Применение логики программы для спецификации и верификации реактивных систем[1]

Тумуров Эрдэм Гармаевич

Институт систем информатики имени А.П. Ершова СО РАН (Новосибирск)

e-mail:erdemus@gmail.com

Введение

Цель настоящей работы – расширить понятие логики программы для представительного класса программ – реактивных систем.

В работе [1] определено понятие логики для программ, невзаимодействующих с внешним окружением программы. Логика такой программы – предикат (логическое утверждение), определяющий логику решения задачи и являющийся точным эквивалентом программы.

Для оператора A с аргументами x и результатами y используется обозначение A(x : y), а для логики оператора – L(A(x : y)). Пусть B и C – операторы, x, y и z – непересекающиеся наборы переменных, E – логическое выражение, зависящее от x. Определим логику для оператора суперпозиции, параллельного оператора и условного оператора, соответственно:

$$L(B(x : z);\ C(z : y)) \cong \exists z.\ L(B(x : z)) \wedge L(C(z : y)) \quad (1)$$

$$L(B(x : y) \parallel C(x : z)) \cong L(B(x : y)) \wedge L(C(x : z)) \quad (2)$$

$$L(\textbf{if}\ (E)\ B(x : y)\ \textbf{else}\ C(x : y)) \cong E \wedge L(B(x : y)) \vee \neg E \wedge L(C(x : y)) \quad (3)$$

Тотальная корректность программы S(x : y) относительно спецификации в виде предусловия P(x) и постусловия Q(x, y) определяется формулой:

$$P(x) \Rightarrow [L(S(x : y)) \Rightarrow Q(x, y)] \wedge \exists y.\ L(S(x : y)) \quad (4)$$

Эффективно и просто строится логика императивных и функциональных программ для различных выражений, операторов и типов, за исключением указателей. На базе логики программы определена формула тотальной корректности программы относительно спецификации в виде предусловия и постусловия. Разработана система правил доказательства корректности программ для различных операторов, а также разработан метод дедуктивной верификации предикатных [2] программ, имеющий преимущества по сравнению с классическим методом Хоара [3].

Для реактивной системы логика программы [4] – набор предикатов на переменных состояния системы. Предикат из этого набора истинен после исполнения некоторой очередной акции трассы, составленной перемешиванием действий (акций) параллельных взаимодействующих процессов. *Действие* – максимальный фрагмент кода программы процесса (без циклов внутри), для которого логика легко строится. Содержательное описание алгоритма функционирования реактивной системы формализуется в виде спецификации, представленной машиной мета-состояний, как аппроксимации логики программы на множестве трасс.

Разрабатываемый аппарат ориентирован на разработку, тестирование, моделирование и верификацию программной и аппаратной части встроенных систем аэрокосмической отрасли, энергетики, медицины и др. приложений, где необходима предельная надежность систем.

[1] Работа выполнена при поддержке РФФИ, проект 12-01-00686

Логика реактивных систем

Условие тотальной корректности (4) не применимо для взаимодействующей программы, так как такая программа может не завершаться или завершается без данных и, как следствие, может не иметь постусловия. Тем не менее, понятие логики программы может быть расширено для класса взаимодействующих программ. Ограничимся обычными реактивными системами, которые являются подмножеством различных взаимодействующих программ.

Реактивная система состоит из нескольких процессов, работающих параллельно, и взаимодействующих между собой и внешним окружением путем обмена сообщениям, передающихся через каналы. Используются новые виды операторов:

loop { B } **send** m(e) **receive** m(x) { B } **else** C **with** (y) { B } F || G

Здесь B и C – утверждения, x и y – списки переменных, e – список выражений, m – имя сообщения, F || G – *параллельная композиция* процессов F и G, **with** (y) { B } – *защищенный оператор* (монитор), блокирующий доступ к переменным y для других процессов, пока не завершится исполнение B. Выход из цикла **loop** { B } реализуется оператором **break** внутри B.

Если оператор **loop** { B } исполняется бесконечно, логика после цикла – **false**. В случае, когда цикл может завершиться оператором **break** внутри B, сложно адекватно определить логику всего цикла, так как определение в виде наименьшей неподвижной точки будет неправильным; существование других полезных определений сомнительно.

Канал – очередь сообщений, отправленных одним процессом, но еще не принятых другим. Определим логику операторов **send** и **receive**:

L(**receive** m(x) C **else** D) ≅ **if** (A≠**nil**) x=A.car ∧ A'=A.cdr ∧ L(C) **else** L(D)
L(**send** m(e)) ≅ A'=append(A, e)

где A – канал сообщений одного вида m, A.car – первый элемент списка, A.cdr – список без первого элемента, A' – состояние канала после выполнения оператора.

Действие – это максимальный фрагмент исходного кода без циклов. Логика любого действия может быть легко построена, используя формулы (1)-(3). Из исходного кода процесса со структурой вложенных циклов можно построить *граф действий*, где каждая дуга определяет *переход* от одного действия к другому. *Трасса* – это последовательность действий на некотором пути в графе действий. Исполнение процесса может быть представлено как исполнение последовательности действий некоторой трассы.

Состояние параллельного оператора F || G – это множество переменных и их значений. Состояние включает глобальные переменные, модифицируемые любым из этих процессов. *Канал* между F и G также принадлежит состоянию. Не допускается одновременный доступ к переменным состояния в параллельных процессах. Возможные конфликты устраняются с помощью защищенных операторов. Отметим, что канал неявно защищен во время исполнения **send** или **receive**. Для действий a из F и b из G параллельная композиция a || b может быть заменена последовательной a; b или b; a, производя то же самое изменение состояния. Это утверждение верно при условии, что для каждого действия ни одна из переменных состояния не может быть изменена более одного раза. Утверждение истинно и в случае нескольких модификаций внутри действия, если они обрамлены защищенным оператором.

Пусть F_T и G_T – множества трасс процессов F и G. Определим множество трасс $(F \,||\, G)_T$ как множество трасс $F_T * G_T$, построенных всеми возможными перемешиваниями (interleaving) трасс из F_T и G_T. **Утверждение:** параллельное исполнение композиции F || G эквивалентно последовательному исполнению некоторой ее трассы.

Логика параллельной композиции F || G для трассы из множества $F_T * G_T$ есть *логика трассы*, определяемая как конъюнкция логик действий, входящих в трассу. Трасса *достижима*, если ее логика не равна **false**.

Спецификация параллельной композиции F || G – это описание совместного поведения процессов. Формальная спецификация F || G может быть задана в форме *машины (конечных) мета-состояний* (ММС). Состояние (вершина) ММС называется *мета-состоянием*. Оно помечается предикатом над переменными состояния, который истинен, когда исполнение ММС достигает этого мета-состояния. Переход в ММС называется *мета-переходом*. Он состоит из последовательности действий, полученных перемешиванием

действий из F и G. Мета-переход может содержать условие, которое истинно во время его завершения.

Корректность параллельной композиции относительно MMC-спецификации. Для любой достижимой трассы t параллельной композиции F || G существует путь $p_1, p_2, \ldots, p_n$ в MMC, где p_j (j=1,...,n) – мета-состояния, и существуют подтрассы $t_1, t_2, \ldots, t_n$, такие что $t = t_1\, t_2 \ldots t_n$, и для каждого $k \le n$, $L(t_1\, t_2 \ldots t_k) \Rightarrow p_k$.

MMC спецификация может быть использована для детального анализа реактивных систем. После доказательства корректности соответствия спецификации и реактивной системы, MMC может быть использована для доказательства временных свойств живости и безопасности реактивной системы. Представленный подход может быть также использован для разработки back-end компилятора, который будет порождать формулы условий корректности для проверки на модели и дедуктивной верификации реактивных систем.

Логика программы для протокола ABP

Понятие логики программы и спецификации MMC иллюстрировано ниже на примере протокола чередования битов (Alternating Bit Protocol, ABP). Это простой коммуникационный протокол. Он состоит из параллельных процессов S и R, взаимодействующих через ненадежные каналы A и B.

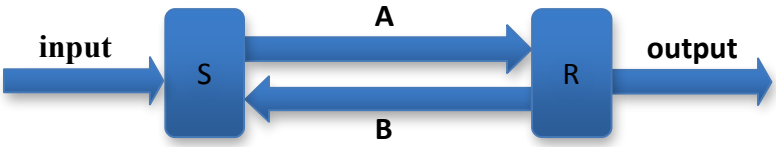

Рис.1. Схема протокола ABP

Управляющие биты n и m, имеющие значение **true** во время запуска, используются для управления процессом передачи данных. В начале отправитель S считывает блок данных для передачи d из входного потока и отправляет сообщение a(n, d) по каналу A. После получения сообщения, отправитель R записывает d в выходной поток, инвертирует бит m:=¬m, и отправляет сообщение-подтверждение b(n) по каналу B. После получения этого сообщения, отправитель S инвертирует бит n:=¬n и считывает следующих блок для передачи.

Каналы A и B ненадежные в том смысле, что они могут терять сообщения. Для обеспечения надежной передачи данных, отправитель S отсылает повторные сообщения a(n, d), пока не получит сообщение подтверждения с таким же битом n.

Программа протокола приведена ниже.

```
type Data;
message a(bool, Data), b(bool);
bool sa, sb;
process ABP()  { S || R }
process S() {
        bool n = true                                   // S1
        loop {
                input(Data d)                           // S2
                loop {
                        receive b(bool j) {             // S3
                                if (j = n) break        //
                        } else {                        //
                                input(sa);              //
                                if (sa) send a(n, d)    //
                        }
                }
                n = ¬n                                  // S4
        }
}
```

```
process R() {
        bool m = true                                   // R1
        loop {
                loop {
                        receive a(bool i, Data d) {     // R2
                                input(sb)               //
                                if (i = m) {            //
                                        output(d);      //
                                        if (sb) send b(i);  //
                                        break           //
                                } else {                //
                                        if (sb) send b(i);  //
                                }                       //
                        }                               //
                }
                m = ¬m                                  // R3
        }
}
```

Переменные sa и sb – флаги для моделирования потери сообщений. Они могут принимать произвольные значения, получаемые из внешнего окружения.

Далее представлены графы действий для процессов S и R.

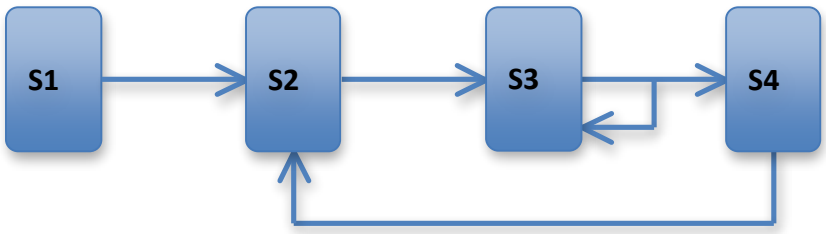

Рис. 2. Граф действий S

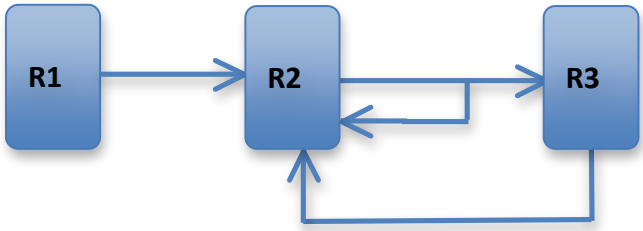

Рис. 3. Граф действий R

Глобальное состояние протокола ABP: G = (aS, aR, n, m, d, A, B), где aS и aR – текущие действия S и R, соответственно; n и m – управляющие биты S и R, соответственно; d – текущий блок данных в S; A и B – каналы для сообщений a и b, соответственно.

Множество имен действий: Actions = { S1, S2, S3, S4, R1, R2, R3 }.

Каналы-очереди A и B представлены списками-очередями сообщений, которые были отправлены, но еще не доставлены. Логики операторов **send** и **receive** для канала A:

L(**receive** a(n, d){ C } **else** D) ≅ **if** (A ≠ **nil**) |n, d| = A.car & A′ = A.cdr & L(C) **else** L(D);
L(**send send** a(n, y)) ≅ A′ = A + |n, y|

Здесь A.car – это первый элемент списка A, и A.cdr – список без первого элемента.

Оператор «+» - конкатенация списков и элементов в новый список.

Логики действий, которые были отмечены в коде разным цветом, построены с помощью формул (1)-(3):

```
L(S1) ≅ n = true & aS' = S2
L(S2) ≅ aS' = S3
L(S3) ≅ if (B ≠ nil) (B' = B.cdr & j = B.car &
                        (if (j = n) aS' = S4 else aS' = S3)
          else if (sa = true) A' = A + |n, d| & aS' = S3
          else A' = A & aS' = S3
L(S4) ≅ n' = ¬n & aS' = S2
L(R1) ≅ m = true & aR' = R2
L(R2) ≅ if (A ≠ nil) (|i, d| = A.car & A' = A.cdr &
             if (i = m) (if (sb=true) B'=B + |i| & aR' = R3 else B'= B & aR' = R3)
             else if (sb=true) B'=B+|i| & aR'=R2
             else B'= B & aR' = R2
          else A' = A & aR' = R2
L(R3) ≅ m'=¬m & aR'=R2
```

Краткая неформальная спецификация протокола ABP, представленная в начале приложения, формализуется в виде MMC спецификации на Рис. 4. X* обозначает последовательность X X ... X с нулем или более X, X+ обозначает последовательность X X ... X с одним или более X. Например, B=[(¬n)*(n)+] описывает канал B, начинающийся с возможно пустой последовательности сообщений b(¬n), и следующими одним или более сообщениями b(n). Ниже, на Рис. 4, для каждого мета-состояния внутри рамки приведен предикат в виде конъюнкции простых условий над переменными состояния. Условия, обозначенные возле мета-переходов не являются обязательными для MMC. Они используются для лучшего понимания срабатывания мета-переходов.

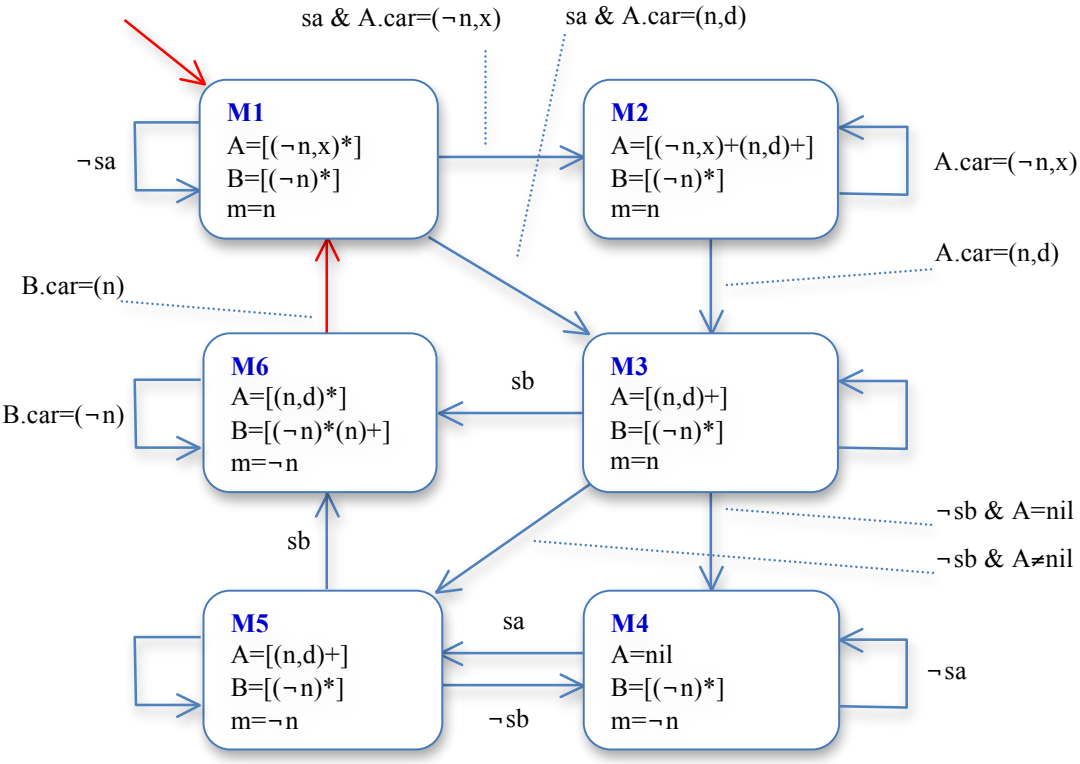

Рис. 4. MMC спецификация протокола ABP

d обозначает текущий блок для передачи, а x – предыдущий. Во время мета-перехода [M6, M1] исполняется инвертирование бита n = ¬n. По этой причине предикат A = [(n, d)*] & B = [(n)*] в мета-состоянии M6 заменяется на A = [(¬n, x)*] & B = [(¬n)*] в мета-состоянии M1.

Можно заметить, что мета-состояния M2 и M3 могут быть заменены одним мета-состоянием, порождающим другую MMC спецификацию. Аналогично, мета-состояния M3,

M4, и M5 могут быть объединены в одно мета-состояние. Напротив, мета-состояние M6 может быть разделено на два мета-состояния, разделяющих условие B=[(¬n)*(n)+] на два: B=[(¬n)+(n)+] и B=[(n)+]. Таким образом, можно построить много ММС спецификаций, от грубых до детализированных.

Каждый мета-переход может быть представлен как последовательность действий, смешанных из действий процессов S и R. Для ММС из Рис. 4 эти последовательности достаточно просты.

T(M1,M1): R2 **or** S3
T(M1,M2): S3
T(M1,M3): S3
T(M2,M2): S3 **or** R2
T(M2,M3): R2
T(M3,M3): S3
T(M3,M4): R2(S3*)R3
T(M3,M5): R2(S3*)R3
T(M3,M6): R2(S3*)R3
T(M4,M4): S3 **or** R2
T(M4,M5): S3
T(M5,M4): R2
T(M5,M5): S3
T(M5,M6): R2
T(M6,M6): S3 **or** R2
T(M6,M1): S3 S4 (R2*) S2

Рис. 5. Последовательности действий, выполняемых во время мета-переходов

Можем считать, что спецификация Рис. 4 достаточно детализирована, так как последовательности действий мета-переходов очень простые.

Литература

1. Шелехов В.И. Логика невзаимодействующих программ. // 4-я Российская школа-семинар «Синтаксис и семантика логических систем». – 2012.
2. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Новосибирск, 2010. – 42с. (Препр. / ИСИ СО РАН; N 153).
3. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12 (10). P. 576–585.
4. Шелехов В.И., Тумуров Э.Г. Логика невзаимодействующих программ и реактивных систем // Вестник БГУ 9/2012. С.81-90