УДК 004.415.5

МЕТОД СТАТИЧЕСКОЙ ПРОВЕРКИ ПОЛНОТЫ И НЕПРОТИВОРЕЧИВОСТИ В ФОРМАЛЬНЫХ МОДЕЛЯХ РАСПРЕДЕЛЕННЫХ ПРОГРАММНЫХ СИСТЕМ

А.В. Колчин, А.А. Летичевский, С.В. Потиенко

Институт кибернетики им. В.М. Глушкова НАН Украины,
03680, Киев, проспект Академика Глушкова, 40.
E-mail: kolchin_av@yahoo.com, Oleksandr.Letychevskyy@iss.org.ua, Stepan.Potiyenko@iss.org.ua

Описан метод выявления таких патологий формальных моделей, как неполнота и противоречивость, а так же гонки в параллельных процессах. Метод реализует проверку свойств на основании анализа описания переходов модели, при этом не строит пространство ее состояний.

The paper describes a new method for discovering of incompleteness, inconsistency and race conditions in formal models. The method implements the properties checking basing on model transitions description, and does not traverse model state space.

Введение

Автоматизация проверки правильности программных систем – актуальная задача современной программной инженерии. Ввиду известных проблем с проверкой достижимости (комбинаторный взрыв, алгоритмическая разрешимость), актуальны методы статической проверки [1], т. е. методы анализа переходов модели без порождения пространства ее состояний. Обнаруженная такими методами ошибка может не быть проблемой ввиду ее недостижимости, но, с другой стороны, отсутствие обнаруженных ошибок будет означать их отсутствие и в пространстве достижимых состояний. В качестве примеров удачных применений подобных методов в индустрии можно привести такие системы, как klocwork [2], FastTrack [3], Relay [4]. Существующие системы автоматической проверки программ опираются на структуру описания проверяемого кода, в частности, интенсивно используют структуру потока управления; явно указанная иерархия структур данных и область видимости переменных позволяет эффективнее строить отношения информационной зависимости и т.д. Особенность предлагаемого в данной работе метода состоит в том, что он работает с множеством отдельных переходов, на которых изначально не определен порядок выполнения (поток управления задан «неявно» атрибутами модели или отсутствует как таковой вообще), а структура модели не всегда позволяет однозначно выделить модули и процессы (все атрибуты модели могут быть «глобальными»). Такая ситуация характерна на начальных стадиях проектирования и разработки программного обеспечения – на этапах формализации требований, построения абстрактных прототипов.

Данная работа расширяет методы, описанные в работах [5–7], в которых предложены алгоритмы проверки полноты и непротиворечивости переходов. Далее описана формальная модель, определения проверяемых свойств, после чего описаны методы проверки и усовершенствования в них.

Формальная модель и проверяемые свойства

Проверяемая модель рассматривается как транзиционная система

$$< S,\ A,\ P,\ T > .$$

Здесь S – множество состояний системы, A – множество атрибутов, P – множество процессов, T – множество переходов, параметризированных идентификатором процесса. Каждое состояние из множества S представляет собой набор значений всех атрибутов из множества A.

Процессы работают параллельно асинхронно и модифицируют состояния системы совершая переходы из T. Переходы являются двойками пред- и постусловий:

$$\forall x(\alpha(p,R,x) \to \beta(p,R,x)) .$$

Предусловие $\alpha(p,R,x)$ – формула базового языка (обычно, исчисление предикатов первого порядка). Постуловие $\beta(p,R,x)$ – набор присваиваний вида $r_i := f(R,x)$, где f так же является формулой базового языка. Здесь p – идентификатор процесса, совершающего данный переход, R – множество атрибутов ($R \subset A, r_i \in R$), x – множество параметров перехода кроме p.

Противоречивость требований – распространенная ошибка моделей поведения систем. Например, требования T1:«если в буфер 1 пришел сигнал A, необходимо вызвать процедуру X» и T2: «если в буфер 2 пришел сигнал Term, необходимо завершить работу» противоречивы, так как допускают неоднозначность при ситуации, когда оба сигнала пришли в соответствующие буферы одновременно – согласно требованию T1

нужно
выполнять процедуру X, но по требованию T2 – завершить работу.

Непротиворечивость в [5] формально определяется отсутствием пересечений предусловий переходов одного процесса.

Определение 1. Два перехода u и v процесса p непротиворечивы, если следующая формула общезначима:

$$\neg(\exists x \alpha_u(p, R_u, x) \wedge \exists y \alpha_v(p, R_v, y)).$$

Здесь α_u и α_v – формулы предусловий, p – идентификатор процесса. Упрощенно:

$$\neg(\alpha_u \wedge \alpha_v).$$

Для доказательства непротиворечивости (детерминированности) в поведении процесса достаточно проверить все пары его переходов.

В работе [5] полнота определяется как общезначимость дизъюнкции предусловий всех переходов одного процесса.

Определение 2. Множество переходов процесса p полно, если следующая формула общезначима:

$$\exists x_1 \alpha_1(p, R_1, x_1) \vee \exists x_2 \alpha_2(p, R_2, x_2) \vee \ldots$$

Здесь α_i – формулы предусловий, R_i – наборы атрибутов, x_i – наборы параметров. Упрощенно:

$$\alpha_1 \vee \alpha_2 \vee \ldots$$

Это означает, что в данной точке потока управления всегда возможен переход. Если выполняется условие полноты, то в каждом состоянии система может совершить хотя бы один переход. Таким образом в полной системе отсутствуют тупики.

Работы [5–7] при проверке полноты и непротиворечивости опираются на заданную пользователем структуру проверяемой системы. В частности, предполагается, что у каждого процесса есть специальный атрибут state (как правило, задающий поток управления), который во всех состояниях имеет конкретное значение и сравнивается с конкретным значением в каждом предусловии. Это ограничение использовалось для разбиения множества переходов на соответствующие подмножества при проверках свойств: полнота проверялась отдельно для каждого процесса, причем на подмножестве переходов, которое строится по принципу сравнения атрибута state с одинаковыми значениями; аналогично, непротиворечивость проверялась на парах переходов из одного подмножества. Далее рассмотрены примеры свойств и проблемы их проверки на моделях разных стилей формализации, в частности, без упомянутых ограничений.

Метод проверки противоречивости

При проверке промышленных программных систем определение 1 непротиворечивости оказалось не практичным: в императивных языках выбор очередного выполняемого оператора внутри одного процесса всегда однозначен, т. е. такие модели детерминированы по построению. С другой стороны, даже при детерминированности всех процессов системы могут возникать патологии, связанные с параллельностью, например, коллизии при обращении к общей памяти.

Рассмотрим пример 1

Переход *u* процесса p1.	**Переход *v* процесса p2.**
Предусловие: p1.state = writing $\wedge$ x > 0	Предусловие: p2.state = sending
Постусловие: p1.state := idle; shared_attr := x	Постусловие: p2.state := ready; shared_attr := 0

Приведенные два перехода показывают пример гонки. В зависимости от последовательности их выполнения, атрибут shared_attr недетерминированно примет разные значения, однако проблем с противоречивостью не обнаружено, так как процессы p1 и p2, согласно определению 1, непротиворечивы (переходы будут в разных подмножествах ввиду принадлежности различным процессам). Более того, изменение структуры формализации может привести к обнаружению большого множества противоречивых переходов, несмотря на неизменность поведения модели (см. примеры 2, а и 2, б).

Пример 2, а. Модульная формализация	Пример 2, б. Формализация на глобальных атрибутах
Предусловие u: p1.state = writing $\wedge$ x > 0	**Предусловие u'**: state_of_p1 = writing $\wedge$ x > 0
Предусловие t: p1.state = writing $\wedge$ x $\leq$ 0	**Предусловие t'**: state_of_p1 = writing $\wedge$ x $\leq$ 0
Предусловие v: p2.state = sending	**Предусловие v'**: state_of_p2 = sending

В примере 2, а противоречий нет, тогда как в измененной структуре (атрибут state процесса p1 заменен глобальным state_of_p1, а процесса p2 соответственно на state_of_p2) будут обнаружены противоречия

(«ложные» с точки зрения исходной модели) между парами переходов u' и v' а так же t' и v', хотя модели находятся в трассовой эквивалентности.

Таким образом, можно резюмировать описанные выше проблемы:

– противоречивость определена как недетерминизм *одного* процесса; такое определение не позволяет выявлять гонки в параллельных процессах (пример 1).

– «ложная» противоречивость может возникнуть при изменении структурного описания модели, причем без изменения ее поведения (примеры 2, а, 2, б).

Усовершенствование метода поиска противоречий

Гонки в параллельных процессах (race condition) – распространенная, сложно диагностируемая потенциальная патология распределенных систем [3, 4, 8]. Трудности в ее устранении начинаются на стадии проявления – такие ошибки не всегда воспроизводимы, т.к. зависят от скорости выполнения процессов. Большая трудоемкость устранения дефектов, возникающих из-за гонок, стимулирует развитие автоматических методов их локализации. Ввиду высокой алгоритмической сложности [9] их выявление особенно актуально для статического анализа [3, 4]. Гонки различают на такие типы:

1) write-write. Это случай, когда два процесса параллельно пишут различные значения одному атрибуту, при этом конечный результат зависит от очередности выполнения.

2) write-read. Это случай, когда один процесс присваивает значение некоторому атрибуту, в то время как другой процесс читает значение этого атрибута при осуществлении своего перехода, при этом конечный результат зависит от очередности выполнения.

Теперь уточним понятие непротиворечивости. Для этого будем рассматривать переходы противоречивыми тогда, когда они не только недетерминированы, но при этом либо пересекаются множества атрибутов, значения которых меняются в их постусловиях (write-write race condition), либо один переход изменяет значения атрибутов, которые читает в предусловии другой переход (write-read race condition). Обозначим через $W(t)$ множество атрибутов из левых частей присваиваний перехода t, а через $R(t)$ – множество атрибутов, входящих в правые части присваиваний или в предусловие перехода t.

Определение 3. Если для двух переходов u и v выполнима формула $(W(u) \bigcap W(v)) \neq \varnothing \wedge \alpha_u \wedge \alpha_v$, то они находятся в отношении write-write противоречия.

Определение 4. Если для двух переходов u и v выполнима формула $(R(u) \bigcap W(v)) \neq \varnothing \wedge \alpha_u \wedge \alpha_v$ *EMBED*, то они находятся в отношении write-read противоречия.

Поскольку вычисление пересечения множества атрибутов эффективнее доказательства недетерминированности переходов, проверку следует начинать с построения множеств W, R.

Для усиления строгости проверки противоречий можно, в случае их обнаружения, дополнительно проверить, что

$$\neg(pt(pt(\alpha_u \wedge \alpha_v, \beta_u), \beta_v) \Leftrightarrow pt(pt(\alpha_u \wedge \alpha_v, \beta_v), \beta_u)) \vee$$

$$\vee \neg pt(pt(\alpha_u \wedge \alpha_v, \beta_u), \beta_v) \vee \neg pt(pt(\alpha_u \wedge \alpha_v, \beta_v), \beta_u) \text{ EMBED.}$$

Здесь $pt(s, \beta_t)$ – предикатный трансформер системы VRS [10], преобразующий состояние s с помощью постусловия β_t перехода t.

Приведем пример работы усовершенствованного метода.

Пример 3, а:

Переход *u*	**Переход *v***
Предусловие: state = writing $\wedge$ x > 0	Предусловие: state = sending $\wedge$ shared_attr < 0
Постусловие: state := idle; shared_attr := x	Постусловие: state := ready; shared_attr := 0

Несмотря на то, что оба перехода присваивают атрибуту shared_attr различные значения, здесь нет write-write гонки, т.к. переходы детерминированы – в предусловии проверяется общий атрибут state.

Пример 3, б:

Переход *u*	**Переход *v***
Предусловие: state_of_p1 = writing $\wedge$ x > 0	Предусловие: state_of_p2 = sending $\wedge$ x < 2
Постусловие: state_of_p1 := idle; shared_attr := x	Постусловие: state_of_p2 := ready; shared_attr := 1

Согласно определению 3, в этом примере есть write-write гонка. Однако, более строгая проверка обнаружит, что обе последовательности (когда они возможны) выполнения – u;v и v;u – приведут к одному и тому же значению атрибута shared_attr, так как единственное значение x, допускающее одновременное выполнение переходов – 1.

Пример 3, в:

Переход *u*	**Переход *v***
Предусловие: p1.state = init $\wedge$ x > 0	Предусловие: p2.state = waiting $\wedge$ y < x
Постусловие: p1.state := start; y := x	Постусловие: p2.state := ready

Пример показывает наличие write-read гонки: переход *u* записывает значение атрибута y, которое в свою очередь читает в предусловии переход *v*.

Отметим так же, что переходы *u* и *v* из примера 1, согласно усовершенствованному методу, будут находиться в отношении write-write гонки.

Метод проверки полноты

Практика использования метода, описанного в [5–7], выявила определенные недостатки: с одной стороны построение формализации было затруднено указанными ограничениями на использование атрибута state, с другой стороны, разбиение переходов на подмножества осуществлялось только на основании этого атрибута. Для пользователей в ряде случаев такие ограничения оказались слишком жесткими. Например, для моделирования прерываний, нужно было вообще не задавать никакого значения state в предусловии. В другом случае возникла потребность изменить атрибуты state разных процессов в одном постусловии. Это заставляло отказываться от его использования (везде задавалось одно значение) и использовать для задания порядка выполнения обычный атрибут, не имеющий синтаксических ограничений. Это привело к фактическому отсутствию разбиения переходов на подмножества, таким образом, в формулу полноты стали попадать предусловия всех переходов модели (см. примеры 4, а и 4, б). А так как промышленные системы содержат большое количество переходов (сотни и даже тысячи), возникли существенные затруднения как при доказательствах больших формул, так и при их анализе.

Пример 4, а. Модульная формализация	Пример 4, б. Формализация на глобальных атрибутах
Предусловие u: p1.state = st1 $\wedge$ x > 0	**Предусловие u'**: state_of_p1 = st1 $\wedge$ x > 0
Предусловие t: p1.state = st1 $\wedge$ x < 0	**Предусловие t'**: state_of_p1 = st1 $\wedge$ x < 0
Предусловие v: p1.state = st2 $\wedge$ x = 0	**Предусловие v'**: state_of_p1 = st2 $\wedge$ x = 0
Вердикт о неполноте:	**Вердикт о неполноте:**
State st1: x=0	state_of_p1=st1 $\wedge$ x=0 $\vee$
State st2: $\neg$ (x=0)	$\vee$ state_of_p2=st1 $\wedge$ $\neg$ (x = 0)

Как видно, формула неполноты теперь не разбита на подформулы, ее запись соответственно увеличилась; в больших примерах из-за отсутствия разбиения формула становится чрезмерно громоздкой и вердикт становится практически нечитаемым.

Другая проблема возникает при использовании техники так называемого «вотч-дога» (watch-dog). Такая техника часто применяется для устранения проблем с зацикливанием, причину которого так и не удалось установить.

Пример 5, а. Модульная формализация	Пример 5, б. Формализация на глобальных атрибутах*
Предусловие u: p1.state = idle $\wedge$ x > 0	**Предусловие u'**: x > 0
Предусловие t: p1.state = idle $\wedge$ x < 0	**Предусловие t'**: x < 0
Предусловие v: p2.state = idle $\wedge$ timer < max	**Предусловие v'**: timer < max
Предусловие w: p2.state = idle $\wedge$ timer $\geq$ max	**Предусловие w'**: timer $\geq$ max
Вердикт о неполноте:	**Вердикт о неполноте:**
State st1: x=0	No incompleteness

* Атрибуты state исключены ввиду их избыточности (здесь они всегда равны idle)

Из примера видно, что после изменения структуры формализации вердикт показывает отсутствие неполноты, хотя поведение модели (множество ее трасс) осталось неизменным. И хотя модель в целом не имеет достижимых тупиков (deadlock), процесс Process1 может зайти в локальный тупик при значении x=0 (состояние «активного тупика», livelock).

Таким образом, можно резюмировать описанные выше проблемы:

- слишком большая формула неполноты при отсутствии разбиения по атрибуту state (примеры 4, а, б);
- проблема «вотч-дога» – им может устраниться всякая неполнота, что приводит к потере обнаружения потенциальных ошибок (примеры 5, а, б).

Усовершенствование метода проверки полноты

В основе усовершенствования лежит принцип разделения переходов на подмножества с целью уменьшения размеров анализируемых формул, а так же более строгого анализа.

Итак, мы ставим перед собой две задачи: (1) не потерять возможную неполноту и (2) уменьшить размер формулы неполноты. Идея решения первой задачи такова: при построении формулы неполноты учитывать только переходы, непротиворечивые (в смысле определения 1) заданному, т. е.

$$incompl(t) \equiv \neg(\alpha_t \vee \bigvee_{\forall i \in T} \varphi_i \mid \varphi_i = \alpha_i \text{ если } \alpha_t \wedge \alpha_i \equiv \varnothing, \text{ иначе } \varphi_i = \varnothing),$$

где t – заданный переход, T – все множество переходов модели, α_i – формула предусловия i-го перехода, *incompl*(t) – формула неполноты для перехода t. Это позволит исключить рассмотрение параллельных независимых действий, и как следствие, ужесточить проверку полноты. Отметим, что теперь в примере 5б переходы v' и w' не будут учитываться при проверке полноты для переходов u' и t' (и наоборот):
incompl(u') $\equiv \neg((\text{state\_of\_p1} = \text{st1} \wedge x > 0) \vee (\text{state\_of\_p1} = \text{st1} \wedge x < 0)) \equiv \neg(\text{state\_of\_p1} = \text{st1}) \vee x = 0$.
incompl(v') $\equiv \neg((\text{timer} < \max) \vee (\text{timer} \geq \max)) \equiv \mathbf{0}$.

Заметим, что *incompl*(u') $\equiv$ *incompl*(t'), а так же *incompl*(v') $\equiv$ *incompl*(w').

Для решения второй проблемы предлагается ввести разбиение переходов по принципу соответствия их предусловий некоторой формуле, т. е. переход t будет рассмотрен при ограничении f если выполняется $\neg f \Rightarrow \neg\alpha_t$. Такое ограничение может задать пользователь (например, указав, что ему интересна проверка полноты при условии state_of_p1=st1), или, его можно строить автоматически, например, на основании статистических данных об использовании атрибутов в предусловиях (если поток управления задан, то он будет среди самых популярных). Далее представлены примеры переходов и формул неполноты; запись *incompl(t, f)* означает формулу неполноты для перехода t при ограничениях f.

Вернемся к рассмотрению примера 4, б. В этой модели атрибут state_of_p1 часто используется, проверяется на равенство с различными значениями, следовательно, у него большие шансы обеспечить хорошее разбиение. Пример 6 показывает возможные формулы неполноты для модели из примера 4, б.

Пример 6. Формулы неполноты

Переходы	**Формулы неполноты**
Предусловие u': state_of_p1 = st1 $\wedge$ x > 0	*incompl*(u', state_of_p1 = st1) $\equiv \neg(x > 0 \vee x < 0) \equiv x = 0$
Предусловие t': state_of_p1 = st1 $\wedge$ x < 0	*incompl*(t', x < 0) $\equiv \neg$(state_of_p1 = st1)
Предусловие v': state_of_p1 = st2 $\wedge$ x = 0	*incompl*(v', state_of_p1 = st2) $\equiv \neg(x = 0)$

Ниже представлены несколько дополнительных примеров, иллюстрирующих гибкость усовершенствованного метода.

Пример 7. Гибкость усовершенствованого метода

Переходы	**Формулы неполноты**
Предусловие T1: z = 1 $\wedge$ y = 1	*incompl*(T1, **1**) $\equiv \neg(z = 1 \wedge y = 1 \vee x = 0 \wedge a > b \wedge y = 0)$
Предусловие T2: x = 0 $\wedge$ a < b $\wedge$ y = 1	*incompl*(T2, x = 0) $\equiv \neg(a < b \wedge y = 1 \vee a > b \wedge y = 0)$
Предусловие T3: x = 0 $\wedge$ a > b $\wedge$ y = 0	*incompl*(T3, x = 0 $\wedge$ y = 0) $\equiv \neg(a > b)$
Предусловие T4: x = 0 $\wedge$ a > b $\wedge$ y = 1 $\wedge$ b < 0	*incompl*(T4, x = 0 $\wedge$ a > b) $\equiv \neg(y = 0 \vee y = 1 \wedge b < 0)$

Выводы

Описанные усовершенствования метода статической проверки полноты и непротиворечивости не чувствительны к изменениям структурного описания моделей, в частности, к отсутствию потока управления. Неполнота проверяется строже, что позволяет обнаружить потенциальные ошибки, такие как активные тупики (livelock); размер формулы неполноты уменьшен за счет введения дополнительного разбиения. Проверка непротиворечивости усовершенствована возможностью обнаружения таких патологий, как гонки в параллельных вычислениях.

Обнаруженные предложенным методом ошибки могут быть недостижимыми, однако отсутствие найденных ошибок означает их отсутствие и во множестве достижимых состояний.

1. *Колчин А.В., Летичевский А.А, Потиенко С.В. и др.* Обзор современных систем и методов верификации формальных моделей // Проблеми програмування. – 2012. – № 4. – С. 59–72.
2. *http*://www.klocwork.com
3. *Flanagan C., Freund S.* FastTrack: efficient and precise dynamic race detection // ACM SIGPLAN Notices - PLDI '09. – 2009. – Vol. 44. – P. 121–133.
4. *Voung J., Jhala R., Lerner S.* Relay: static race detection on millions of lines of code // In Proc. of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. – 2007. – P. 205–214.
5. *Летичевский А.А. (мл.).* Об одном классе базовых протоколов // Проблеми програмування. – 2005. – № 4. – С. 3–19.
6. *Потиенко С.В.* Статическая проверка требований и подходы к решению проблемы достижимости // Искусственный интеллект. – 2009. – № 1. – С. 192–197.
7. *Potiyenko S.* Static verification of basic protocols systems with unbounded number of agents // 3rd International Workshop SCSS 2010, Symbolic Computation in Software Science, Hagenberg, Austria, July 29-03. – 2010. – P. 51–54.
8. *Naik M., Aiken A., Whaley J.* Effective static race detection for Java. // Doctorial thesis, Stanford University Stanford, CA, USA. –161P. –2008.

9. *Netzer R., Miller B.* On the complexity of event ordering for shared-memory parallel program executions // Int. conf. On parallel processing. – 1990. – P. 1193–1197.
10. *Летичевский А.А., Годлевский А.Б. и др.* Свойства предикатного трансформера системы VRS // Кибернетика и системный анализ. – 2010. – № 4. – С. 3–16.

HYPERLINKHYPERLINKHYPERLINK