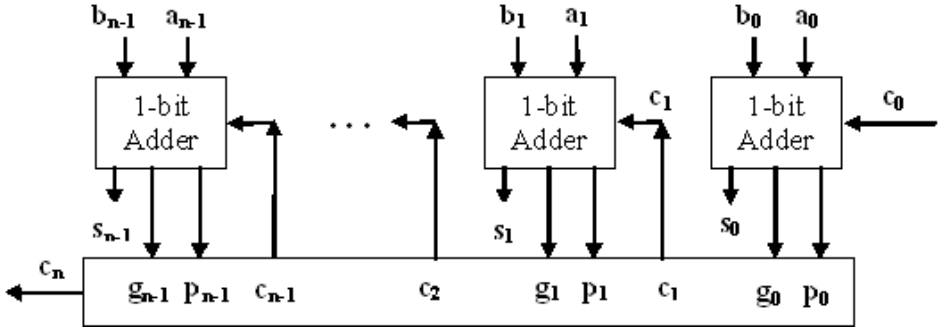

Рис. 2. Сумматор с параллельным переносом

Задачу со спецификацией [**true**, Adder_look(n, a, b, c0, s, c)] обозначим предикатом Carry1(n, a, b, c0: s, c). Программа для CarryLookaheadAdder сводится к программе для предиката Carry1, аналогичной программе (5), и ее корректность может быть доказана леммой:

Adder_look_ls14 : **lemma**

$$\mathsf{Adder}(\mathsf{n}, \mathsf{a}, \mathsf{b}, \mathsf{c0}, \mathsf{s}, \mathsf{cout}) \,\&\, \mathsf{Adder\_look}(\mathsf{n}, \mathsf{a}, \mathsf{b}, \mathsf{c0}, \mathsf{s1}, \mathsf{c}) \;\Rightarrow\; \mathsf{s} = \mathsf{s1} \;\&\; \mathsf{cout} = \mathsf{c}[\mathsf{n}]$$

Лемма аналогична Adder_rip_ls14. Ее доказательство было достаточно трудоемким, а доказательство Adder_look_ls14 было бы намного сложнее. Оказывается, проблема легко решается доказательством тождества:

Adder_look_rip : **lemma**

$$\mathsf{Adder\_look}(\mathsf{n}, \mathsf{a}, \mathsf{b}, \mathsf{c0}, \mathsf{s}, \mathsf{c}) \Leftrightarrow \mathsf{Adder\_rip}(\mathsf{n}, \mathsf{a}, \mathsf{b}, \mathsf{c0}, \mathsf{s}, \mathsf{c})$$

7. Сумматор Линь

Сумматор CarryLookaheadAdder быстрее, чем RippleCarryAdder, однако требует больше энергетических затрат, поскольку использует больше логических операций. По этим двум показателям третий сумматор LingAdder лучше, чем CarryLookaheadAdder. Сумматор LingAdder является усовершенствованной версией сумматора с параллельным переносом. Система уравнений Линь, представленная ниже, выводится в работе [8] из формул PG, carry_pg и sum_pg для сумматора CarryLookaheadAdder. В

сумматоре LingAdder используются битовые векторы h и t вместо p.

type n1below(**nat** n) = $-1\,..\,n-1$;
type tvec(**nat** n) = **array**(bit, n1below(n));
formula TG(**nat** n, bvec(n) a, b, g, tvec(n) t) =
    $t[-1] = \mathbf{true}\ \&\ \mathbf{forall}$ below(n) k. $(t[k] = (a[k] \vee b[k]))\ \&\ (g[k] = (a[k]\ \&\ b[k]))$;
formula harry(**nat** n, bvec(n) g, tvec(n) t, bit c0, bvec(n + 1) h) =
    $h[0] = c0\ \&\ \mathbf{forall}$ below(n) k. $h[k+1] = (g[k] \vee t[k-1]\ \&\ h[k])$;
formula sum_h(**nat** n, bvec(n) g, tvec(n) t, bvec(n + 1) h, bvec(n) s) =
    forall below(n) k. $s[k] = ((t[k]\ \mathbf{xor}\ h[k+1]) \vee g[k]\ \&\ t[k-1]\ \&\ h[k])$;
formula HC(**nat** n, bit c0, tvec(n) t, bvec(n + 1) h, c) =
    $c[0] = c0\ \&\ \mathbf{forall}$ below(n) k. $c[k+1] = (t[k]\ \&\ h[k+1])$;

Соответствующая обобщенная задача вычисления суммы s и переносов c представлена предикатом Adder_ling:

formula Adder_ling(**nat** n, bvec(n) a, b, bit c0, bvec(n) s, bvec(n + 1) c) =
  exists bvec(n) g, tvec(n) t, bvec(n + 1) h.
    TG(n, a, b, g, t) & harry(n, g, t, c0, h) & sum_h(n, g, t, h, s) & HC(n, c0, t, h, c);

Как и для сумматора CarryLookaheadAdder доказательство корректности программы существенно упрощается доказательством тождества:

Adder_ling_look : **lemma**
    Adder_ling(n, a, b, c0, s, c) $\Leftrightarrow$ Adder_look(n, a, b, c0, s, c)

Естественный способ доказательства леммы – повторить серию математических выкладок работы [8], что соответствует:

Adder_look_ling_imp : **lemma**
    Adder_look(n, a, b, c0, s, c) $\Rightarrow$ Adder_ling(n, a, b, c0, s, c)

Для доказательства в обратную сторону достаточно доказать однозначность предиката Adder_ling:

Adder_ling_uniq : **lemma**
    Adder_ling(n, a, b, c0, s, c) & Adder_ling(n, a, b, c0, s1, c1) $\Rightarrow$ s = s1 & c = c1

Отметим, что обоснование единственности решения системы уравнений Линь в работе [8] отсутствует. Введем предикаты: tg(n, a, b: g, t) для спецификации [**true**, TG(n, a, b, g, t)], Harry(n, g, t, c0: h) для [**true**, harry(n, g, t, c0, h)], sum(n, g, t, h: s) для [**true**, sum_h(n, g, t, h, s)], hc(n, c0, t, h: c) для [**true**, HC(n, c0, t, h, c)]. Тогда правая часть определения предиката Adder_ling эквивалентна:

LS(tg(n, a, b: g, t); Harry(n, g, t, c0: h); {sum(n, g, t, h: s) || hc(n, c0, t, h: c)}),

что определяет программу для [**true**, Adder_ling(n, a, b, c0, s, c)].

8. Опыт верификации и синтеза сумматоров

Сумматоры RippleCarryAdder, CarryLookaheadAdder и LingAdder были запрограммированы на языке P. Для каждой программы были сгенерированы условия корректности в соответствии с системой правил серии L. Сгенерированные формулы оттранслированы на язык спецификаций известной системы автоматического[4] доказательства PVS [5]. Доказательство формул на PVS оказалось нетривиальным и в 5-10 раз[5] более трудоемким процессом по сравнению с обычной разработкой программ без формальной верификации.

Трансляция программ сумматоров на языки VHDL и Verilog, ориентированные на проектирование интегральных схем, возможна после развертки рекурсивных определений предикатов для конкретного значения n. Опыт доказательства корректности развернутой программы сумматора RippleCarryAdder для $\mathsf{n} = 4$ показал, что доказательство оказывается более сложным, чем для произвольного n.

Попытки доказательства условий корректности для LingAdder применением привычной техники доказательства по индукции оказались безуспешными. Решено было воспроизвести в доказательстве математические выкладки работы [8], что определило переход к доказательству в стиле синтеза и начало исследований по синтезу предикатных программ. Отметим, что дедуктивная верификация сумматора Линь проведена впервые.

Верификация и синтез зеркальны. Если для предикатной программы сгенерируем условия корректности, а затем по этим формулам проведем синтез, то получим эквивалентную программу. Наоборот, если на основе спецификации программы и набора формул синтезируем программу, а затем для программы сгенерируем условия корректности, то получим эквивалентную систему формул. Однако результатом синтеза будет иная программа, чем при обычном программировании. Причина в том, что в процессе программирования мы невольно совершаем оптимизацию программы. Например, при обычном программировании предикат hc вырождается в оператор $\mathsf{cout} = (\mathsf{t}[\mathsf{n}] \,\&\, \mathsf{h}[\mathsf{n}+1])$. Оптимизации программы меняют ее логику, обычно в сторону усложнения. Поэтому верификация оказывается более сложной и трудоемкой, чем синтез. При синтезе программы доказательство и связанное с ним построение теорий в PVS реализуются более естественным образом, более вариативно и с большей согласованностью между теориями. Это дает возможность сократить объем доказательств почти в два раза по сравнению с дедуктивной верификацией, особенно если требуется синтезировать несколько разных программ с одной спецификацией.

9. Смежные работы

Наиболее популярным методом синтеза является извлечение программы из конструктивного (интуиционистского) доказательства (метод proofs-as-programs) [6] с

[4]Для всякого доказательства система PVS автоматически контролирует его правильность. Доказать автоматически PVS может лишь достаточно простые утверждения.

[5]Оценка автора по опыту дедуктивной верификации примерно 20 небольших программ.

применением правил резолюции и унификации. Синтез предикатных программ проводится в классической логике и, в соответствии с обзором [9], реализуется по шаблонам формул (метод schema-guided); в качестве шаблонов используются правила доказательства корректности. Метод шаблонов применяется в основном для синтеза логических программ. Имеется работа [10], где метод шаблонов применяется для синтеза императивных программ. В этой работе, возможно впервые, проявилась декларированная в разделе 8 симметричность (зеркальность) дедуктивной верификации и программного синтеза. В *предикативном программировании* Э. Хехнера [7] программа синтезируется как результат последовательных уточнений (refinements):

$$\mathsf{Q}(\mathsf{x},\,\mathsf{y}) \Leftarrow \mathsf{Q}_1(\mathsf{x},\,\mathsf{y}) \Leftarrow \mathsf{Q}_2(\mathsf{x},\,\mathsf{y}) \Leftarrow \ldots \Leftarrow \mathsf{Q}_\mathsf{n}(\mathsf{x},\,\mathsf{y}) \Leftarrow \mathsf{LS}(\mathsf{S})(\mathsf{x},\,\mathsf{y}).$$

Однако вывод $\mathsf{LS}(\mathsf{S})(\mathsf{x},\,\mathsf{y}) \Rightarrow \mathsf{Q}(\mathsf{x},\,\mathsf{y})$ гарантирует лишь частичную корректность программы S. Данная формула — часть универсальной формулы корректности (2).

Среди многочисленных работ по дедуктивной верификации сумматоров наиболее показательной является [12]. Представление складываемых чисел в виде суперсписков (powerlists) – списков длиной 2^k – позволяет проводить доказательство в рамках логики первого порядка. В нашем случае использование битовых векторов определяет логику второго порядка. Несмотря на заверения авторов, верификация программы сумматора Ripple carry вряд ли является проще ранее представленных. Верификация сумматора Carry look-ahead сводится к доказательству эквивалентности Ripple carry и Carry look-ahead. Подобное сведение реализуется также в нашей работе (лемма Adder_look_rip) и других работах.

Методы верификации и синтеза, предлагаемые в настоящей работе, опробованы примерно для 20 небольших программ. Доказательство на PVS формул корректности требовало времени в 5—10 раз больше времени обычного программирования. Синтез программ стандартных функций floor, isqrt и ilog2 [11] оказался по трудоемкости существенно меньшим по сравнению с дедуктивной верификацией этих программ. Представленные методы верификации и синтеза целесообразны для применения в приложениях с высокой ценой ошибки.

Автор благодарен анонимному рецензенту за множество полезных замечаний.

Список литературы

1. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Новосибирск, 2010. 42 с. (Препр. / ИСИ СО РАН; N 153).

2. Shelekhov V. The language of calculus of computable predicates as a minimal kernel for functional languages // BULLETIN of the Novosibirsk Computing Center. Series: Computer Science. IIS Special Issue. 29(2009). 2009. P. 107-117.

3. Шелехов В.И. Модель корректности программ на языке исчисления вычислимых предикатов. Новосибирск, 2007. 50 с. (Препр. / ИСИ СО РАН; N 145).

4. Flynn M.J., Oberman S.F. Advanced computer arithmetic design. Wiley, New York. 2001. 344 p. (http://en.wikipedia.org/wiki/Adder_(electronics))

5. Owre S., Rushby J.M., and Shankar N. PVS: A Prototype Verification System // LNCS, 607. 1992. P. 748–752. (http://pvs.csl.sri.com/)

6. Sorensen M.H., Urzyczyn P. Lectures on the Curry-Howard Isomorphism // Logic and the Foundations of Mathematics. 2006. Vol. 149. 457 p.

7. Hehner E.C.R. A Practical Theory of Programming, second edition // ON M5S 2E4, University of Toronto. 2004. 242p. (http://www.cs.toronto.edu/ hehner/aPToP/)

8. Doran R. W. Variants of an Improved Carry Look-Ahead Adder // IEEE Transactions on Computers. 1988. Vol. 37, No.9. P. 1110–1113.

9. Basin D., DeVille Y., Flener P., Hamfelt A., and Nilsson J. Synthesis of programs in computational logic // LNCS, 3049. P. 30–65, 2004.

10. Srivastava S., Gulwani S., and Foster J. From Program Verification to Program Synthesis. POPL, 2010. P. 313–326.

11. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций floor, isqrt и ilog2 в технологии предикатного программирования // Тр. 12-й межд. конф. "Проблемы управления и моделирования в сложных системах". Самарский научный центр РАН, 2010. С. 622-630.

12. Kapur D., Subramaniam M. Mechanical verification of adder circuits using Rewrite Rule Laboratory // Formal Methods in System Design. 1998. Vol. 13, No. 2. P. 127–158.

Verification and synthesis of addition programs under the rules of statement correctness

Shelekhov V.I.

Keywords: predicate programming, deductive verification, program synthesis, total correctness of program

Deductive verification and synthesis of binary addition programs are performed on the base of the rules of program correctness for statements of the predicate programming P language. The paper presents the sketch of verification and synthesis of the programs for the Ripple carry, Carry look-ahead and Ling adders. The correctness conditions of the programs were translated into the specification language of the PVS verification system. The time needed to prove the program correctness conditions in the PVS is more than the time of the ordinary programming by a factor of 10. However, for program synthesis, development of PVS theories and proofs are easier and faster than that for program verification.

Сведения об авторе: Шелехов Владимир Иванович, Институт систем информатики им. А.П. Ершова, зав. лаб. системного программирования