СОДЕРЖАНИЕ

Номер 4, 2020

CONTENTS

No. 4, 2020

ПРОГРАММИРОВАНИЕ, 2020, № 4, с. 3–13

ТЕОРИЯ ПРОГРАММИРОВАНИЯ: ФОРМАЛЬНЫЕ МОДЕЛИ И СЕМАНТИКА

УДК 519.7

ТЕСТОВЫЕ ЭКВИВАЛЕНТНОСТИ ВРЕМЕННЫХ СЕТЕЙ ПЕТРИ

© 2020 г. Е. Н. Боженкова[a,b,*], И. Б. Вирбицкайте[a,b,]**

[a] *Институт систем информатики им. А.П. Ершова СО РАН*
630090 Новосибирск, пр. ак. Лаврентьева, д. 6, Россия

[b] *Новосибирский государственный университет*
630090 Новосибирск, ул. Пирогова, д. 2, Россия

**E-mail: bozhenko@iis.nsk.su*

***E-mail: virb@iis.nsk.su*

Поступила в редакцию 10.02.2020 г.
После доработки 20.02.2020 г.
Принята к публикации 15.03.2020 г.

В данной работе определяется и исследуется семейство тестовых эквивалентностей в интерливинговой семантике, семантике частичного порядка и комбинации этих семантик в контексте непрерывно-временных безопасных сетей Петри (элементарных сетевых систем, переходы которых помечены временными интервалами и каждый переход, имеющий достаточное количество фишек во входных местах, должен срабатывать тогда, когда его счетчик достигнет некоторого значения, принадлежащего его временному интервалу). Для этого разрабатываются три представления поведения непрерывно-временной сети Петри: последовательности срабатываний сетевых переходов, представляющие семантику интерливинга, временные причинные сети-процессы, из которых выводятся частичные порядки, и временное причинное дерево, вершинами которого являются последовательности срабатываний переходов, а дуги помечены информацией о частичных порядках. Устанавливаются взаимосвязи между рассматриваемыми эквивалентностями и показывается совпадение семантик временных причинных сетей-процессов и временных причинных деревьев.

DOI: 10.31857/S0132347420040044

1. ВВЕДЕНИЕ

Тестовые эквивалентности используются при сравнении поведения систем и проверке соответствия между заданной спецификацией и полученной реализацией, а также при установлении выполнимости логических формул. Понятие тестовой эквивалентности параллельных процессов было предложено М. Хеннеси и Р. де Николой в статье [1]. Тест – это специальный процесс, который выполняется параллельно с тестируемым процессом. Такое выполнение считается успешным, если тест достигает выделенного успешного состояния, и процесс проходит тест, если каждое его совместное выполнение с процессом является успешным. Два процесса считаются тестово эквивалентными, если они проходят одни и те же наборы тестов. Чтобы облегчить исследование и применение тестовых эквивалентностей, были найдены их альтернативные характеризации – например, сравнение проводится по совокупности всех тестов, которые представляют собой вычисления процессов и множества возможных их продолжений. Концепция тестовой эквивалентности интуитивно понятна и привела к появлению математической теории эквивалентностей и предпорядков на процессах.

Изначально тестовые эквивалентности были детально исследованы в контексте моделей систем переходов (см., например, [2, 3]), которые базируются на интерливинговой семантике – отношение параллелизма между действиями системы представляется не напрямую, а посредством недетерминированного выбора между выполнениями линейно-упорядоченных поддействий. Интерливинговые тестовые эквивалентности для элементарных сетевых систем изучались в статье [4]. Чтобы преодолеть ограничения интерливингового подхода, отношение параллелизма часто моделируется как отсутствие причинной зависимости, представленной, как правило, частичным порядком, между действиями системы. В работах [5, 6] тестовые эквивалентности рассматривались в семантике частичного порядка в рамках моделей структур событий. Кроме того, тестовые эквивалентности активно изучались в контексте моделей структур событий для семантики причинных деревьев – поведение системы представляется в виде дерева, в котором дуги помечаются действиями и сведениями об их предшественниках, т.е. сохраняется

информация о причинной зависимости. Взаимосвязи между семантиками частичного порядка и причинных деревьев были хорошо изучены для моделей структур событий в работах [6–8]. Чаще всего семантика частичного порядка сетей Петри представляется посредством так называемых причинных сетей-процессов, включающих события и условия, находящиеся в отношениях причинной зависимости и параллелизма (см. [9–11] среди других статей). Сравнение разновидностей тестовой эквивалентности в частично-упорядоченной семантике сетей Петри было проведено в статье [4]. Исследование семантики причинных деревьев в контексте сетей Петри, на сколько нам известно, не проводилось.

При верификации сложных систем, критичных с точки зрения безопасности, важно исследовать не только качественные, но и количественные характеристики поведения систем. Для этих целей тестовые эквивалентности были применены в контексте ряда моделей с реальным временем. Для систем переходов с дискретным временем в работах [12] и [13] были даны альтернативные характеризации временных тестовых эквивалентностей с использованием расширенного понятия, так называемых, допустимых множеств. Семантическая теория на основе тестовых эквивалентностей была предложена для алгебр процессов с временными ограничениями в статьях [14] и [15], где формулируются альтернативные характеризации тестовых предпорядков через, так называемые, трассы отказов. Авторы статьи [15] доказали возможность дискретизации в контексте разработанной ими временной алгебры процессов и, как следствие, сведение непрерывно-временных тестовых отношений к дискретно-временным. В работе [16] интерливинговые тестовые отношения, а также результаты по их альтернативной характеризации и дискретизации распространяются на модель сетей Петри с временными характеристиками, сопоставленными фишкам, и с временными интервалами, связанными с дугами из мест в переходы. Тестовые отношения исследуются одновременно для временных и причинно-зависимых семантик моделей структур событий в статье [17]. Кроме того, в [18–20] дается классификация эквивалентностей из спектра линейного/ветвящегося времени для семантик интерливинга, причинных деревьев и частичного порядка в контексте моделей непрерывно-временных структур событий. Частично-упорядоченная семантика в работах [21, 22] была предложена для дискретно-временных сетей Петри, где с каждым переходом связана длительность его срабатывания, а также в статье [23] – для непрерывно-временных безопасных сетей Петри, где каждому переходу сопоставлен интервал временных задержек его срабатывания. Однако, насколько нам известно, в литературе по временным сетям Петри не представлены исследования тестовых эквивалентностей в семантиках причинных сетей-процессов и причинных деревьев. Только в работах [24, 25] изучались взаимосвязи трассовых и бисимуляционных эквивалентностей в интерливинговой и частично-упорядоченной семантиках непрерывно-временных безопасных сетей Петри.

Цель данной работы состоит в определении, изучении и сравнении тестовых эквивалентностей в семантиках интерливинга, причинных сетей и причинных деревьев в контексте непрерывно-временных безопасных сетей Петри (элементарных сетевых систем, переходы которых помечены временными интервалами и каждый переход, имеющий достаточное количество фишек во входных местах, должен срабатывать тогда, когда его счетчик достигнет некоторого значения, принадлежащего его временному интервалу). Устанавливаются взаимосвязи между рассматриваемыми эквивалентностями и показывается совпадение эквивалентностей в семантиках временных причинных сетей-процессов и временных причинных деревьев.

2. ВРЕМЕННЫЕ СЕТИ ПЕТРИ: СИНТАКСИС И ИНТЕРЛИВИНГОВАЯ СЕМАНТИКА

В этом разделе рассмотрим базовую терминологию непрерывно-временных сетей Петри и их интерливинговую семантику. Сначала напомним определения структуры и поведения сетей Петри. Пусть Act – множество действий.

О п р е д е л е н и е 1. (*Помеченная над* Act) *сеть Петри* (СП) – это набор $\mathcal{N} = (P, T, F, M_0, L)$, где P – конечное множество мест, T – конечное множество переходов ($P \cap T = \varnothing$ и $P \cup T \neq \varnothing$), $F \subseteq (P \times T) \cup (T \times P)$ – отношение инцидентости, $\varnothing \neq M_0 \subseteq P$ – начальная разметка, $L : T \to Act$ – помечающая функция. Для элемента $x \in P \cup T$ определим множество ${}^\bullet x = \{y \mid (y, x) \in F\}$ входных и множество $x^\bullet = \{y \mid (x, y) \in F\}$ выходных элементов, которые для подмножества $X \subseteq P \cup T$ элементов обобщаются соответственно до множеств ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$ и $X^\bullet = \bigcup_{x \in X} x^\bullet$.

Разметка M СП $\mathcal{N}$ – это произвольное подмножество P. Переход $t \in T$ *готов сработать* при разметке M, если ${}^\bullet t \subseteq M$ [1]. Обозначим через $En(M)$ множество всех переходов, готовых сработать при разметке M. Если переход t готов сработать при

[1] Для удобства последующих определений здесь не используется классическое определение: переход $t \in T$ готов сработать при разметке M, если ${}^\bullet t \subseteq M$ и $M \cap t^\bullet = \varnothing$. Второе требование будет введено в определении свойства свободы от контактов.

разметке M, то его срабатывание приводит к новой разметке M' (обозначается $M \xrightarrow{t} M'$), если $M' = (M \setminus {}^\bullet t) \cup t^\bullet$. Будем использовать обозначение $M \xrightarrow{\vartheta} M'$, если $\vartheta = t_1 \ldots t_k$ и $M = M^0 \xrightarrow{t_1} M^1 \ldots M^{k-1} \xrightarrow{t_k} M^k = M'$ ($k \geq 0$). Тогда, ϑ – это *последовательность срабатываний из M (в M')* и M' *– разметка, достижимая из разметки M*, в СП $\mathcal{N}$. Пусть $RM(\mathcal{N})$ – множество всех разметок, достижимых из M_0, в СП $\mathcal{N}$.

СП $\mathcal{N}$ называется *T-ограниченной*, если ${}^\bullet t \neq \varnothing \neq t^\bullet$ для всех переходов $t \in T$; *свободной от контактов*, если для произвольной разметки $M \in RM(\mathcal{N})$ и любого перехода t, готового сработать при разметке M, выполняется условие $M \cap t^\bullet = \varnothing$.

Под непрерывно-временной сетью Петри (ВСП) [23] понимается СП, в которой с каждым переходом связан временной интервал, указывающий возможные временные моменты срабатывания перехода, готового по наличию фишек в его входных местах; готовый переход может сработать, только когда достигнута нижняя граница и не превышена верхняя граница его интервала, и, если он еще не сработал, то обязан сработать, когда достигнута верхняя граница его интервала.

Область $\mathbb{T}$ временных значений – множество неотрицательных рациональных чисел. Считаем, что $[\tau_1, \tau_2]$ – замкнутый интервал между двумя временными значениями $\tau_1, \tau_2 \in \mathbb{T}$. Также, бесконечность может появляться как правая граница в открытых справа интервалах. Пусть *Interv* – множество всех таких интервалов.

Определение 2. (*Помеченная над Act*) *временная сеть Петри* (ВСП) – это пара $\mathcal{TN} = (\mathcal{N}, D)$, где $\mathcal{N}$ – (помеченная над *Act*) базовая сеть Петри и $D : T \to Interv$ – статическая временная функция, сопоставляющая каждому переходу временной интервал. Границы временного интервала $D(t) \in Interv$ называются ранним (*Eft*) и поздним (*Lft*) временами срабатывания перехода $t \in T$.

Состояние ВСП $\mathcal{TN}$ – это пара $S = (M, I)$, где M – разметка СП $\mathcal{N}$ и $I : En(M) \to \mathbb{T}$ – динамическая временная функция. Начальное состояние ВСП $\mathcal{TN}$ – это пара $S_0 = (M_0, I_0)$, где M_0 – начальная разметка СП $\mathcal{N}$ и $I_0(t) = 0$ для всех $t \in En(M_0)$. Переход t, готовый сработать при разметке M в СП $\mathcal{N}$, *готов сработать в состоянии $S = (M, I)$ в относительный момент времени* $\theta \in \mathbb{T}$ в ВСП $\mathcal{TN}$, если ($Eft(t) \leq I(t) + \theta$) и верно, что ($I(t') + \theta \leq Lft(t')$ для всех $t' \in En(M)$). Если переход t готов сработать в состоянии $S = (M, I)$ в относительный момент времени θ, то его срабатывание приводит в новое состояние $S' = (M', I')$ (обозначается $S \xrightarrow{(t,\theta)} S'$) такое, что верно: $M \xrightarrow{t} M'$ и $\forall t' \in T_\diamond$

$$I'(t') = \begin{cases} I(t') + \theta, & \text{если } t' \in En(M \setminus {}^\bullet t), \\ 0, & \text{если } t' \in En(M') \setminus En(M \setminus {}^\bullet t), \\ \text{не определено иначе.} \end{cases}$$

Будем писать $S \xrightarrow{\sigma} S'$, если $\sigma = (t_1, \theta_1) \ldots (t_k, \theta_k)$ и $S = S^0 \xrightarrow{(t_1,\theta_1)} S^1 \ldots S^{k-1} \xrightarrow{(t_k,\theta_k)} S^k = S'$ ($k \geq 0$). Тогда, σ – *последовательность срабатываний из S (в S')* и S' *– состояние*, *достижимое из S*, в ВСП $\mathcal{TN}$. Пусть $\mathcal{FS}(\mathcal{TN})$ – множество всех последовательностей срабатываний из S_0 и $RS(\mathcal{TN})$ – множество всех состояний, достижимых из S_0, в ВСП $\mathcal{TN}$. Для $\sigma = (t_1, \theta_1) \ldots (t_k, \theta_k) \in \mathcal{FS}(\mathcal{TN})$ $L(\sigma) = (a_1, \theta_1) \ldots (a_k, \theta_k)$, если $a_i = L(t_i)$ для всех $1 \leq i \leq k$. Определим *интерливинговый язык ВСП* $\mathcal{TN}$ следующим образом: $\mathcal{L}(\mathcal{TN}) = \{L(\sigma) \in (Act \times \mathbb{T})^* \mid \sigma \in \mathcal{FS}(\mathcal{TN})\}$.

ВСП $\mathcal{TN}$ называется *T-ограниченной*, если базовая СП T-ограничена; *свободной от контактов*, если для любого состояния $S = (M, I) \in RS(\mathcal{TN})$ и любого перехода t, готового сработать в состоянии S в относительный момент времени θ, верно, что $(M \setminus {}^\bullet t) \cap t^\bullet = \varnothing$[2]; *прогрессирующей по времени*, если для любой последовательности переходов $\{t_1, t_2, \ldots, t_n\} \subseteq T$ такой, что $t_i^\bullet \cap {}^\bullet t_{i+1} \neq \varnothing$ ($1 \leq i < n$) и $t_n^\bullet \cap {}^\bullet t_1 \neq \varnothing$, выполняется неравенство $\sum_{1 \leq i \leq n} Eft(t_i) > 0$[3]. В дальнейшем будем рассматривать только T-ограниченные, свободные от контактов и прогрессирующие по времени ВСП.

Пример 1. Пример помеченной над $Act = \{a, b, c, d\}$ ВСП $\mathcal{TN}$ показан на рис. 1, где места представлены окружностями, переходы – барьерами; рядом с элементами ВСП размещены их имена; между элементами, включенными в отношение инцидентности, изображены стрелки; каждое место, входящее в начальную разметку, отмечено наличием в нем фишки (жирной точки); значения помечающей и статической временной функций указаны рядом с переходами. Нетрудно проверить, что переходы t_1 и t_3 готовы сработать при начальной разметке $M_0 = \{p_1, p_2\}$ и, более того, готовы сработать в начальном состоянии $S_0 = (M_0, I_0)$, где $I_0(t) = \begin{cases} 0, & \text{если } t \in \{t_1, t_3\}, \\ \text{не определено иначе,} \end{cases}$ в относительный момент времени $\theta \in [2, 3]$. При этом, $\sigma = (t_1, 3)\ (t_3, 0)\ (t_2, 2)\ (t_3, 2)\ (t_1, 0)\ (t_5, 2)\ (t_4, 0)$ –

[2] Заметим, что если базовая СП $\mathbb{N}$ свободна от контактов, то и ВСП $\mathcal{TN}$ свободна от контактов, но обратное неверно.

[3] Свойство прогрессирования по времени гарантирует корректность измененного определения совойства свободы от контактов.

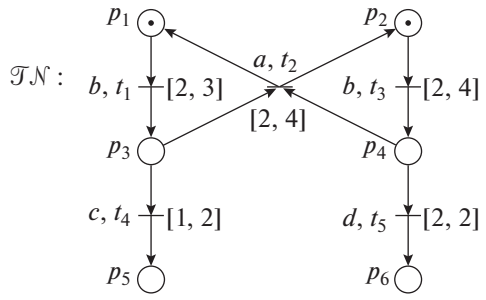


Рис. 1. Пример временной сети Петри.

последовательность срабатываний из S_0 в ВСП $\mathcal{TN}$. Кроме того, $\mathcal{TN}$ является T-ограниченной, свободной от контактов и прогрессирующей по времени. □

3. ПРИЧИННО-ЗАВИСИМЫЕ СЕМАНТИКИ ВРЕМЕННЫХ СЕТЕЙ ПЕТРИ

3.1. Базовые определения

Сначала рассмотрим определения, связанные с временными сетями.

О п р е д е л е н и е 3. (*Помеченной над Act*) *временной сетью* называется конечная, ациклическая сеть $TN = (B, E, G, l, \tau)$, где B – множество условий, E – множество событий, $G \subseteq (B \times E) \cup (E \times B)$ – отношение инцидентности такое, что $\{e | (e,b) \in G\} = \{e | (b,e) \in G\} = E$, $l : E \to Act$ – помечающая функция и $\tau : E \to \mathbb{T}$ – временная функция такая, что $eG^+e' \Rightarrow \tau(e) \le \tau(e')$.

Введем дополнительные обозначения для временной сети $TN = (B, E, G, l, \tau)$. Пусть $\prec = G^+$, $\preceq = G^*$ и $\tau(TN) = \max\{\tau(e) | e \in E\}$. Определим множества: ${}^\bullet x = \{y | (y,x) \in G\}$ и $x^\bullet = \{y | (x,y) \in G\}$ для $x \in B \cup E$; ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$ и $X^\bullet = \bigcup_{x \in X} x^\bullet$ для $X \subseteq B \cup E$; ${}^\bullet TN = \{b \in B | {}^\bullet b = \varnothing\}$ и $TN^\bullet = \{b \in B | b^\bullet = \varnothing\}$.

$TN = (B, E, G, l, \tau)$ называется (*помеченной над Act*) *временной причинной сетью*, если $|{}^\bullet b| \le 1$ и $|b^\bullet| \le 1$ для всех условий $b \in B$. Заметим, что $\eta(TN) = (E_{TN}, \preceq_{TN} \cap (E_{TN} \times E_{TN}), l_{TN}, \tau_{TN})$ является (*помеченным над Act*) *временным частично-упорядоченным множеством* (*ВЧУМ*)[4].

Введем дополнительные определения и обозначения для временной причинной сети $TN = (B, E, G, l, \tau)$:

• $\downarrow e = \{x | x \preceq e\}$ – множество *предшественников события* $e \in E$, $Earlier(e) = \{e' \in E | \tau(e') < \tau(e)\}$ – множество *временных предшественников события* $e \in E$;

• $E' \subseteq E$ – *левозамкнутое* подмножество E, если $\downarrow e' \cap E \subseteq E'$ для каждого $e' \in E'$. Для такого подмножества будем использовать обозначение $Cut(E') = (E'^\bullet \cup {}^\bullet TN) \setminus {}^\bullet E'$. $E' \subseteq E$ – *непротиворечивое по времени* подмножество E, если $\tau(e') \le \tau(e)$ для всех $e' \in E'$ и $e \in E \setminus E'$;

• последовательность $\rho = e_1 \ldots e_k$ $(k \ge 0)$ событий из E – *линеаризация временной причинной сети* TN, если каждое событие из E встречается в последовательности только один раз и выполняется следующее условие: $(e_i \prec e_j \vee \tau(e_i) < \tau(e_j)) \Rightarrow i < j$ для всех $1 \le i, j \le k$. Определим множество $E^l_\rho = \bigcup_{1 \le i \le l} e_i$ $(0 \le l \le k)$. Очевидно, что E^l_ρ являются левозамкнутыми и непротиворечивыми по времени подмножествами E и, кроме того, $\tau(e_k) = \tau(TN)$.

Из определений временной причинной сети и ее линеаризации получаем справедливость следующей

Л е м м а 1. *Любая временная причинная сеть имеет линеаризацию.*

Временные причинные сети $TN = (B, E, G, l, \tau)$ и $TN' = (B', E', G', l', \tau')$ *изоморфны* (обозначается $TN \simeq TN'$), если существует биективное отображение β: $B \cup E \to B' \cup E'$ такое, что: (а) $\beta(B) = B'$ и $\beta(E) = E'$; (б) $xGy \Leftrightarrow \beta(x) G' \beta(y)$ для всех x, $y \in B \cup E$; (в) $l(e) = l'(\beta(e))$ и $\tau(e) = \tau'(\beta(e))$ для всех $e \in E$. Кроме того, будем говорить, что TN является префиксом TN' (обозначается $TN \to TN'$), если $B \subseteq B'$, E – левозамкнутое и непротиворечивое по времени подмножество E', $E' \setminus E = \{e\}$, $G = G' \cap (B \times E \cup E \times B)$, $l = l'|_E$ и $\tau = \tau'|_E$.

П р и м е р 2. На рис. 2 показана временная причинная сеть $TN = (B, E, G, l, \tau)$, где условия представлены окружностями, а события – барьерами; рядом с элементами сети размещены их имена; между элементами, включенными в отношение инцидентности, изображены стрелки; значения функций l и τ указаны рядом с событиями. Определим временные причинные сети $TN' = (B', E', G', l', \tau')$, где $B' = \{b_1, b_2, b_3, b_4\}$, $E' = \{e_1, e_3\}$, $G' = G \cap (B' \times E' \cup E' \times B')\}$, $l' = l|_{E'}$, $\tau' = \tau|_{E'}$, и $TN'' = (B'', E'', G'', l'', \tau'')$, где $B'' = \{b_1, b_2, b_3\}$, $E'' = \{e_1\}$, $G'' = G \cap (B'' \times E'' \cup E'' \times B'')\}$, $l'' = l|_{E''}$, $\tau'' = \tau|_{E''}$. Легко проверить, что TN'' является префиксом TN'. □

[4] (Помеченный над *Act*) ВЧУМ – это набор $\eta = (X, \preceq, \lambda, \tau)$, состоящий из конечного множества элементов X; рефлексивного, антисимметричного и транзитивного отношения $\preceq$; помечающей функции λ: $X \to Act$ и временной функции τ: $X \to \mathbb{T}$ такой, что $e \preceq e' \Rightarrow \tau(e) \le \tau(e')$. Пусть $\tau(\eta) = \max\{\tau(x) | x \in X\}$.

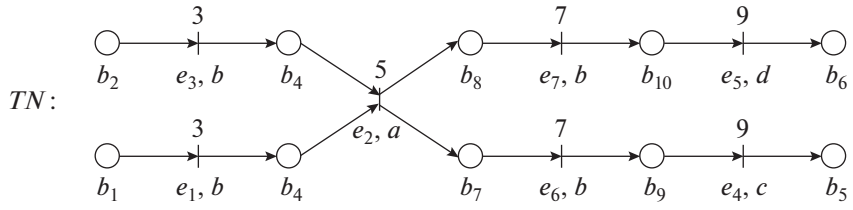


Рис. 2. Пример временной причинной сети.

3.2. Временные причинные сети-процессы временных сетей Петри

В этом разделе рассмотрим понятие временных причинных сетей-процессов ВСП, предложенное в статье [23].

О п р е д е л е н и е 4. Пусть $\mathcal{TN} = ((P, T, F, M_0, L), D)$ – ВСП и $TN = (B, E, G, l, \tau)$ – временная причинная сеть. Отображение φ: $B \cup E \to P \cup T$ называется *гомоморфизмом из TN* в $\mathcal{TN}$, если выполняются следующие условия:

- $\varphi(B) \subseteq P$, $\varphi(E) \subseteq T$;
- ограничение φ на ${}^\bullet e$ является биекцией между ${}^\bullet e$ и ${}^\bullet\varphi(e)$ и ограничение φ на $e^\bullet$ является биекцией между $e^\bullet$ и $\varphi(e)^\bullet$ для всех $e \in E$;
- ограничение φ на ${}^\bullet TN$ является биекцией между ${}^\bullet TN$ и M_0;
- $l(e) = L(\varphi(e))$ для всех $e \in E$.

Пара $\pi = (TN, \varphi)$ называется *временным причинным сетью-процессом ВСП* $\mathcal{TN}$, если TN – временная причинная сеть и φ – гомоморфизм из TN в $\mathcal{TN}$.

Пусть $\pi = (TN, \varphi)$ – временной причинный сеть-процесс ВСП $\mathcal{TN}$, $B' \subseteq B_{TN}$ и $t \in En(\varphi(B'))$. Тогда глобальный момент времени, когда фишки появляются во всех входных местах перехода t, определяется следующим образом: $\mathbf{TOE}_\pi(B', t) = \max(\{\tau_{TN}({}^\bullet b) | b \in B'_{[t]} \setminus {}^\bullet TN\} \cup \{0\})$, где $B'_{[t]} = \{b \in B' | \varphi_{TN}(b) \in {}^\bullet t\}$.

Для того, чтобы значения временных функций временных причинных сетей-процессов ВСП соответствовали временным интервалам срабатывания сетевых переходов, вводится понятие корректных временных причинных сетей-процессов ВСП.

О п р е д е л е н и е 5. Временной причинный сеть-процесс $\pi = (TN, \varphi)$ ВСП $\mathcal{TN}$ называется *корректным*, если для каждого $e \in E$ выполняются следующие условия:

- $\tau(e) \geq \mathbf{TOE}_\pi({}^\bullet e, \varphi(e)) + Eft(\varphi(e))$,
- $\forall t \in En(\varphi(C_e)) \diamond \tau(e) \leq \mathbf{TOE}_\pi(C_e, t) + Lft(t)$, где $C_e = Cut(Earlier(e))$.

Пусть $\mathcal{CP}(\mathcal{TN})$ – множество корректных временных причинных сетей-процессов ВСП $\mathcal{TN}$. Через $\mathcal{TP}os(\mathcal{TN}) = \{TP | \exists \pi = (TN, \varphi) \in \mathcal{CP}(\mathcal{TN}) : TP \simeq^5 \simeq \eta(TN)\}$ обозначим множество ВЧУМов, изоморфных ВЧУМам, полученным из корректных временных причинных сетей-процессов ВСП $\mathcal{TN}$.

П р и м е р 3. Определим отображение φ из временной причинной сети TN (см. рис. 2) в ВСП $\mathcal{TN}$ (см. рис. 1) следующим образом: $\varphi(b_i) = p_i$ $(1 \leq i \leq 6)$, $\varphi(b_i) = p_{i-6}$ $(7 \leq i \leq 10)$ и $\varphi(e_i) = t_i$ $(1 \leq i \leq 5)$, $\varphi(e_6) = t_1$, $\varphi(e_7) = t_3$. Далее, для временной причинной сети TN', заданной в примере 2, определим $\varphi' = \varphi|_{E' \cup B'}$. Легко видеть, что $\pi = (TN, \varphi)$ и $\pi' = (TN', \varphi')$ являются временными причинными сетями-процессами ВСП $\mathcal{TN}$.

Для множества $\tilde{B} = \{b_3, b_4\} \subset B$ и перехода $t_2 \in En(\varphi(\tilde{B}))$ вычислим $\mathbf{TOE}_\pi(\tilde{B}, t_2) = \max(\{\tau_{TN}({}^\bullet b) | b \in \tilde{B}_{[t_2]} \setminus {}^\bullet TN\} \cup \{0\}) = \max(\{\tau(e_1) = 3, \tau(e_3) = 3\} \cup \{0\}) = 3$. Также, нетрудно проверить, что временные причинные сети-процессы $\pi = (TN, \varphi)$ и $\pi' = (TN', \varphi')$ являются корректными. □

Будем говорить, что $\pi = (TN, \varphi)$ и $\pi' = (TN', \varphi')$ из $\mathcal{CP}(\mathcal{TN})$ *изоморфны* (обозначается $\pi \simeq \pi'$), если существует изоморфизм f: $TN \simeq TN'$ такой, что $\varphi(x) = \varphi'(f(x))$ для всех $x \in B \cup E$; а также будем писать $\pi \to \pi'$ в $\mathcal{TN}$, если $TN \to TN'$ и $\varphi = \varphi'|_{B \cup E}$.

Рассмотрим взаимосвязи между последовательностями срабатываний и корректными временными причинными сетями-процессами ВСП. Для $\pi = (TN, \varphi) \in \mathcal{CP}(\mathcal{TN})$ определим функцию FS_π, которая отображает линеаризацию $\rho = e_1 \ldots e_k$ TN в последовательность вида: $FS_\pi(\rho) = (\varphi(e_1), \tau(e_1) - 0) \ldots (\varphi(e_k), \tau(e_k) - \tau(e_{k-1}))$.

У т в е р ж д е н и е 1. *Пусть $\mathcal{TN}$ – ВСП. Тогда*

[5] Два ВЧУМ $\eta = (X, \preceq, \lambda, \tau)$ и $\eta' = (X', \preceq', \lambda', \tau')$ изоморфны (обозначается $\eta \simeq \eta'$), если существует биекция β: $X \to X'$ такая, что (а) $x \preceq y \Leftrightarrow \beta(x) \preceq' \beta(y)$ для всех $x, y \in X$; (б) $\lambda(x) = \lambda'(\beta(x))$ и $\tau(x) = \tau'(\beta(x))$ для всех $x \in X$.

(а) *если* $\pi = (TN, \varphi) \in \mathcal{CP}(\mathcal{TN})$ *и* ρ *– линеаризация TN, то существует единственная последовательность срабатываний* $FS_\pi(\rho) \in \mathcal{FS}(\mathcal{TN})$;

(б) *если* $\sigma \in \mathcal{FS}(\mathcal{TN})$, *то существует единственный* (*с точностью до изоморфизма*) *временной причинный сеть-процесс* $\pi_\sigma = (TN, \varphi) \in \mathcal{CP}(\mathcal{TN})$ *и единственная линеаризация* ρ_σ *TN такие, что* $FS_{\pi_\sigma}(\rho_\sigma) = \sigma$.

Доказательство. Пункт (*а*) без факта единственности последовательности срабатываний $FS_\pi(\rho)$ и пункт (*б*) без факта единственности линеаризации ρ_σ – это переформулировки результатов, доказанных в теоремах соответственно 19 и 21, 22 в [23].

(*а*) Единственность последовательности срабатываний $FS_\pi(\rho)$ следует из определений гомоморфизма φ и функции FS_π.

(*б*) Пусть $\rho_\sigma = e_1 \ldots e_n$ $(n \geq 0)$ – линеаризация TN такая, что $FS_{\pi_\sigma}(\rho_\sigma) = \sigma = (t_1, \theta_1)\ldots(t_n, \theta_n) \in \mathcal{FS}(\mathcal{TN})$. Предположим обратное, т.е. существует линеаризация $\overline{\rho} = \overline{e}_1 \ldots \overline{e}_n$ TN такая, что $FS_{\pi_\sigma}(\overline{\rho}) = \sigma$ и $\overline{\rho} \neq \rho_\sigma$. Так как все линеаризации TN конечны, то можно найти минимальное k такое, что $e_k \neq \overline{e}_k$. Ясно, что $\varphi(e_k) = \varphi(\overline{e}_k) = t_k$. Поскольку $\mathcal{TN}$ – T-ограниченная ВСП, то $^\bullet t_k \neq \varnothing$. Возьмем произвольное место $p_k \in {}^\bullet t_k$. По определению гомоморфизма, существуют условия $b \in {}^\bullet e_k$ и $\overline{b} \in {}^\bullet \overline{e}_k$ такие, что $\varphi(b) = \varphi(\overline{b}) = p_k$. В силу определения временной причинной сети, верно, что $b \neq \overline{b}$.

Рассмотрим возможные случаи.

– $\{b, \overline{b}\} \subseteq {}^\bullet TN$. Тогда верно, что $p_k \in M_0$. Это противоречит определению гомоморфизма φ.

– $b \in {}^\bullet TN$ и $\overline{b} \notin {}^\bullet TN$ (случай, когда $\overline{b} \in {}^\bullet TN$ и $b \notin {}^\bullet TN$, аналогичен). Поскольку $b \in {}^\bullet TN$, то получаем, что $p_k \in M_0$, по определению гомоморфизма φ. Тогда имеем, что $b = b_{0, p_k}$, по построению π_σ в [23]. Предполагая, что $\overline{b} \notin {}^\bullet TN$, найдем событие $\tilde{e}$ такое, что $\{\tilde{e}\} = {}^\bullet \overline{b}$. Так как k – минимальное, то в обеих линеаризациях ρ и $\overline{\rho}$ событие $\tilde{e}$ имеет один и тот же порядковый номер, т.е. $\tilde{e} = e_i = \overline{e}_i$ для некоторого $0 < i < k$. По определению функции FS_{π_σ}, верно, что $\varphi(\tilde{e}) = t_i$. Тогда $p_k \in t_i^\bullet$, согласно определению φ. Кроме того, имеем, что $\overline{b} = b_{i, p_k}$, в силу построения π_σ в [23]. Таким образом, получили противоречие со свойством (41) из [23]: не существует b_{i, p_k} для любого $0 < i < k$.

– $b, \overline{b} \notin {}^\bullet TN$. Следовательно, существует событие $\tilde{e}$ $(\hat{e})$ такое, что $\{\tilde{e}\} = {}^\bullet b$ $(\{\hat{e}\} = {}^\bullet \overline{b})$. В силу определения гомоморфизма φ, верно, что $\tilde{e} \neq \hat{e}$. Так как k – минимальное, то в обеих линеаризациях ρ и $\overline{\rho}$ событие $\tilde{e}$ $(\hat{e})$ имеет один и тот же порядковый номер, т.е. $\tilde{e} = e_i = \overline{e}_i$ для некоторого $1 \leq i < k$ $(\hat{e} = e_j = \overline{e}_j$ для некоторого $1 \leq j < k)$. Тогда $i \neq j$, согласно определению линеаризации. По определению функции FS_{π_σ}, имеем, что $\varphi(\tilde{e}) = t_i$ $(\varphi(\hat{e}) = t_j)$. Из определения гомоморфизма следует, что $p_k \in t_i^\bullet$ $(p_k \in t_j^\bullet)$. По построению π_σ в [23] верно, что $b = b_{i, p_k}$ $(\overline{b} = b_{j, p_k})$. В случае, когда $i < j < k$ $(j < i < k)$, получаем противоречие со свойством (41) из [23]: не существует b_{l, p_k} для любого $i < l < k$ $(j < l < k)$. □

Пример 4. Для временного причинного сети-процесса $\pi = (TN, \varphi)$ ВСП $\mathcal{TN}$ (см. пример 3) и линеаризации $\rho = e_1 e_3 e_2 e_7 e_6 e_5 e_4$ временной причинной сети TN получаем, что $FS_\pi(\rho) = (t_1, 3)\ (t_3, 0)\ (t_2, 2)\ (t_3, 2)\ (t_1, 0)\ (t_5, 2)\ (t_4, 0)$ является последовательностью срабатываний ВСП $\mathcal{TN}$ (см. пример 1). □

Используя определение префикса временной причинной сети и утверждение 1, легко показать, что если последовательность срабатываний и временной причинный сеть-процесс ВСП взаимосвязаны, то их непосредственные расширения тоже взимосвязаны.

Лемма 2. *Пусть* $\sigma \in \mathcal{FS}(\mathcal{TN})$ *и* $\pi \in \mathcal{CP}(\mathcal{TN})$ *такие, что* $\sigma = FS_\pi(\rho)$, *где* ρ *– линеаризация* TN_π. *Тогда*

(а) *если* $\sigma(t, \theta) \in \mathcal{FS}(\mathcal{TN})$, *то существует* $\tilde{\pi} \in \mathcal{CP}(\mathcal{TN})$ *такой, что* $\pi \to \tilde{\pi}$ *в* $\mathcal{TN}$ *и* $\sigma(t, \theta) = FS_{\tilde{\pi}}(\rho e)$, *где* ρe *– линеаризация* $TN_{\tilde{\pi}}$;

(б) *если* $\pi \to \tilde{\pi}$ *в* $\mathcal{TN}$, *то существует* $\sigma(t, \theta) \in \mathcal{FS}(\mathcal{TN})$ *такая, что* $\sigma(t, \theta) = FS_{\tilde{\pi}}(\rho e)$, *где* ρe *– линеаризация* $TN_{\tilde{\pi}}$.

3.3. Временные причинные деревья временных сетей Петри

Причинные деревья [8] – это деревья синхронизации, у которых в пометках дуг кроме имен действий содержится дополнительная информация о предшественниках этих действий, что обеспечивает интерливинговое представление параллельных процессов с описанием причинной зависимости между их действиями. Добавляя времена выполнения действий в пометки причинных деревьев, получаем временные причинные деревья. Во временном причинном дереве ВСП $\mathcal{TN}$ вершинами являются последовательности срабатываний из множества $\mathcal{FS}(\mathcal{TN})$ и дуги проводятся между двумя вершинами, если одна последовательность является непосредственным расширением другой. Информация о предшественниках для пометок дуг получается из отношений инци-

дентности соответствующих временных причинных сетей-процессов ВСП $\mathcal{TN}$.

Определение 6. *Временное причинное дерево ВСП* $\mathcal{TN}$, $TCT(\mathcal{TN})$, – это дерево $(\mathcal{FS}(\mathcal{TN}), A, \phi)$, где $\mathcal{FS}(\mathcal{TN})$ – множество вершин с корнем ε; $A = \{(\sigma, \sigma(t,\theta)) | \sigma, \sigma(t,\theta) \in \mathcal{FS}(\mathcal{TN})\}$ – множество дуг; ϕ – помечающая функция такая, что $\phi(\epsilon) = \epsilon$ и $\phi(\sigma, \sigma(t,\theta)) = (l_{\mathcal{TN}}(t), \theta, K)$, где $K = \{n - l + 1 \mid \sigma(t,\theta) = FS_{\pi_{\sigma(t,\theta)}}(e_1 \ldots e_n e)$, где $e_1 \ldots e_n e$ – линеаризация $TN_{\pi_{\sigma(t,\theta)}}$, и $e_l \prec_{TN_{\pi_{\sigma(t,\theta)}}} e\}$. Пусть $path(\sigma)$ – путь в $TCT(\mathcal{TN})$ из корня в вершину σ[6]. Через $\mathcal{L}(TCT(\mathcal{TN})) = \{\phi(path(\sigma)) \in (Act \times \mathbb{T} \times 2^{\mathbb{N}})^* | \sigma \in \mathcal{FS}(\mathcal{TN})\}$ обозначим множество последовательностей пометок путей временного причинного дерева ВСП $\mathcal{TN}$.

Пример 5. Рассмотрим ВСП $\mathcal{TN}$ (см. рис. 1) и последовательность срабатываний $\sigma = (t_1, 3)$ $(t_3, 0)$ $(t_2, 2)$ $(t_3, 2)$ $(t_1, 0)$ $(t_5, 2)$ $(t_4, 0) \in \mathcal{FS}(\mathcal{TN})$. Получаем, что последовательность пометок пути из корня в вершину σ имеет вид: $\phi(path(\sigma)) = (a, 3, \varnothing)$ $(b, 0, \varnothing)$ $(a, 2, \{1, 2\})$ $(b, 2, \{1, 2, 3\})$ $(a, 0, \{2, 3, 4\})$ $(d, 2, \{2, 3, 4, 5\})$ $(c, 0, \{2, 4, 5, 6\})$. □

Установим взаимосвязи между корректными временными причинными сетями-процессами и помеченными путями во временных причинных деревьях двух ВСП.

Утверждение 2. *Пусть* $\mathcal{TN}$ *и* $\mathcal{TN}'$ *– ВСП и* $TCT(\mathcal{TN}) = (\mathcal{FS}(\mathcal{TN}), A, \phi)$ *и* $TCT(\mathcal{TN}') = (\mathcal{FS}(\mathcal{TN}'), A', \phi')$ *– их временные причинные деревья. Тогда*

(а) *если* $\pi \in \mathcal{CP}(\mathcal{TN})$ *и* $\pi' \in \mathcal{CP}(\mathcal{TN}')$ *– временные сети-процессы и* $f: \eta(TN_\pi) \to \eta(TN_{\pi'})$ *– изоморфизм, то* $\phi(path(FS_\pi(\rho))) = \phi'(path(FS_{\pi'}(f(\rho))))$ *для любой линеаризации* ρ TN_π;

(б) *если* $\phi(path(\sigma)) = \phi'(path(\sigma'))$ *для* $\sigma \in \mathcal{FS}(\mathcal{TN}')$ *и* $\sigma' \in \mathcal{FS}(\mathcal{TN})$, *то существует изоморфизм* $f: \eta(TN_{\pi_\sigma}) \to \eta(TN_{\pi_{\sigma'}})$ *такой, что* $f(\rho_\sigma) = \rho_{\sigma'}$.

Доказательство. (*а*) Следует из утверждения 1(*а*) и свойств изоморфизма *f*.

(*б*) Следует из утверждения 1(*б*), определения 6 и свойств гомоморфизма φ и функции *FS*. □

Рассмотрим и докажем вспомогательный полезный факт.

Утверждение 3. Пусть $\mathcal{TN}$ и $\mathcal{TN}'$ – ВСП. Тогда $\mathcal{L}(\mathcal{TN}) = \mathcal{L}(\mathcal{TN}') \Leftarrow \mathcal{L}(TCT(\mathcal{TN})) = \mathcal{L}(TCT(\mathcal{TN}')) \Leftrightarrow \mathcal{TP}os(\mathcal{TN}) = \mathcal{TP}os(\mathcal{TN}')$.

[6] Мы определяем $path(\epsilon) = \epsilon$. Заметим, что в $TCT(\mathcal{TN})$ для любой вершины $\sigma \in \mathcal{FS}(\mathcal{TN})$ существует путь из корня в вершину σ.

Доказательство. Факт, что $\mathcal{L}(\mathcal{TN}) = \mathcal{L}(\mathcal{TN}') \Leftarrow \mathcal{L}(TCT(\mathcal{TN})) = \mathcal{L}(TCT(\mathcal{TN}'))$, непосредственно следует из определений.

Теперь проверим, что $\mathcal{L}(TCT(\mathcal{TN})) = \mathcal{L}(TCT(\mathcal{TN}')) \Rightarrow \mathcal{TP}os(\mathcal{TN}') = \mathcal{TP}os(\mathcal{TN}_2)$. Возьмем произвольное ВЧУМ $TP \in \mathcal{TP}os(\mathcal{TN})$. Это означает, что можно найти временной причинный сеть-процесс $\pi = (TN, \varphi) \in \mathcal{CP}(\mathcal{TN})$ такой, что $\eta(TN) \simeq TP$. Рассмотрим произвольную линеаризацию ρ *TN*. Согласно лемме 1, хотя бы одна линеаризация *TN* существует. Из утверждения 1(*а*) следует, что найдется последовательность срабатываний $\sigma = FS_\pi(\rho) \in \mathcal{FS}(\mathcal{TN})$. Согласно утверждению 1(*б*), можем без потери общности считать, что $\pi = \pi_\sigma$ и $\rho = \rho_\sigma$. По определению, в $TCT(\mathcal{TN})$ существует путь *u* из корня в вершину σ. Кроме того, верно, что $\phi(u) \in \mathcal{L}(TCT(\mathcal{TN})) = \mathcal{L}(TCT(\mathcal{TN}'))$. Это означает наличие в $TCT(\mathcal{TN}')$ пути *u*' из корня в вершину $\sigma' \in \mathcal{FS}(\mathcal{TN}')$ такого, что $\phi(u') = \phi(u)$. В силу утверждения 1(*б*), существуют единственный (с точностью до изоморфизма) временной причинный сеть-процесс $\pi_{\sigma'} = (TN_{\sigma'}, \varphi_{\sigma'}) \in \mathcal{CP}(\mathcal{TN}')$ и единственная линеаризация $\rho_{\sigma'}$ $TN_{\sigma'}$ такие, что $FS_{\pi_{\sigma'}}(\rho_{\sigma'}) = \sigma'$. Из утверждения 2(*б*) следует, что найдется изоморфизм $f: \eta(TN_\sigma) \to \eta(TN_{\sigma'})$ такой, что $f(\rho_\sigma) = \rho_{\sigma'}$. Таким образом, получаем, что $\eta(TN_{\sigma'}) \simeq TP$, т.е. $TP \in \mathcal{TP}os(\mathcal{TN}')$.

И наконец, проверим, что $\mathcal{TP}os(\mathcal{TN}) = \mathcal{TP}os(\mathcal{TN}') \Rightarrow \mathcal{L}(TCT(\mathcal{TN})) = \mathcal{L}(TCT(\mathcal{TN}'))$. Возьмем произвольное $w \in \mathcal{L}(TCT(\mathcal{TN}))$. Это означает, что существует путь *u* в $TCT(\mathcal{TN})$ из корня в вершину $\sigma \in \mathcal{FS}(\mathcal{TN})$ такой, что $\phi(u) = w$. Согласно утверждению 1(*б*), можно найти единственный (с точностью до изморфизма) временной причинный сеть-процесс $\pi_\sigma = (TN_\sigma, \varphi_\sigma) \in \mathcal{CP}(\mathcal{TN})$ и единственную линеаризацию ρ_σ TN_σ такие, что $FS_{\pi_\sigma}(\rho_\sigma) = \sigma$. Значит, верно, что $\eta(TN_\sigma) \in \mathcal{TP}os(\mathcal{TN}) = \mathcal{TP}os(\mathcal{TN}')$. Тогда существует временной причинный сеть-процесс $\pi' = (TN', \varphi') \in \mathcal{CP}(\mathcal{TN}')$ такой, что $\eta(TN_\sigma) \simeq \eta(TN')$. Следовательно, найдется изоморфизм $f: \eta(TN_\sigma) \to \eta(TN')$. Применяя утверждение 2(*а*), получаем, что $w = \phi(path(FS_{\pi_\sigma}(\rho_\sigma))) = \phi'(path(FS_{\pi'}(f(\rho_\sigma)))) \in \mathcal{L}(TCT(\mathcal{TN}'))$. □

4. ТЕСТОВЫЕ ЭКВИВАЛЕНТНОСТИ

При интерливинговом подходе к определению тестовой эквивалентности в качестве тестов рассматриваются последовательности *w* выполняемых действий (вычисления системы) и множества *W* возможных дальнейших действий. Процесс проходит тест, если после выполнения

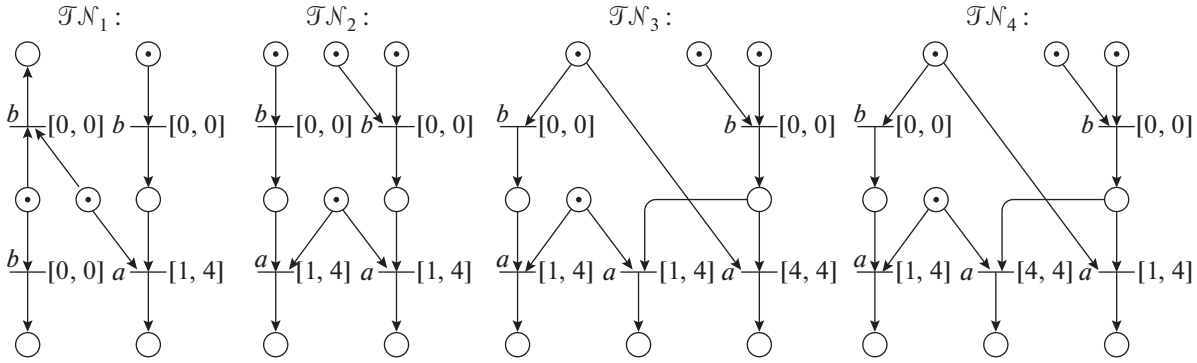


Рис. 3.

каждой последовательности w действий дальше может выполниться хотя бы одно действие из W. Два процесса тестово эквивалентны, если они проходят одно и то же множество тестов. Во временном варианте добавляется информация о временах выполнения действий.

Определение 7. Пусть $\mathcal{TN}$ и $\mathcal{TN}'$ – ВСП.

Для последовательности $w \in (Act \times \mathbb{T})^*$ и множества $W \subseteq (Act \times \mathbb{T})$, $\mathcal{TN}$ **after** w $\mathbf{MUST}_{int}$ W, если для всех $\sigma \in \mathcal{FS}(\mathcal{TN})$ таких, что $L(\sigma) = w$, существуют $(a, \theta) \in W$ и $\sigma(t, \theta) \in \mathcal{FS}(\mathcal{TN})$ такие, что $L(\sigma(t, \theta)) = w(a, \theta)$.

$\mathcal{TN}$ и $\mathcal{TN}'$ называются *ИНТ-тестово эквивалентными* (обозначается $\mathcal{TN} \sim_{int} \mathcal{TN}'$), если для любой последовательности $w \in (Act \times \mathbb{T})^*$ и любого множества $W \subseteq (Act \times \mathbb{T})$, $\mathcal{TN}$ **after** w $\mathbf{MUST}_{int}$ $W \Leftrightarrow \mathcal{TN}'$ **after** w $\mathbf{MUST}_{int}$ W.

Пример 6. ВСП $\mathcal{TN}_2$, $\mathcal{TN}_3$ и $\mathcal{TN}_4$, изображенные на рис. 3, ИНТ-тестово эквивалентны, тогда как $\mathcal{TN}_1$ и $\mathcal{TN}_2$ не являются таковыми. Легко проверить, что $TCT(\mathcal{TN}_2)$ **after** $w = (b,0)(b,0)$ $\mathbf{MUST}_{int}$ $W = \{(a, 3.9)\}$. Однако в $TCT(\mathcal{TN}_1)$ существует последовательность срабатываний, которая помечена w и после которой невозможно срабатывание перехода, помеченного a, в момент времени 3.9. Таким образом, не выполняется $TCT(\mathcal{TN}_1)$ **after** w $\mathbf{MUST}_{int}$ W. □

Тестовые эквивалентности, учитывающие отношение причинной зависимости между действиями, были впервые введены Асето и др. в статье [5] в контексте моделей структур событий. При этом в качестве вычислений процесса вместо последовательностей выполняемых действий рассматривались их частично-упорядоченные мультимножества (ЧУММы). В работе [6] вместо множеств дальнейших действий использовались непосредственные расширения выполняемых ЧУММов. Кроме того, в [6] была предложена еще одна версия причинной тестовой эквивалентности, которая использует в качестве вычислений ЧУМы выполняемых действий и которая, как было показано, является более строгой эквивалентностью. Следуя этому подходу, далее определяется временная ЧУМ-тестовая эквивалентность для ВСП с использованием ее корректных временных причинных сетей-процессов.

Определение 8. Пусть $\mathcal{TN}$ и $\mathcal{TN}'$ – ВСП.

Для ВЧУМ TP и множества **TP** ВЧУМов такого, что $TP \prec$[7] TP' для любого $TP' \in$ **TP**, $\mathcal{TN}$ **after** TP $\mathbf{MUST}_{tpos}$ **TP**, если для любого $\pi = (TN, \varphi) \in \mathcal{CP}(\mathcal{TN})$ и для любого изоморфизма f: $\eta(TN) \to TP$ существуют $TP' \in$ **TP**, $\pi' = (TN', \varphi') \in \mathcal{CP}(\mathcal{TN})$ и изоморфизм f': $\eta(TN') \to TP'$ такие, что $\pi \to \pi'$ и $f \subseteq f'$.

$\mathcal{TN}$ и $\mathcal{TN}'$ называются *ВЧУМ-тестово эквивалентными* (обозначается $\mathcal{TN} \sim_{tpos} \mathcal{TN}'$), если для любого ВЧУМ TP и любого множества **TP** ВЧУМов такого, что $TP \prec TP'$ для всех $TP' \in$ **TP**, выполняется условие: $\mathcal{TN}$ **after** TP $\mathbf{MUST}_{tpos}$ **TP**′ $\Leftrightarrow$ $\mathcal{TN}'$ **after** TP $\mathbf{MUST}_{tpos}$ **TP**.

Пример 7. Рассмотрим ВСП $\mathcal{TN}_2$, $\mathcal{TN}_3$ и $\mathcal{TN}_4$, изображенные на рис. 3. Легко видеть, что $\mathcal{TN}_2$ и $\mathcal{TN}_3$ ВЧУМ-тестово эквивалентны, тогда как $\mathcal{TN}_3$ и $\mathcal{TN}_4$ не являются таковыми. Убедимся в последнем. Определим ВЧУМ $TP = (\{x_1, x_2\}, \preceq, \lambda, \tau)$, где $\preceq = \{(x_i, x_i) | 1 \le i \le 2\}$, $\lambda(x_1) = \lambda(x_2) = b$, $\tau(x_1) = \tau'(x_2) = 0$; и ВЧУМ $TP' = (\{x_1, x_2, x_3\}, \preceq', \lambda', \tau')$ где $\preceq' = \{(x_i, x_i) | 1 \le i \le 3\} \cup \{(x_2, x_3)\}$, $\lambda'(x_1) = \lambda'(x_2) = b$, $\lambda'(x_3) = a$, $\tau'(x_1) = \tau'(x_2) = 0$ и $\tau'(x_3) = 3.9$. Для любого временного причинного сети-процесса $\pi_3 = (TN_3, \varphi_3) \in \mathcal{CP}(\mathcal{TN}_3)$, в котором E_{TN_3} состоит из двух параллельных событий с пометками b и времен-

[7] ВЧУМ $\eta = (X, \preceq, \lambda, \tau)$ называется *префиксом* ВЧУМ $\eta' = (X', \preceq', \lambda', \tau')$ (обозначается $\eta \prec \eta'$), если $X \subseteq X'$, $X' \backslash X = \{x\}$, $\preceq = \preceq' \cap (X \times X)$, $\lambda = \lambda'|_X$, $\tau = \tau'|_X$ и x является максимальным относительно $\preceq'$ элементом X'.

ными значениями, равными 0, и для любого изоморфизма $f_3 : \eta(TN_3) \to TP$ можно найти временной причинный сеть-процесс $\pi'_3 = (TN'_3, \varphi'_3) \in \mathcal{CP}(\mathcal{TN}_3)$, в котором $E_{TN'_3}$ состоит из двух параллельных событий с пометками b и временными значениями 0 и третьего события с пометкой a и временным значением 3.9, находящегося в отношении причинной зависимости с одним из b, и изоморфизм $f'_3 : \eta(TN'_3) \to TP'$ такие, что $\pi_3 \to \pi'_3$ и $f_3 \subset f'_3$. Однако, это не так в случае ВСП $\mathcal{TN}_4$. □

Далее определим тестовую эквивалентность для ВСП на основе их временных причинных деревьев. При этом будем придерживаться метода, использованного для модели структур событий в [6]. Тесты будут строиться с учетом временных значений на основе множества пометок $Act \times \mathbb{T} \times 2^{\mathbb{N}}$ дуг деревьев.

О п р е д е л е н и е 9. Пусть $\mathcal{TN}$ и $\mathcal{TN}'$ – ВСП и $TCT(\mathcal{TN}) = (\mathcal{FS}(\mathcal{TN}), A, \phi)$ и $TCT(\mathcal{TN}') = (\mathcal{FS}(\mathcal{TN}'), A', \phi')$ – их временные причинные деревья.

Для последовательности $w \in (Act \times \mathbb{T} \times 2^{\mathbb{N}})^*$ и множества $\mathbf{W} \subseteq (Act \times \mathbb{T} \times 2^{\mathbb{N}})$, $TCT(\mathcal{TN})$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W}$, если для всех путей u в $TCT(\mathcal{TN})$ из корня в вершину n таких, что $\phi(u) = w$, существуют пометка $(a, d, K) \in \mathbf{W}$ и дуга r из вершины n такие, что $\phi(r) = (a, d, K)$;

$\mathcal{TN}$ и $\mathcal{TN}'$ называются *ВПД-тестово эквивалентными* (обозначается $\mathcal{TN} \sim_{tct} \mathcal{TN}'$), если для любой последовательности $w \in (Act \times \mathbb{T} \times 2^{\mathbb{N}})^*$ и для любого множества $\mathbf{W} \subseteq (Act \times \mathbb{T} \times 2^{\mathbb{N}})$, $TCT(\mathcal{TN})$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W} \Leftrightarrow TCT(\mathcal{TN}')$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W}$.

П р и м е р 8. Рассмотрим ВСП $\mathcal{TN}_2$, $\mathcal{TN}_3$ и $\mathcal{TN}_4$, изображенные на рис. 3. Легко видеть, что $\mathcal{TN}_2$ и $\mathcal{TN}_3$ ВПД-тестово эквивалентны, а $\mathcal{TN}_3$ и $\mathcal{TN}_4$ не являются таковыми. Убедимся в последнем факте. Для этого определим $w = (b, 0, \varnothing)(b, 0, \varnothing)$ и $\mathbf{W} = \{(a, 3.9, \{1\})\}$. Легко проверить, что $TCT(\mathcal{TN}_3)$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W}$. В $TCT(\mathcal{TN}_4)$ существуют два пути, помеченных $(b, 0, \varnothing)(b, 0, \varnothing)$, один из них заканчивается в вершине, из которой есть дуга с пометкой $(a, 3.9, \{1\})$, а из вершины, в которую ведет другой путь, такой дуги нет. Таким образом, не выполняется $TCT(\mathcal{TN}_4)$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W}$. □

Из определений ИНТ-, ВЧУМ- и ВПД-тестовых эквивалентностей очевидным образом следует

Л е м м а 3. *Пусть $\mathcal{TN}_1$ и $\mathcal{TN}_2$ – ВСП. Тогда*

$$\mathcal{TN}_1 \sim_{int} \mathcal{TN}_2 \Rightarrow \mathcal{L}(\mathcal{TN}_1) = \mathcal{L}(\mathcal{TN}_2),$$

$$\mathcal{TN}_1 \sim_{tpos} \mathcal{TN}_2 \Rightarrow \mathcal{TP}os(\mathcal{TN}_1) = \mathcal{TP}os(\mathcal{TN}_2),$$

$$\mathcal{TN}_1 \sim_{tct} \mathcal{TN}_2 \Rightarrow \mathcal{L}(TCT(\mathcal{TN}_1)) = \mathcal{L}(TCT(\mathcal{TN}_2)).$$

Установим связи между ИНТ- и ВПД-тестовыми эквивалентностями.

Т е о р е м а 1. $\mathcal{TN}_1 \sim_{tct} \mathcal{TN}_2 \Rightarrow \mathcal{TN}_1 \sim_{int} \mathcal{TN}_2$.

Д о к а з а т е л ь с т в о. Предположим, что $\mathcal{TN}_1 \sim_{tct} \mathcal{TN}_2$. Покажем, что верно $\mathcal{TN}_1 \sim_{int} \mathcal{TN}_2$. Предположим обратное, т.е. существуют $w \in (Act \times \mathbb{T})^*$ и $W \subseteq (Act \times \mathbb{T})$ такие, что $\mathcal{TN}_1$ **after** w $\mathbf{MUST}_{int}$ W, однако $\neg(\mathcal{TN}_2$ **after** w $\mathbf{MUST}_{int}$ $W)$. Последнее означает, что существует $\sigma_2 \in \mathcal{FS}(\mathcal{TN}_2)$ такая, что $L(\sigma_2) = w$, и для любых $(a, \theta) \in W$ и $\sigma_2(t', \theta) \in \in \mathcal{FS}(\mathcal{TN}_2)$ не верно, что $L(\sigma_2(t', \theta)) = w(a, \theta)$. Используя определения, получаем, что $w \in \mathcal{L}(\mathcal{TN}_2)$ и, более того, $\tilde{w} = \phi_2(path(\sigma_2)) \in \mathcal{L}(TCT(\mathcal{TN}_2))$, при этом $\tilde{w}|_{(Act \times \mathbb{T})^*} = w$. Определим множество $\mathbf{W} = \{(a, \theta, K) | (a, \theta) \in W$, $\exists \sigma \in \mathcal{FS}(\mathcal{TN}_1) : L(\sigma) = w$ и $\phi_1(path(\sigma)) = \tilde{w}$, $\exists$ дуга r из σ в $TCT(\mathcal{TN}_1) : \phi_1(r) = = (a, \theta, K)\}$. Покажем, что $\mathcal{TN}_1$ **after** $\tilde{w}$ $\mathbf{MUST}_{tct}$ $\mathbf{W}$. Возьмем произвольную $\sigma_1 \in \mathcal{FS}(\mathcal{TN}_1)$ такую, что $\phi_1(path(\sigma_1)) = \tilde{w}$. Такая σ_1 существует, поскольку $\tilde{w} \in \mathcal{L}(TCT(\mathcal{TN}_1))$, по лемме 3. Более того, имеем, что $w \in \mathcal{L}(\mathcal{TN}_1)$, по утверждению 3. Так как $\mathcal{TN}_1$ **after** w $\mathbf{MUST}_{int}$ W, то существуют $(a, \theta) \in W$ и $\sigma_1(t, \theta) \in \mathcal{FS}(\mathcal{TN}_1)$ такие, что $L(\sigma_1(t, \theta)) = w(a, \theta)$. Согласно построению временного причинного дерева, найдется дуга $r = (\sigma_1, \sigma_1(t, \theta))$ в $TCT(\mathcal{TN}_1)$ такая, что $\phi_1(r) = (a, \theta, K)$. Значит, имеем, что $\phi_1(r) \in \mathbf{W}$. В силу произвольности выбора σ_1, верно, что $\mathcal{TN}_1$ **after** $\tilde{w}$ $\mathbf{MUST}_{tct}$ $\mathbf{W}$. Таким образом, пришли к противоречию, так как легко проверить, что $\neg(\mathcal{TN}_2$ **after** $\tilde{w}$ $\mathbf{MUST}_{tct}$ $\mathbf{W})$. □

В заключение, покажем совпадение тестовых эквивалентностей для ВСП в семантиках временных частично-упорядоченных множеств и временных причинных деревьев.

Т е о р е м а 2. *Пусть $\mathcal{TN}_1$ и $\mathcal{TN}_2$ – ВСП. Тогда*

$$\mathcal{TN}_1 \sim_{tpos} \mathcal{TN}_2 \Leftrightarrow \mathcal{TN}_1 \sim_{tct} \mathcal{TN}_2.$$

Д о к а з а т е л ь с т в о. Рассмотрим доказательство слева направо (доказательство справа налево аналогично). Пусть $TCT(\mathcal{TN}_i) = (\mathcal{FS}(\mathcal{TN}_i), A_i, \phi_i)$ $(i = 1, 2)$. Предположим, что $\mathcal{TN}_1 \sim_{tpos} \mathcal{TN}_2$. Тогда, согласно лемме 3, имеем $\mathcal{TP}os(\mathcal{TN}_1) = \mathcal{TP}os(\mathcal{TN}_2)$. По утверждению 3, получаем, что $\mathcal{L}(TCT(\mathcal{TN}_1)) = = \mathcal{L}(TCT(\mathcal{TN}_2))$. Покажем, что $\mathcal{TN}_1 \sim_{tct} \mathcal{TN}_2$. Возьмем произвольные $w \in (Act \times \mathbb{T} \times 2^{\mathbb{N}})^*$ и $\mathbf{W} \subseteq (Act \times \mathbb{T} \times 2^{\mathbb{N}})$. Без потери общности полагаем, что $|w| = n$ $(n \geq 0)$. Предположим, что $TCT(\mathcal{TN}_1)$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W}$. Проверим, что $TCT(\mathcal{TN}_2)$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W}$.

Если $w \notin \mathcal{L}(TCT(\mathcal{TN}_1)) = \mathcal{L}(TCT(\mathcal{TN}_2))$, то результат очевиден. Рассмотрим случай, когда $w \in \mathcal{L}(TCT(\mathcal{TN}_1)) = \mathcal{L}(TCT(\mathcal{TN}_2))$. Тогда можно выбрать любой путь u из корня в некоторую вершину $\sigma \in \mathcal{FS}(\mathcal{TN}_1)$ такую, что $\phi_1(u) = w$. Согласно утверждению 1($б$), существует единственный (с точностью до изоморфизма) временной причинный сеть-процесс $\pi_\sigma = (TN_\sigma, \varphi_\sigma) \in \mathcal{CP}(\mathcal{TN}_1)$ и единственная линеаризация $\rho_\sigma = e_1^\sigma \ldots e_n^\sigma\ TN_\sigma$ такие, что $FS_{\pi_\sigma}(\rho_\sigma) = \sigma$. Обозначим $TP_w = \eta(TN_\sigma) \in \in \mathcal{TP}os(\mathcal{TN}_1)$.

Для каждой $(a, \theta, K) \in \mathbf{W}$ сконструируем ВЧУМ $TP_{(a,\theta,K)} = (X, \preceq, \lambda, \tau)$ следующим образом: $X = E_{TN_\sigma} \cup \{e_{(a,\theta,K)}\}$ $(e_{(a,\theta,K)} \notin E_{TN_\sigma})$; $\preceq = \preceq_{TN_\sigma} \cup \{(e^\sigma_{n-k+1}, e_{(a,\theta,K)}) | k \in K\}$; $\lambda|_{E_{TN_\sigma}} = \lambda_{TN_\sigma}$, $\lambda(e_{(a,\theta,K)}) = a$; $\tau|_{E_{TN_\sigma}} = \tau_{TN_\sigma}$, $\tau(e_{(a,\theta,K)}) = \tau(TN_\sigma) + \theta$. Обозначим множество всех построенных ВЧУМ как $\mathbf{TP_W} = \{TP_{(a,\theta,K)} | (a, \theta, K) \in \mathbf{W}\}$.

Проверим, что $\mathcal{TN}_1$ **after** TP_w $\mathbf{MUST}_{tpos}$ $\mathbf{TP}_W$. Возьмем произвольный временной причинный сеть-процесс $\pi_1 = (TN_1, \varphi_1) \in \mathcal{CP}(\mathcal{TN}_1)$ и изоморфизм f_1: $\eta(TN_1) \to TP_w$. Так как $TP_w \in \mathcal{TP}os(\mathcal{TN}_1)$, то такие π_1 и f_1 существуют. Из утверждения 2(a) получаем, что $e_1^1 \ldots e_n^1 = \rho_1 = (f_1)^{-1} : \eta(TN_\sigma) \to \eta(TN_1)(\rho_\sigma)$ является линеаризацией TN_1 такой, что $w = \phi(path(\sigma_1 = FS_{\pi_1}(\rho_1)))$. Так как $TCT(\mathcal{TN}_1)$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W}$, то существует пометка $(a_1', \theta_1', K_1') \in \mathbf{W}$ и дуга r_1 из вершины σ_1 такие, что $\phi_1(r_1) = (a_1', \theta_1', K_1')$. Тогда можно найти $TP_1' = TP_{(a_1', \theta_1', K_1')} \in \mathbf{TP_W}$. Следовательно, по построению множества $\mathbf{TP_W}$, получаем, что $\{e_{(a_1', \theta_1', K_1')}\} = E_{TP_1'} \setminus E_{TN_\sigma}$, $a_1' = \lambda_{TP_1'}(e_{(a_1', \theta_1', K_1')})$, $\theta_1' = \tau_{TP_1'}(e_{(a_1', \theta_1', K_1')}) - \tau(TN_\sigma)$, $K_1' = \{n - l + 1 \mid e_l^\sigma \preceq_{TP_1'} e_{(a_1', \theta_1', K_1')}\}$. Более того, по определению $TCT(\mathcal{TN}_1)$, существует $\sigma_1(t_1', \theta_1') \in \mathcal{FS}(\mathcal{TN}_1)$, $(t_1' \in T_{\mathcal{TN}_1})$, такая что $r_1 = (\sigma_1, \sigma_1(t_1', \theta_1'))$ и $\phi_1(\sigma_1', \sigma_1(t_1', \theta_1')) = (l_{\mathcal{TN}_1}(t_1') = a_1', \theta_1', K_1')$. Из леммы 2($a$) следует наличие временного причинного сети-процесса $\pi_1' = (TN_1', \varphi_1') \in \mathcal{CP}(\mathcal{TN}_1)$ такого, что $\pi_1 \to \pi_1'$ и $\sigma_1(t_1', \theta_1') = FS_{\pi_1'}(\rho_1 e_1')$ для некоторой линеаризации $\rho_1 e_1'\ TN_1'$, т.е. $\varphi_1'(e_1') = t_1'$. Определим функцию f_1': $\eta(TN_1') \to TP_1'$ так: $f_1'|_{E_{\eta(TN_1)}} = f_1$, $f_1'(e_1') = e_{(a_1', \theta_1', K_1')}$. Кроме того, $\lambda_{\eta(TN_1')}(e_1') = a_1' = \lambda_{TP_1}(e_{(a_1', \theta_1', K_1')})$; $\tau_{\eta(TN_1')}(e_1') = \theta_1' + \tau(TN_1) = \theta_1' + \tau(TN_\sigma) = \tau_{TP_1'}(e_{(a_1', \theta_1', K_1')})$; $e^1_{n-k+1} \preceq_{\eta(TN_1')} e_1' \Leftrightarrow f_1'(e^1_{n-k+1}) = e^\sigma_{n-k+1} \preceq_{TP_1} e_{(a_1', \theta_1', K_1')}$, для всех $k \in K_1'$. Следовательно, f_1' является изоморфизмом и $f_1 \subseteq f_1'$. Таким образом, $\mathcal{TN}_1$ **after** TP_w $\mathbf{MUST}_{tpos}$ $\mathbf{TP_W}$. Тогда, по предположению теоремы, получаем, что $\mathcal{TN}_2$ **after** TP_w $\mathbf{MUST}_{tpos}$ $\mathbf{TP_W}$.

Далее покажем, что $TCT(\mathcal{TN}_2)$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W}$. Возьмем произвольный путь u_2 в $TCT(\mathcal{TN}_2)$ из корня в вершину σ_2 такой, что $\phi_2(u_2) = w$. Так как $w \in \mathcal{L}(TCT(\mathcal{TN}_2))$, то найдется хотя бы один такой путь u_2 в $TCT(\mathcal{TN}_2)$. Согласно утверждению 1($б$), существует единственный (с точностью до изоморфизма) временной причинный сеть-процесс $\pi_{\sigma_2} = (TN_2, \varphi_2) \in \mathcal{CP}(\mathcal{TN}_2)$ и единственная линеаризация $\rho_2 = e_1^2 \ldots e_n^2\ TN_2$ такие, что $FS_{\pi_{\sigma_2}}(\rho_2) = \sigma_2$. Используя утверждение 2($б$), получаем наличие изоморфизма $f_2 : \eta(TN_2) \to TP_w$ такого, что $f_2(\rho_2) = \rho_\sigma$. Так как $\mathcal{TN}_2$ **after** TP_w $\mathbf{MUST}_{tpos}$ $\mathbf{TP_W}$, то существуют $TP_2' \in \mathbf{TP_W}$, $\pi_{\sigma_2}' = (TN_2', \varphi_2') \in \mathcal{CP}(\mathcal{TN}_2)$ и изоморфизм $f_2' : \eta(TN_2') \to TP_2'$ такие, что $\pi_{\sigma_2} \to \pi_2'$ и $f_2 \subseteq f_2'$. Согласно лемме 2($б$), найдется $\sigma_2(\tilde{t}, \tilde{\theta}) \in \mathcal{FS}(\mathcal{TN}_2)$ такая, что для некоторой линеаризации $\rho_2 e_2'\ TN_2'$ имеем, что $\sigma_2(\tilde{t}, \tilde{\theta}) = FS_{\pi_2'}(\rho_2 e_2')$. По построению $\mathbf{TP_W}$, существует пометка $(a, \theta, K) \in \in \mathbf{W}$ такая, что $TP_2' = TP_{(a,\theta,K)}$, и, следовательно, $\{e_{(a,\theta,K)}\} = E_{TP_2'} \setminus E_{TN_\sigma}$. Так как $TN_2 \to TN_2'$ и $TP_w \prec\cdot\ TP_2'$, то $\{e_2'\} = E_{TN_2'} \setminus E_{TN_2}$ и $f_2'(e_2') = e_{(a,\theta,K)}$. Поскольку f_2' – изоморфизм, верно: $\lambda_{\eta(TN_2')}(e_2') = \lambda_{TP_2'}(e_{(a,\theta,K)}) = a$, $\tau_{\eta(TN_2')}(e_2')\tau_{TP_2'} = (e_{(a,\theta,K)}) = \tau(TN_\sigma) + \theta = \tau(TN_2) + \theta$, и $e_i^\sigma \preceq_{TP_2'} e_{(a,\theta,K)} \Leftrightarrow (f_2')^{-1}(e_i^\sigma) = e_i^2 \preceq_{\eta(TN_2')} e_2'$ для всех $1 \le i \le n$. Тогда получаем, что $(\tilde{t}, \tilde{\theta}) = (\varphi_2'(e_2'), \theta)$ и $e^2_{n-k+1} \preceq_{\eta(TN_2')} e_2'$ для всех $k \in K$. Следовательно, в $TCT(\mathcal{TN}_2)$ существует дуга $r_2 = (\sigma_2, \sigma_2(\tilde{t}, \tilde{\theta}))$ такая, что $\phi_2(r_2) = (a, \theta, K)$. Таким образом, имеем, что $TCT(\mathcal{TN}_1)$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W} \Rightarrow TCT(\mathcal{TN}_2)$ **after** w $\mathbf{MUST}_{tct}$ $\mathbf{W}$.

В силу симметрии, верно, что $\mathcal{TN}_1 \sim_{tpos} \mathcal{TN}_2 \Rightarrow \Rightarrow \mathcal{TN}_1 \sim_{tct} \mathcal{TN}_2$. □

5. ЗАКЛЮЧЕНИЕ

В данной статье было показано, что хорошо известные в теории безвременных и временных моделей структур событий причинно-зависимые тестовые эквивалентности могут быть обобщены на модели непрерывно-временных сетей Петри.

В частности, были введены и изучены тестовые эквивалентности в интерливинговой, частично-упорядоченной и комбинированных семантиках в контексте безопасных сетей Петри, переходы которых помечены временными интервалами и каждый переход, имеющий достаточное количество фишек во входных местах, должен срабатывать тогда, когда его счетчик достигнет некоторого значения, принадлежащего его временному интервалу. При исследованиях были построены три представления вычислений непрерывно-временной сети Петри: последовательности срабатываний, представляющие интерливинговую семантику, временные сети-процессы, из причинных сетей которых выводятся частичные порядки, и причинное дерево, построенное из последовательностей срабатываний и частичных порядков причинных сетей. Были найдены взаимосвязи, с одной стороны, между последовательностями срабатываний и корректными временными причинными сетями-процессами, и, с другой стороны, между последними и помеченными путями во временных причинных деревьях. Было установлено, что интерливинговая тестовая эквивалентость слабее, чем тестовая эквивалентность, определенная с использованием временного причинного дерева. Как основной результат, доказано совпадение тестовых эквивалентностей в семантиках временного частичного порядка и временного причинного дерева. Заметим, что подобный результат верен и для безвременных версий тестовых эквивалентностей в контексте свободных от контактов элементарных сетевых систем.

В дальнейшем планируется исследовать взаимосвязи рассмотренных эквивалентностей и семантик с другими эквивалентностями из спектров линейного/ветвящегося времени и интерливинга/частичного порядка ([25]). Также следует изучить возможность расширения полученных результатов на модели непрерывно-временных сетей Петри с невидимыми действиями.

СПИСОК ЛИТЕРАТУРЫ

1. *De Nicola R., Hennessy M.* Testing equivalence for processes // Theoretical Computer Science. 1984. V. 34. P. 83–133.
2. *De Nicola R.* Extensional equivalences for transition systems // Acta Informatica. 1987. V. 24. № 2. P. 211–237.
3. *Cleaveland R., Hennessy M.* Testing equivalence as a bisimulation equivalence // Lecture Notes in Computer Science. 1989. V. 407. P. 11–23.
4. *Pomello, L., Rozenberg, G., Simone C.* A Survey of Equivalence Notions for Net Based Systems // Lecture Notes in Computer Science. 1992. V. 609. P. 410–472.
5. *Aceto L., De Nicola R., Fantechi A.* Testing equivalences for event structures // Lecture Notes in Computer Science. 1987. V. 280. P. 1–20.
6. *Goltz U., Wehrheim H.* Causal testing // Lecture Notes in Computer Science. 1996. V. 1113. P. 394–406.
7. *Aceto L.* History preserving, causal and mixed-ordering equivalence over stable event structures // Fundamenta Informaticae. 1992. V. 17. № 4. P. 319–331.
8. *Darondeau Ph., Degano P.* Refinement of actions in event structures and causal trees // Theoretical Computer Science. 1993. V. 118. № 1. P. 21–48.
9. *Nielsen M., Rozenberg G., Thiagarajan P.S.* Behavioural notions for elementary net systems // Distributed Computing. 1990. V. 4. № 1. P. 45–57.
10. *Hoogers P.W., Kleijn H.C.M., Thiagarajan P.S.* An event structure semantics for general Petri nets // Theoretical Computer Science. 1996. V. 153. № 1–2. P. 129–170.
11. *van Glabbeek R.J., Goltz U., Schicke J.-W.* On causal semantics of Petri nets // Lecture Notes in Computer Science. 2011. V. 6901. P. 43–59.
12. *Cleaveland R., Zwarico A.E.* A theory of testing for real-time // Proc. 6th IEEE Symp. on Logic in Comput. Sci. (*LICS'91*), Amsterdam, The Netherlands. 1991. P. 110–119.
13. *Llana L., de Frutos D.* Denotational semantics for timed testing// Lecture Notes in Computer Science. 1997. V. 1233. P. 368–382.
14. *Hennessy M., Regan T.* A process algebra for timed systems // Information and Computation. 1995. V. 117. P. 221–239.
15. *Corradini F., Vogler W., Jenner L.* Comparing the Worst-Case Efficiency of Asynchronous Systems with PAFAS // Acta Informatica. 2002. V. 38. 11–12. P. 735–792.
16. *Bihler E., Vogler W.* Timed Petri Nets: Efficiency of Asynchronous Systems // Lecture Notes in Computer Science. 2004. V. 3185. P. 25–58.
17. *Murphy D.* Time and duration in noninterleaving concurrency // Fundamenta Informaticae. 1993. V. 19. P. 403–416.
18. *Andreeva M., Bozhenkova E., Virbitskaite I.* Analysis of timed concurrent models based on testing equivalence // Fundamenta Informaticae. 2000. V. 43. P. 1–20.
19. *Andreeva M., Virbitskaite I.* Timed equivalences for timed event structures // Lecture Notes in Computer Science. 2005. V. 3606. P. 16–25.
20. *Andreeva M., Virbitskaite I.* Observational Equivalences for Timed Stable Event Structures // Fundamenta Informaticae. 2006. V. 72. № 1–3. P. 1–19.
21. *Valero V., de Frutos D., Cuartero F.* Timed processes of timed Petri nets // Lecture Notes in Computer Science. 1995. V. 935. P. 490–509.
22. *Вирбицкайте И.Б., Боровлев В.А., Попова-Цейгманн Л.* Истинно-параллельная и недетерминированная семантика временных сетей Петри // Программирование. 2016. № 4. С. 4–16.
23. *Aura T., Lilius J.* A causal semantics for time Petri nets // Theoretical Computer Scinece. 2000. V. 243. № 1–2. P. 409–447.
24. *Бушин Д.И., Вирбицкайте И.Б.* Компаративная трассовая семантика временных сетей Петри // Программирование. 2015. № 3. С. 20–31.
25. *Virbitskaite I., Bushin D., Best E.* True concurrent equivalences in time Petri nets // Fundamenta Informaticae. 2016. V. 149. № 4. P. 401–418.

ПРОГРАММИРОВАНИЕ, 2020, № 4, с. 14–26

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.421.6

ДЕДУКТИВНАЯ ВЕРИФИКАЦИЯ REFLEX-ПРОГРАММ

© 2020 г. И. С. Ануреев[a,b,*], Н. О. Гаранина[a,b,], Т. В. Лях[b,***], А. С. Розов[b,****], В. Е. Зюбин[b,*****], С. П. Горлач[c,******]**

[a] *Институт систем информатики им. А.П. Ершова СО РАН*
630090 Новосибирск, пр. ак. Лаврентьева, д. 6, Россия
[b] *Институт автоматики и электрометрии СО РАН*
630090 Новосибирск, проспект Академика Коптюга, д. 1, Россия
[c] *Университет Мюнстера*
48149 Мюнстер, ул. Эйнштейна, д. 62, Германия
**E-mail: anureev@gmail.com*
***E-mail: garanina@iis.nsk.su*
****E-mail: antsys_nsu@mail.ru*
*****E-mail: rozov@iae.nsk.su*
******E-mail: zyubin@iae.nsk.su*
*******E-mail: gorlatch@uni-muenster.de*

Поступила в редакцию 10.02.2020 г.
После доработки 20.02.2020 г.
Принята к публикации 15.03.2020 г.

В этой статье представлен новый двухшаговый метод верификации для управляющего программного обеспечения. Новизна метода состоит в том, чтобы свести проверку временных свойств управляющей программы к дедуктивной верификации императивной программы в стиле Хоара, которая явно моделирует время и историю выполнения управляющей программы. Метод применяется к программам на языке Reflex – предметно-ориентированном расширении языка C, разработанном в качестве альтернативы языкам стандарта IEC 61131-3. Reflex – это процесс-ориентированный язык, описывающий управляющие программы в терминах взаимодействующих процессов, управляемых событиями операций и операций с дискретными временными интервалами. На первом шаге аннотированная Reflex-программа транслируется в эквивалентную аннотированную императивную программу на ограниченном подмножестве языка C, расширенном логическим типом bool и супертипом value, объединяющем значения, которые могут возвращать функции и операции языка Reflex, а также оператором havoc x, присваивающем произвольное значение переменной x. На втором шаге выполняется дедуктивная верификация полученной императивной программы. Мы иллюстрируем наш метод на примере дедуктивной верификации Reflex-программы, управляющей сушилкой для рук. Пример включает исходную Reflex-программу, набор требований, результирующую аннотированную программу, порожденные условия корректности и результаты доказательства этих условий в Z3py – интерфейсе к SMT-решателю Z3, представленном на языке Python.

DOI: 10.31857/S0132347420040020

1. ВВЕДЕНИЕ

Растущая сложность систем управления, используемых в нашей повседневной жизни, требует переоценки инструментов проектирования и разработки. Такая переоценка особенно важна для критических с точки зрения безопасности систем, где неправильное поведение и/или отсутствие надежности может привести к неприемлемой потере средств или даже человеческой жизни. Такие системы широко распространены в промышленности, особенно на химических и металлургических заводах. Поскольку поведение сложных систем управления определяется программным обеспечением, исследование управляющего программного обеспечения представляет большой интерес. Корректное поведение в различных условиях окружающей среды должно быть гарантировано. При отказе оборудования, например, поломках механических частей объекта управления или исполнительных устройств, система должна автоматически сформировать управляющие воздействия, которые предотвращают опасные последствия таких отказов. Общепринятое название этого свойства системы управления – отказоустойчивость (fault tolerant behavior) [1].

Из-за специфики предметной области системы управления обычно основаны на промышленных контроллерах, также известных как программируемые логические контроллеры (ПЛК), и специализированные языки используются для проектирования управляющего программного обеспечения для таких систем. Наиболее распространенными в области программирования ПЛК являются языки стандарта IEC 61131-3 [2]. Однако, поскольку сложность управляющего программного обеспечения возрастает, а качество становится более приоритетным, 35-летняя технология, основанная на подходе IEC 61131-3, не всегда способна удовлетворить современные требования [3]. Поэтому исследователи предпринимают попытки либо модифицировать модель разработки IEC 61131-3, например, обогатить модель объектно-ориентированными концепциями [4], либо вообще предложить взамен IEC 61131-3 альтернативные понятийно-языковые средства, например, [5–8].

Для устранения ограничений и проблем при разработке современного комплексного управляющего программного обеспечения в [9] была предложена парадигма процесс-ориентированного программирования. Эта парадигма представляет управляющее программное обеспечение в виде набора взаимодействующих процессов, где процессы – это конечные автоматы, дополненные неактивными состояниями и специальными функциями управления процессами и обработки временных интервалов. По сравнению с известными модификациями конечных автоматов, например, последовательными взаимодействующими процессами [10], диаграммами состояний Харела [11], автоматами ввода/вывода [12], Esterel [13], гибридными автоматами [14], исчислением взаимодействующих систем [15] и их временными расширениями [16, 17], этот метод обеспечивает средства для спецификации параллелизма и при этом сохраняет линейность потока управления на уровне процесса. Таким образом, он обеспечивает концептуальную основу для разработки процессно-ориентированных языков, предназначенных для создания программного обеспечения для ПЛК. Процесс-ориентированный подход был реализован в таких предметно-ориентированных языках как SPARM [18], Reflex [19] и IndustrialC [20]. Эти языки являются C-подобными и, поэтому, просты в изучении. Трансляторы для них порождают C-код, который обеспечивает кросс-платформенную переносимость. Благодаря прямой поддержке конечных автоматов и операций с плавающей запятой, эти языки позволяют достаточно просто создавать программное обеспечение для ПЛК.

Язык SPARM является предшественником языка Reflex и в настоящее время не используется. IndustrialC нацелен на использование периферийных устройств микроконтроллера (регистров, таймеров, ШИМ и т.д.) и расширяет Reflex средствами обработки прерываний. Программа на языке Reflex специфицируется как набор взаимодействующих параллельных процессов. Reflex включает специализированные инструкции для управления процессами и обработки временных интервалов. Он также предоставляет инструкции для связывания программных переменных с физическими сигналами ввода/вывода. Процедуры чтения/записи данных через регистры и их сопоставление с переменными порождаются транслятором автоматически.

Reflex предполагает цикл управления с фиксированным временем исполнения и строгую инкапсуляцию зависимых от платформы подпрограмм ввода-вывода в библиотеку, что является широко применяемым методом в системах на основе IEC 6113-3. Для обеспечения простоты поддержки и межплатформенной переносимости генерация исполняемого кода осуществляется в два этапа: транслятор Reflex генерирует C-код, а затем C-компилятор создает исполняемый код для целевой платформы. Reflex – простой язык с точки зрения операторов и типов данных. Он не имеет указателей, массивов и циклов. Несмотря на простоту, этот язык был успешно применен в нескольких индустриальных системах управления, например, в программном обеспечении для управления печью установкой для выращивания монокристаллического кремния [21]. Семантическая простота языка вместе с практической применимостью делает Reflex привлекательным для теоретических исследований.

В настоящее время проект Reflex сфокусирован на средствах проектирования и разработки критических с точки зрения безопасности систем. Благодаря своей системной независимости Reflex легко интегрируется со сторонними средствами, например, с LabVIEW [22]. Это позволяет разрабатывать программное обеспечение, сочетающее управляемое событиями поведение с расширенным графическим интерфейсом пользователя, удаленными датчиками и исполнительными устройствами, устройствами с поддержкой LabVIEW и т.д. Используя гибкость LabVIEW, был разработан набор тренажеров для учебных целей [23]. Основанные на LabVIEW симуляторы включают в себя 2D-анимацию, инструменты для отладки и языковую поддержку для обучения разработке управляющего программного обеспечения. Одним из результатов, полученных в этом направлении, является набор инструментов динамической верификации на основе LabVIEW для Reflex-программ. Динамическая верификация рассматривает программное обеспечение как черный ящик и проверяет его соответствие требованиям, наблюдая за поведением программного обеспечения во время выполнения на наборе тестовых случаев. Хотя такая процедура может помочь обнаружить наличие ошибок в программном обеспечении, она не может гарантировать их отсутствие [24].

В отличие от динамической проверки, формальные методы обычно признаются как единственный способ обеспечить полноту верификации проверяемого свойства программного обеспечения. Поэтому очень важно адаптировать формальные методы верификации для Reflex-программ.

В этой статье мы предлагаем метод дедуктивной верификации Reflex-программ. Оригинальная двухшаговая схема метода позволяет нам свести проверку временных свойств Reflex-программ к дедуктивной верификации C-подобной программы, которая явно моделирует время и историю выполнения Reflex-программы.

Работа имеет следующую структуру. В разделе 2, мы формулируем подход к спецификации временных свойств Reflex-программ и пример Reflex-программы, управляющей сушилкой для рук, а также ее временные свойства. В разделе 3 на примере программы управления сушилкой для рук описывается алгоритм трансляции аннотированных Reflex-программ в язык VOL[Reflex] (Verification-Oriented Language) – ограниченное подмножество аннотированных императивных программ на языке C, расширенное логическим типом bool и супертипом value, объединяющем значения, которые могут возвращать функции и операции языка Reflex, а также оператором havoc x, присваивающем произвольное значение переменной x. В разделе 4 представлена формальная трансляционная семантика языка Reflex в язык VOL[Reflex] (далее VOL), описывающая правила трансляции Reflex-инструкций в VOL-инструкции. Трансляция аннотированной Reflex-программы в VOL-программу – первый шаг нашего метода дедуктивной верификации. Второй шаг – дедуктивная верификация VOL-программы, состоящая в порождении условий корректности для нее и их доказательстве, – определяется в разделе 5. Он включает алгоритм порождения условий корректности и примеры условий корректности для VOL-программы – результата трансляции программы управления сушилкой для рук. В заключительном разделе 6 мы обсуждаем особенности нашего метода и направления будущих исследований.

2. СПЕЦИФИКАЦИЯ ВРЕМЕННЫХ СВОЙСТВ REFLEX-ПРОГРАММ

Наш метод верификации сводит верификацию Reflex-программ к верификации VOL-программ. Reflex-программа вместе с ее проверяемыми временными свойствами транслируется в эквивалентную VOL-программу с соответствующим набором проверяемых свойств. В этом разделе мы определим подход к спецификации для свойств Reflex-программ. Этот подход иилюстрируется на примере Reflex-программы, управляющей сушилкой для рук.

Мы специфицируем свойства Reflex-программ, используя два языка: язык аннотаций и язык аннотаторов. *Язык аннотаций* – это язык логических формул – *аннотаций*, описывающих свойства программы. *Язык аннотаторов* – это язык разметки, сопоставляющий аннотации конструкциям программы. Элементы этого языка называются *аннотаторами*. Программа, расширенная аннотаторами, называется *аннотированной программой*.

Аннотации Reflex-программ являются формулами многосортной логики первого порядка, представленными в синтаксисе интерфейса Z3py [25] к SMT-решателю Z3 [26], который используется для доказательства условий корректности, порождаемых для VOL-программ – результатов трансляции Reflex-программ.

Временные свойства Reflex-программ могут выражаться в этих аннотациях за счет использования модели дискретного времени, глобального таймера, локальных таймеров процессов и историй значений программных переменных.

Модель дискретного времени основана на периодичности взаимодействия между Reflex-программой и объектом управления. Reflex-программа и управляемый ею объект взаимодействуют через входные и выходные порты, связанные с программными переменными. В начале каждого цикла управления программа читает значения из входных портов и записывает их в соответствующие переменные. Изменение значения переменной в результате записи значения из входного порта называется *внешней модификацией* переменной. В конце цикла управления программа записывает новые значения в выходные порты. Запись значений из входных портов в переменные и чтение значений из

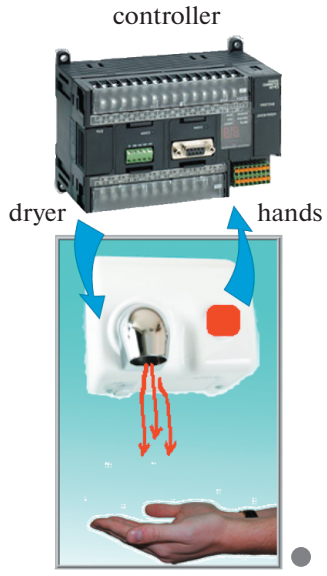


Рис. 1. Hand Dryer.

переменных в выходные порты происходит периодически с фиксированным периодом (программным циклом), измеряемым в миллисекундах. Выполнение одного программного цикла называется тактом работы Reflex-программы.

Время в аннотациях моделируется глобальным таймером и локальными таймерами процессов. Таймеры задаются переменными типа `int`. Эти переменные не могут быть переменными Reflex-программы. *Глобальный таймер* `timer` подсчитывает число тактов работы Reflex-программы. Каждый программный процесс p_i имеет свой *локальный таймер* p_{i_timer}, который также как и глобальный таймер считает время в количестве тактов работы программы.

Каждая программная переменная x интерпретируется в аннотациях как массив, в котором индекс i означает i-й цикл выполнения программы, а элемент $x[i]$ хранит значение переменной x на этом цикле. Таким образом, в контексте аннотации, x хранит историю своих изменений.

Аннотации также используют следующие переменные:

- `val` – для доступа к значению, возвращаемую функцией или операцией в Reflex-программе;
- `val_1`, ..., `val_`n – для хранения значений аргументов функций и операций, вычисляемых в Reflex-программе (n – максимальная местность функций и операций, используемых в программе);
- p_{i_state} – для доступа к текущему состоянию программного процесса p_i.

Обозначим множество аннотаций F, так что $f \in F$ является аннотацией, специфицирующей некоторое свойство Reflex-программы.

Язык аннотаторов для Reflex-программ включает четыре вида аннотаторов. *Инвариантный аннотатор* INV f; специфицирует, что свойство f должно выполняться перед каждым программным циклом. *Аннотатор начального условия* `ICON` f; специфицирует, что свойство f должно выполняться перед первым программным циклом. *Аннотатор внешнего условия* `ECON` f; накладывает ограничения на внешние модификации переменных: свойство f должно выполняться после каждой внешней модификации. *Аннотатор функций* `REQUIRES` P_f; `ENSURES` Q_f; должен размещаться непосредственно после прототипа функции $t\ f\ (t_1\ x_1, ..., t_n\ x_n)$. Прототипы функций используются чтобы вызывать функции, написанные на других языках программирования, в Reflex-программах. Этот аннотатор специфицирует предусловие P_f и постусловие Q_f функции f. Формулы P_f и Q_f зависят от $x_1, ..., x_n$. Постусловие Q_f также зависит от `val`, которая хранит значение, возвращаемое f.

Проиллюстрируем наш подход к спецификации временных свойств на примере программы управления сушилкой для рук (рис. 1, Листинг 1).

Программа использует вход с инфракрасного датчика, показывающего присутствие или отсутствие рук под сушилкой, и, на основании этих данных, управляет вентилятором и обогревателем через общий выходной сигнал.

Сформулируем несколько временных свойств для этой программы. Первое свойство заключается в том, что сушилка должна включиться за время, приемлемое с точки зрения пользователя (не позднее 100 миллисекунд), когда под ней появляются руки. Второе свойство постулирует, что сушилка никогда не включается произвольно. Эти свойства задаются ниже формулами r_1 и r_2.

```
PROGR HandDryerController {
 /* ========================== */
 /* == ANNOTATIONS : */
 /* INV inv; */
 /* ICON True ; */
 /* ECON True ; */
 /* == END OF ANNOTATIONS */
 TACT 100;
 CONST ON TRUE ;
 CONST OFF FALSE ;
 /* ============================ */
 /* I/O ports specification */
 /* direction , name , address , */
 /* offset , size of the port */
 /* ============================ */
 INPUT SENSOR_PORT 0 0 8;
 OUTPUT ACTUATOR_PORT 1 0 8;
```

```
/* ============================ */
/* processes definition */
/* ============================ */
PROC Ctrl {
 /* ===== VARIABLES ============= */
 BOOL hands = { SENSOR_PORT [1]} FOR ALL;
 BOOL dryer = { ACTUATOR_PORT [1]} FOR ALL;
 /* ===== STATES ================ */
 STATE Waiting {
  IF ( hands == ON) {
   dryer = ON;
   SET NEXT ;
  } ELSE dryer = OFF;
 }
 STATE Drying {
  IF ( hands == ON) RESET TIMEOUT ;
  TIMEOUT 10 SET STATE Waiting ;
 }
} /* \ PROC */
} /* \ PROGRAM */
```

Листинг 1: Reflex-программа управления сушилкой для рук

В Reflex-программах инструкция PROGR специфицирует имя и тело программы. Аннотаторы добавляются в начало тела программы как специальный вид комментариев. В нашем случае аннотаторами являются `INV inv;`, `ICON True;` и `ECON True;` (аннотация `inv` определена ниже). Инструкция `TACT` специфицирует число миллисекунд, соответствующее одному программному циклу. Инструкция `CONST` используется для объявления программных констант. Инструкции `INPUT` и `OUTPUT` описывают входные и выходные порты, соответственно. Программные переменные определяются через объявления переменных. Например, объявление переменной `BOOL hands = SENSOR_PORT[1] FOR ALL` связывает логическую переменную `hands` с первым битом порта `SENSOR_PORT` и указывает, что все процессы могут использовать эту переменную. Инструкция `PROC` используется для описания процессов программы. В нашем примере программа имеет один процесс `Ctrl` (контроллер), который управляет сушилкой для рук, то есть вентилятором и обогревателем. Инструкция `STATE` объявляет возможные состояния процесса. Процесс `Ctrl` может находиться в двух состояниях `WAITING` и `DRYING`. Действия, выполняемые процессом в состоянии, описываются в теле этого состояния с помощью операторов и операций.

В дополнении к С-подобным операторам и операциям, язык Reflex включает ряд специализированных операторов и операций для работы с процессами. Оператор `RESET TIMEOUT`; сбрасывает локальный таймер текущего процесса. Оператор `TIMEOUT` *x stm*; запускает выполнение оператора *stm*, когда локальный таймер текущего процесса достигает значения *x*. Оператор `SET NEXT`; переключает текущий процесс в следующее состояние (в порядке, в котором состояния объявляются в теле процесса), а оператор `SET STATE s`; переключает процесс в состояние *s*. Эти два оператора также обнуляют локальный таймер процесса.

Программа управления сушилкой не накладывает ограничения на начальное и внешнее условия. Они определяются как True. Заметим, что если бы язык проекций не включал тип `bool` как в [27] и константы ON и OFF определялись бы как `1` и `0`, то эти условия имели бы вид:

$$And(Or(\texttt{dryer[0]==0}, \texttt{dryer[0]==1}),\ Or(\texttt{hands[0]==0}, \texttt{hands[0]==1})$$

и

$$Or(\texttt{hands[timer]==0}, \texttt{hands[timer]==1}),$$

специфицируя, что переменные `dryer` и `hands` могут принимать только значения `0` и `1`. Включение типа `bool` позволяет упростить аннотации и, соответственно, порождаемые условия корректности.

Инвариант `inv` вида $And(r_1, r_2, ap)$ включает свойства r_1 и r_2, которые специфицируют желаемое поведение программы и конъюнкцию *ap* вспомогательных свойств, необходимых для их верификации. Эти вспомогательные свойства формулируются следующим образом: 1) значения программных констант равны их преодопределенным значениям, 2) глобальный таймер и все локальные таймеры неотрицательны, 3) текущие значения перемен-

ных hands и dryer совпадают с их предыдущими значениями (это свойство обеспечивает переход на новый такт выполнения программы), 4) сушилка может находиться только в двух состояниях WAITING и DRYING, и 5) сушилка в состоянии DRYING всегда включена. Мы опускаем формальную спецификацию для *ap*, так как она довольно громоздка.

Свойство r_1 вида

```
ForAll(i,
  Implies(
    And(0 <= i, i < timer),
    Implies(
      And(
        Implies(i > 0, hands[i − 1]==OFF),
        hands[i]==ON),
      dryer[i]==ON)))
```

специфицирует, что сушилка включается (dryer[*i*]==ON) не позднее, чем через 100 миллисекунд (один такт) после появления рук.

Свойство r_2 вида

```
ForAll(i,
  Implies(
    And(0 <= i, i < timer − 1),
    Implies(
      And(dryer[i]==OFF,
          hands[i + 1]==OFF),
      dryer[i + 1]==OFF)))
```

описывает требование, что сушилка никогда не включается самопроизвольно.

В следующем разделе мы представим на примере программы управления сушилкой для рук алгоритм трансляции аннотированной программы Reflex в VOL-программу, для которой порождаются условия корректности.

3. ТРАНСЛЯЦИЯ АННОТИРОВАННОЙ REFLEX-ПРОГРАММЫ В VOL-ПРОГРАММУ

Reflex-программы и VOL-программы используют один и тот же язык аннотаций. Язык аннотирования для VOL-программ включает аннотатор функций и три дополнительных аннотатора. Аннотатор ASSUME *f*; специфицирует, что *f* считается истинной в точке размещения этого аннотатора в программе. Аннотатор ASSERT *f*; утверждает, что *f* должна быть истинной в точке размещения этого аннотатора в программе. Инвариантный аннотатор INV *l f*; – специальный вариант именованного ASSERT-аннотатора с именем *l*, который обрабатывается специальным образом нашим алгоритмом порождения условий корректности.

VOL-программа – результат трансляции Reflex-программы управления сушилкой для рук – имеет следующий вид:

```
# define TACT 100
# define ON TRUE
# define OFF FALSE
# define STOP_STATE 0
# define ERROR_STATE 1
# define Ctrl_Waiting 2
# define Ctrl_Drying 3

value val;
int timer ;
int Ctrl_state ;
int Ctrl_timer ;
bool hands [];
bool dryer [];

inline void init () {
 timer = 0;
 Ctrl_state = Ctrl_Waiting ;
 Ctrl_timer = 0;
 ASSUME True ;
}

inline void Ctrl_exec () {
 switch ( Ctrl_state ) {
  case Ctrl_Waiting :
   if ( hands [ timer ] == ON) {
   dryer [ timer ] = ON;
   Ctrl_timer = 0;
   Ctrl_state = Ctrl_Drying ;
  }
else
  dryer [ timer ] = OFF;
  break ;
case Ctrl_Drying :
  if ( hands [ timer ] == ON) {
   Ctrl_timer = 0;
   Ctrl_state = Ctrl_Drying ;
  }
  if ( Ctrl_timer >= 10) {
   Ctrl_timer = 0;
   Ctrl_state = Ctrl_Waiting ;
  }
  break ;
 }
}

void main () {
```

```
  init ();
  for (;;) {
   INV lab inv;
   havoc hands [ timer ];
   ASSUME True ;
   Ctrl_exec ();
   Ctrl_timer = Ctrl_timer + 1;
   timer = timer + 1;
   hands [ timer ] = hands [timer -1];
   dryer [ timer ] = dryer [timer -1];
  }
 }
```

Листинг 2: VOL-программа управления сушилкой для рук

Рассмотрим отношения между инструкциями исходной Reflex-программы и результирующей VOL-программы.

Макроконстанта TACT, специфицирующая время программного цикла, заменяет конструкцию TACT. Константы Reflex-программы (например, ON и OFF) также заменяются макроконстантами. Макроконстанты STOP_STATE и ERROR_STATE кодируют состояние останова (специфицирующее, что выполнение процесса на текущем такте завершается нормально) и состояние ошибки (специфицирующее, что выполнение процесса на текущем такте завершается с ошибкой). Для каждого программного процесса p_i и для каждого состояния s этого процесса, макроконстанта p_i_s кодирует это состояние.

Переменные val, $\mathtt{val}_1$, ..., val_n рассматриваются как глобальные переменные VOL-программы типа value. В нашем случае, $n = 0$, так в программе не вычисляются функции и операции. Переменная timer специфицирует глобальный таймер. Для каждого программного процесса p_i, переменные p_i_state и p_i_timer специфицируют текущее состояние и локальный таймер процесса p_i. Тип t каждой Reflex-переменной x заменяется типом динамического массива t [].

Функция init() инициализирует процессы программы. Она устанавливает значение 0 для глобального таймера и всех локальных таймеров, устанавливает первый процесс в его начальное состояние и все остальные процессы в состояние останова и накладывает ограничения на начальные значения Reflex-переменных, используя аннотатор ASSUME f (для программы управления сушилкой для рук ASSUME *True*).

Для каждого процесса p_i, функция p_i_exec специфицирует действия процесса p_i во время программного цикла. Тело функции p_exec представляет оператор switch, где метки – макроконстанты, кодирующие состояния процесса p_i_exec. Все специфические для языка Reflex операторы и операции в телах состояний процессов заменяются C-инструкциями в соответствие с их семантикой.

Бесконечный цикл for(;;), специфицирующий действия всех процессов во время программного цикла является последним оператором результирующей программы. Его тело начинается с инвариантного аннотатора INV *lab inv*;, специфицирующего инвариант *inv* Reflex-программы. Следующий фрагмент

havoc hands[timer]; ASSUME *True*;

специфицирует внешние модификации Reflex-переменных (в нашем случае, hands) и ограничение *True* на них. Оператор havoc x;, добавленный в VOL из [28], моделирует присваивание произвольного значения переменной x из множества значений типа этой переменной. Третий фрагмент – это последовательность вызовов функций p_i_exec() для каждого процесса p_i. Следующий фрагмент увеличивает на 1 значения глобального таймера и всех локальных таймеров. Последний фрагмент специфицирует, что значения Reflex-переменных сохраняются после инкрементации глобального таймера, что соответствует переходу к новому такту выполнения Reflex-программы. Для программы управления сушилкой для рук, этот фрагмент имеет вид:

hands[timer] = hands[timer-1];

dryer[timer] = dryer[timer-1];.

В следующем разделе мы опишем общий вид правил трансляции Reflex-программ в VOL-программы (трансляционную семантику аннотированных Reflex-программ).

4. ТРАНСЛЯЦИОННАЯ СЕМАТИКА АННОТИРОВАННЫХ REFLEX-ПРОГРАММ

Пусть C_R и C_V – множество инструкций языков Reflex и VOL, соответственно. Программы на этих языках также включаются в эти множества. Пусть $C = C_R \cup C_V$. Трансляционная семантика языка Reflex задается бинарным отношением $\rightsquigarrow \in C \times C$ таким, что $\neg(c \rightsquigarrow c)$ для $c \in C_C$. Это отношение определяется правилами трансляции для Reflex-инструкций.

Программа. Для Reflex-программы p определим: $\{p_1, \ldots, p_n\}$ – множество имен процессов; dec_T – декларация такта; dec_c, dec_v и dec_f – последовательности деклараций констант, переменных и внешних функций; m_i – число состояний процесса p_i; $s_{i1}, \ldots, s_{im_i}$ – последовательность имен состояний p_i в порядке их определения в программе p; $body(s_{ik_i})$ – тело состояния s_{ik_i}, из которого исключены декларации переменных; $V = \{v_1, \ldots, v_k\}$ – множество имен переменных программы p; $V_e = \{v_{e1}, \ldots, v_{es}\} \subseteq V$ – подмножество имен переменных, изменяемых объектом управления. Закоди-

руем номера состояний процессов последовательными натуральными числами, начиная с 2 (коды 0 и 1 используются для состояний останова и ошибки). Пусть $num(p_i, s_{ik_i})$ – код состояния s_{ik_i} процесса p_i. Тогда правило трансляции для p имеет вид:

```
p ⇝
dec_T dec_c
#define STOP_STATE 0
#define ERROR_STATE 1
#define p1_s11 num(p1, s11)
… // повторить для других состояний p1
… // повторить для других процессов
value val; value val_1; value val_ n;
int timer;
int p1_state; int p1_timer
… // повторить для других процессов
dec_v dec_f
inline void init() {
  timer = 0;
  p1_state = p1_s11; p1_timer =0;
  p2_state = STOP_STATE; p2_timer =0;
  … // повторить для других процессов
  ASSUME icon;
}
dec1 … decn
void main() {
  init();
  for (;;) {
     INV lab inv;
     havoc ve1[timer];
     … // повторить для других переменных
     ASSUME econ;
     p1_exec(); …; pn_exec();
     p1_timer = p1_timer + 1;
     … // повторить для других процессов
     timer = timer + 1;
     v1[timer] = v1[timer-1];
     … // повторить для других переменных
  }
}
```

Здесь *icon* и *econ* – начальное и внешнее условия, соответственно; *inv* – инвариант программного цикла.

Определение dec_i функции p_i_exec, выполняющей процесс p_i в его текущем состоянии, имеет вид:

```
inline void pi_exec {
  switch (pi_state) {
    case pi_si1: body(si1) break;
    …
    case pi_sim_i: body(sim_i) break;
  }
}
```

Заметим, что мы не определяем семантику портов, а свойства программы задаем в терминах ее переменных. Поэтому декларации для портов отсутствуют в правой части правила для программы.

Правило подстановки. Правило подстановки позволяет применять правила трансляции к частям инструкций языка Reflex. Пусть c, c', c_1, $c_2 \in C$, и $c[c']$ обозначает место вхождения c' в c. Тогда правило подстановки имеет вид:

Если $c_1 \rightsquigarrow c_2$, то $c[c_1] \rightsquigarrow c[c_2]$.

Типы. Поскольку типы языка Reflex совпадают с типами языка VOL, правило трансляции для типов не требуется.

Декларация такта. Пусть k – число. Декларация такта определяется правилом:

TACT k; $\rightsquigarrow$ #define TACT k.

Декларации констант. Пусть u – имя константы, e – Reflex-выражение, и $\varnothing$ – пустой результат трансляции (удаление транслируемой инструкции). Декларации констант определяются правилами:

const u k; $\rightsquigarrow$ #define u k

const u e; $\rightsquigarrow$ #define u (e)

enum {u_1, u_2, ...} $\rightsquigarrow$

#define u_1 0

enum {u_2 = 1, ...}

enum {u_1 = k, u_2, ...} $\rightsquigarrow$

#define u_1 k

enum {u_2 = $k+1$, ...}

enum {} $\varnothing$

Декларации переменных. Декларации переменных определяются правилами:

t u ...; $\rightsquigarrow$ t u;

FROM PROC pj u; $\rightsquigarrow \varnothing$.

Мы не учитываем в дедуктивной верификации связи переменных с портами и уровни доступа переменных. Поэтому, декларации Reflex-переменных непосредственно транслируются в декларации VOL-переменных.

Далее мы считаем, что Reflex-конструкции, для которых определяются правила, встречаются в теле некоторого состояния процесса p_i.

Операции над состояниями. Трансляционная семантика операций над состояниями определяется правилами:

(PROC IN STATE ACTIVE) $\rightsquigarrow$

(PROC p_i IN STATE ACTIVE);

(PROC p_j IN STATE ACTIVE) $\rightsquigarrow$

((p_j_state != STOP_STATE) &&

(p_j_state != ERROR_STATE));

(PROC IN STATE INACTIVE) $\rightsquigarrow$

(PROC p_i IN STATE INACTIVE);

(PROC p_j IN STATE INACTIVE) $\rightsquigarrow$

((p_j_state == STOP_STATE) ||

(p_j_state == ERROR_STATE));

(PROC IN STATE STOP) $\rightsquigarrow$

(PROC p_i IN STATE STOP);

(PROC p_j IN STATE STOP) $\rightsquigarrow$

(p_j_state == STOP_STATE);

(PROC IN STATE ERROR) $\rightsquigarrow$

(PROC p_i IN STATE ERROR);

(PROC p_j IN STATE ERROR) $\rightsquigarrow$

(p_j_state == ERROR_STATE).

Операторы управления состояниями. Операторы управления состояниями включают операторы STOP, ERROR, START, RESTART, SET и NEXT.

Оператор STOP определяется правилами:

STOP; $\rightsquigarrow$ STOP PROC p_i;

STOP PROC p_j; $\rightsquigarrow$

{p_j_timer=0; p_j_state= STOP_STATE;}.

Оператор ERROR определяется правилом:

ERROR; $\rightsquigarrow$

{p_i_timer = 0; p_i_state = ERROR_STATE;}.

Оператор START определяется правилом:

START PROC p_j; $\rightsquigarrow$

{p_j_timer = 0; p_j_state = p_j_s_1;}.

Оператор RESTART определяется правилом:

RESTART $\rightsquigarrow$ START PROC p_i;.

Оператор SET определяется правилом:

SET STATE s_{ij}; $\rightsquigarrow$

{p_i_timer = 0; p_i_state = p_i_s_{ij};}.

Оператор NEXT определяется правилом:

Если p_i_state = s_{ij}, то

SET NEXT; $\rightsquigarrow$

{p_i_timer = 0; p_i_state = $s_{i(j+1)}$;}.

Операторы управления таймаутами. Операторы управления таймаутами включают операторы RESET и TIMEOUT.

Оператор RESET определяется правилом:

RESET TIMEOUT; $\rightsquigarrow$ p_i_timer = 0;.

Пусть st – Reflex-оператор. Оператор TIMEOUT определяется правилом:

TIMEOUT e st $\rightsquigarrow$ if (p_i_timer = e) st.

Доказательство корректности трансляции (эквивалентности Reflex-программы и VOL-программы) не рассматривается в этой работе. Эквивалентность означает функциональную эквивалентность этих программ, где входами обеих программ являются вектора внешних модификаций для каждой Reflex-переменной, а выходами – вектора значений для каждой Reflex-переменной, так же как текущие состояния процессов и значения глобального и локальных таймеров. Определение эквивалентности основывается на операционной семантике аннотированных Reflex-программ и VOL-программ.

Таким образом, мы сводим верификацию временных свойств Reflex-программ к дедуктивной ве-

рификации VOL-программ. Дальше мы опишем правила порождения условий корректности для VOL-программ, примеры порожденных условий корректности для программы управления сушилкой для рук и их доказательство в SMT-решателе Z3.

5. ГЕНЕРАЦИЯ УСЛОВИЙ КОРРЕКТНОСТИ ДЛЯ С-ПРОЕКЦИЙ REFLEX-ПРОГРАММ

Подобно многим другим системам дедуктивной верификации, таким как FramaC [29], Spark [30], KeY [31], Dafny [32], и т.д., наш алгоритм порождения условий корректности определяет предикатный трансформер. Мы используем Z3 чтобы доказать порожденные условия корректности. Специфика алгоритма заключается в том, что он применяется к программе, которая является бесконечным циклом и некоторые переменные этой программы изменяются внешним образом на каждой итерации этого цикла. Алгоритм $sp(A, P)$ рекурсивно вычисляет сильнейшее постусловие [33], выраженное в многосортной логике первого порядка, для программного фрагмента A и предусловия P. Его применение начинается со всей программы и предусловия *True*. Результат его работы – множество условий корректности, сохраненное в переменной *vcs*. Алгоритм использует специальные переменные *vars* и *reached* . Переменная *vars* хранит информацию о переменных и их типах как множество пар вида $x{:}t$, где x – переменная, а t – ее тип. Переменная *reached* хранит множество имен инвариантных аннотаторов, которые были пройдены алгоритмом. Она используется чтобы гарантировать завершение алгоритма. Начальные значения этих переменных – пустые множества.

Мы определяем алгоритм порождения условий корректности *sp* как упорядоченное множество равенств вида $sp(A, P) = [a_1; \ldots; a_n; e]$. Эта нотация означает, что действия $a_1, \ldots, a_n$ последовательно выполняются перед тем, как выражение e вычисляется. Каждое действие a_i вида $v \mathrel{+}= S$ добавляет элементы из множества S в множество v. Равенство $sp(A, P) = e$ является сокращением для $sp(A, P) = [e]$.

Мы используем следующую нотацию в определении алгоритма. Пусть $array(t)$ обозначает тип массивов с элементами типа t. Пусть выражение e имеет тип t, $\{x{:}t, y{:}array(t)\} \subseteq vars$, $\{z{:}t,\ v{:}t\} \cap \cap vars = \varnothing$ для каждого t, и e' – результат преобразования VOL-выражения e в Z3py-выражение. Функция $Store(a, i, v)$ является функцией модификации массива из языка Z3.

В силу простоты языка VOL, алгоритм *sp* имеет следующую компактную форму:

1. $sp(t\ f(t_1 x_1, \ldots, t_n x_n);, P) =$
   $[vars \mathrel{+}= \{x_1{:}t_1, \ldots, x_n{:}t_n, ret\_f{:}t\}; P]$;
2. $sp(t\ x;, P) = [vars \mathrel{+}= \{x{:}t\}; P]$;
3. $sp(\texttt{\#define}\ ce;, P) =$
   $[vars \mathrel{+}= \{c : t\};\ And(P, c == e')]$;
4. $sp(\texttt{havoc}\ y[i];, P) =$
   $[vars \mathrel{+}= \{z{:}t, v{:}t\};$
   $And(P(y \leftarrow z), y == Store(z, i, v))]$;
5. $sp(\texttt{havoc}\ x;, P) =$
   $[vars \mathrel{+}= \{z{:}t\};\ And(P(x \leftarrow z), x == z)]$;
6. $sp(x[i] = e;, P) =$
   $[vars \mathrel{+}= \{z{:}array(t)\};$
   $And(P(y \leftarrow z), y == Store(z, i, e'(y \leftarrow z)))]$;
7. $sp(x = e;, P) =$
   $[vars \mathrel{+}= \{z{:}t\};$
   $And(P(x \leftarrow z), x == e'(x \leftarrow z))]$;
8. $sp(\{B\}, P) = sp(B, P)$;
9. $sp(\texttt{if}\ (e)\ B\ \texttt{else}\ C, P) =$
   $Or(sp(B, And(P, econv(e))),$
   $sp(C, And(P, Not(econv(e)))))$;

10. $sp($`switch (`e`) `l_1`:`B_1 `break; ...`l_n`:`B_n `break;` $, P) =$
$Or(sp(B_1, And(P, e' == l_1)),$
$...,$
$sp(B_n, And(P, e' == l_n)),$
$And(P, e' != l_1, ..., e' != l_n));$
11. $sp($`for(;;) {`B`}`$, P)$ $sp(B$ `for(;;) {`B`}`$, P);$
12. $sp(f(e_1, ..., e_n);, P) =$
$sp(e_1;$ `val_1 := val;` $...e_n;$ `val_n := val;`
`ASSERT` $P_f[x_1 \leftarrow$ `val_1` $, ..., x_n \leftarrow$ `val_n`$];$
`havoc val;`
`ASSUME` $Q_f[x_1 \leftarrow$ `val_1` $, ..., x_n \leftarrow$ `val_n`$];, P);$
13. $sp(e_1 + e_2;, P) =$
$sp(e_1;$ `val_1 := val;` $...e_2;$ `val_2 := val;`
`val :=` $($`val_1 +val_2`$);, P);$

Другие операции определяются аналогичным образом. Заметим, что, для простоты, мы не рассматриваем приведение типов и считаем, что операции над числами в идеальной арифметике и машинной арифметике совпадают. Более точное определение операционной и аксиоматической семантик для операций может быть найдено в [34, 35].

14. $sp($`ASSUME` $e;, P) = And(P, e);$
15. $sp($`ASSERT` $e;, P) =$
$[vcs$ += $\{$Im $plies(P, e)\};$ $And(P, e)];$
16. если $l \notin reached$, то $sp($`INV` $le;$ $A, P) =$
$[reached$ += $\{l\};$ vcs += $\{Implies(P, e)\};$ $sp(A, e)];$
17. если $l \in reached$, то $sp($`INV` $le;$ $A, P) =$
$[vcs$ += $\{Implies(P, e)\};$ $e];$
18. $sp(stA, P) = sp(A, sp(st, P)).$

Этот алгоритм завершается, так как каждый вложенный вызов *sp* выполняется на меньшем программном фрагменте во всех случаях кроме `for(;;)` (случай 11), и, согласно случаю 16, алгоритм может проходить инвариантный аннотатор в начале тела цикла `for(;;)` только один раз.

Вычисление условий корректности для пути аннотированной программы, управляющей сушилкой для рук (Листинг 2), начинающегося в точке `#define TACT 100` и заканчивающегося в точке `INV` *lab inv*; имеет в качестве результата:

- $vcs = \{f\}$, где условие корректности f имеет вид:

```
Implies(
  And(True, TACT == 100, ON == True,
    OFF == False, STOP_STATE == 0,
    ERROR_STATE == 1,
    Ctrl_Waiting == 2,
    Ctrl_Drying == 3, timer == 0,
    Ctrl_state == Ctrl_Waiting,
    Ctrl_timer == 0, dryer[0] == OFF,
    True),
  inv);
```

- множество *vars* включает следующие типизированные переменные и константы:

```
TACT:int, ON:bool, OFF:bool, val:value,
STOP_STATE:int, ERROR_STATE:int,
Ctrl_Waiting:int, Ctrl_Drying:int,
Ctrl_state:int, Ctrl_timer:int,
dryer:array(bool), hands:array(bool),
timer:int, timer_1:int,
Ctrl_state_1:int, Ctrl_timer_1:int;
```

- $reached = \{lab\}$.

Другие семь условий корректности, начинающиеся в точке INV *lab inv*; и заканчивающиеся в той же самой точке и соответствующие различным ветвям оператора switch и операторов if, порождаются аналогичным образом. Все порожденные условия корректности успешно доказаны в Z3py.

Порождение условий корректности для VOL-программ – результатов трансляции аннотированных Reflex-программ – и их доказательство

завершают описание нашего метода дедуктивной верификации Reflex-программ.

6. ЗАКЛЮЧЕНИЕ

В этой работе мы предложили метод дедуктивной верификации Reflex-программ. Этот метод включает языки аннотаций и аннотаторов для Reflex-программ, ориентированные на описание временных свойств этих программ, алгоритм трансляции аннотированной Reflex-программы в VOL-программу, язык аннотаторов и алгоритм порождения условий корректности для VOL-программ.

Наш метод верификации имеет несколько отличительных свойств. Во-первых, он моделирует взаимодействие между Reflex-программой и управляемым ею объектом через входные и выходные порты, связанные с программными переменными. Инструкция havoc языка VOL позволяет моделировать запись значений из входных портов в программные переменные. Эти внешние значения переменных ограничиваются аннотатором ASSUME. Проверка значений, читаемых из программных переменных в выходные порты, специфицируется аннотаторами ASSERT и INVARIANT. Во-вторых, этот метод сводит верификацию ряда временных свойств Reflex-программ к дедуктивной верификации императивных программ за счет явного моделирования времени в VOL-программах – результатах трансляции Reflex-программ – с помощью глобальных и локальных таймеров, а также историй значений Reflex-переменных, представленных массивами. В-третьих, наш алгоритм порождения условий корректности может применяться к бесконечным циклам, свойственным системам управления.

Имеются несколько направлений для дальнейшего развития метода. Мы планируем расширить его на текстовые языки стандарта IEC 61131-3. Подобно языку Reflex, эти языки используются для программ, которые взаимодействуют с объектом управления только между программными циклами.

Другое направление состоит в исследовании новых временных свойств, для которых верификация также может быть сведена к дедуктивной верификации. Особенно нас интересуют временные аспекты, связанные с историей значений состояний процессов и локальных таймеров процессов, которые позволили бы оценить производительность Reflex-программ.

Явное моделирование времени в Reflex-аннотациях не очень удобный способ описания временных свойств Reflex-программ. Мы планируем использовать временные логики (LTL, CTL and MTL) и их расширения чтобы описывать эти свойства более естественным образом и разработать алгоритм трансляции таких описаний в формулы с явным моделированием времени. Чтобы сделать эту задачу выполнимой, мы планируем использовать специализированные онтологические паттерны [36] вместо произвольных формул этих логик.

В дополнение к решателю Z3 мы намерены использовать другие средства машинной поддержки доказательства чтобы расширить класс проверяемых свойств. В частности, решатель Z3 не смог доказать свойство, что сушилка будет работать в течение не менее 10 секунд после удаления рук, поскольку это свойство требует расширенной индукции. Интерактивный доказатель теорем ACL2 [37] с более развитыми индукционными схемами является хорошим кандидатом для решения этой проблемы.

Чтобы упростить доказательство корректности алгоритмов трансляции и порождения условий корректности, мы планируем использовать унифицированное графо-онтологическое представление для программ на языках Reflex и VOL. Такое представление позволяет свести трансляционную семантику к трансформационной в рамках одного общего графо-онтологического языка и применить онтологический подход для разработки операционной семантики этих языков [38].

7. БЛАГОДАРНОСТИ

Исследование выполнено в рамках темы госзадания ИАиЭ СО РАН (№ АААА-А19-119120290056-0) и при финансовой поддержке РФФИ в рамках научных проектов 17-07-01600 и 20-01-00541.

Acknowledgments: The reported study was funded by State budget of the Russian Federation (IAE project No. AAAA-A19-119120290056-0), by RFBR, project number 17-07-01600 and project number 20-01-00541, and by BMBF (Germany) within the project HPC2SE.

СПИСОК ЛИТЕРАТУРЫ

1. *Blanke M., Kinnaert M., Lunze J., Staroswiecki M.* Diagnosis and Fault-Tolerant Control. 2nd edn. Springer-Verlag Berlin, Heidelberg. 2006.
2. IEC 61131-3: Programmable controllers Part 3: Programming languages. Rev. 2.0. Intern. Electrotechnical Commission Std. 2003.
3. *Basile F., Chiacchio P., Gerbasio D.* On the Implementation of Industrial Automation Systems Based on PLC // IEEE Transactions on Automation Science and Engineering, 2013. V. 4. № 10. P. 990–1003.
4. *Thramboulidis K., Frey G.* An MDD Process for IEC 61131-based Industrial Automation Systems // 16th IEEE Intern. Conf. on Emerging Technologies and Factory Automation (ETFA11), Toulouse, France, 2011. P. 1–8.
5. IEC 61499: Function Blocks for Industrial Process Measurement andControl Systems. Parts 1–4. Rev. 1.0. Intern. Electrotechnical Commission Std., 2004–2005.
6. *Wagner F., Schmuki R., Wagner T., Wolstenholme P.* Modeling Software with Finite State Machines. Auerbach Publications. USA, Boston, MA. 2006.

7. *Samek M.* Practical UML statecharts in C/C++: event-driven programming for embedded systems. 2nd edition. 2009. Newnes, Oxford.
8. Control Technology Corporation. QuickBuilder™Reference Guide. 2018. https://controltechnologycorp.com/docs/QuickBuilder_Ref.pdf. Last accessed 20 Jan 2019
9. *Zyubin V.E.* Hyper-automaton: A Model of Control Algorithms // Proceedings of the IEEE Intern. Siberian Conf. on Control and Communications (SIBCON-2007). The Tomsk IEEE Chapter & Student Branch. Tomsk, Russia, 2007. P. 51–57.
10. *Hoare C.A.R.* Communicating Sequential Processes. Prentice-Hall Int., 1985.
11. *Harel D.* Statecharts: a Visual Formalism for Complex Systems // Science of Computer Programming, 1987. V. 8. № 3. P. 231–274.
12. *Lynch N., Tuttle M.* An Introduction to Input/Output Automata // CWI Quarterly, 1989. V. 2. № 3. P. 219–246.
13. *Berry G.* The Foundations of Esterel // Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, Foundations of Computing Series, 2000. P. 425–454.
14. *Henzinger T.A.* The Theory of Hybrid Automata // Inan M.K., Kurshan R.P. (eds) Verification of Digital and Hybrid Systems, NATO ASI Series (Series F: Computer and Systems Sciences), Springer, Berlin, Heidelberg, 2000. V. 170. P. 265–292.
15. *Milner R.* Communication and Concurrency. Series in Computer Science. Prentice Hall, New Jersey. 1989.
16. *Kaynar D.K., Lynch N., Segala R., Vaandrager F.* Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems // 24th IEEE Intern. Real-Time Systems Symposium (RTSS'03), 2003, IEEE Computer Society Cancun, Mexico. P. 166–177.
17. *Kof L., Schätz B.*: Combining Aspects of Reactive Systems // Proc. of Andrei Ershov Fifth Int. Conf. Perspectives of System Informatics, Novosibirsk, 2003. P. 239–243.
18. *Zyubin V.* SPARM Language as a Means for Programming // Microcontrollers. Optoelectronics, Instrumentation, and Data Processing, 1996. V. 2. № 7. P. 36–44.
19. *Liakh T.V., Rozov A.S., Zyubin V.E.* Reflex Language: a Practical Notation for Cyber-Physical Systems // System Informatics, 2018. V. 12. № 6. P. 85–104.
20. *Rozov A.S., Zyubin V.E.* Process-oriented programming language for MCU-based automation // Proc. of the IEEE Intern. Siberian Conf. on Control and Communications, The Tomsk IEEE Chapter Student Branch, Tomsk, Russia, 2013. P. 1–4.
21. *Bulavskij D., Zyubin V., Karlson N., Krivoruchko V., Mironov V.* An Automated Control System for a Silicon Single-Crystal Growth Furnace // Optoelectronics, instrumentation, and data processing, 1996. V. 2. № 5. P. 25–30.
22. *Travis J., Kring J.* LabVIEW for Everyone: Graphical Programming Made Easy and Fun. 3rd Edition. Prentice Hall PTR, Upper Saddle River, NJ, USA. 2006.
23. *Zyubin V.* Using Process-Oriented Programming in LabVIEW // In: Proc. of the Second IASTED Intern. Multi-Conference on "Automation, control, and information technology": Control, Diagnostics, and Automation.Novosibirsk, 2010. P. 35–41.
24. *Randell B.* Software Engineering Techniques // Report on a conference sponsored by the NATO Science Committee. Brussels, Scientific Affairs Division, NATO, Rome, Italy, 1970. P. 16.
25. Z3 API in Python, https://ericpony.github.io/z3py-tutorial/guide-examples.htm Last accessed 20 Jan 2019
26. *Moura L., Bjurner N.*: Z3: An Efficient SMT Solver. TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, 2008. V. 4963. P. 337–340.
27. *Anureev I.S., Garanina N.O., Liakh T.V., Rozov A.S., Zyubin V.E., Gorlatch S.* Two-Step Deductive Verification of Control Software Using Reflex // Proceedings of A.P. Ershov Informatics Conference (PSI-19). A.P. Ershov Institute of Informatics Systems: IPC NSU, Novosibirsk, Russia, LNCS, 2019. V. 11964. P. 50–63.
28. *Barnett M., Chang B.-Y.E., DeLine R., Jacobs B., Leino K.R.M.* Boogie: A Modular Reusable Verifier for Object-Oriented Programs // In: Proc. of the 4th Intern. Conf. on Formal Methods for Components and Objects, LNCS. 2005. V. 4111. P. 364–387.
29. FramaC Homepage, https://frama-c.com/
30. Spark Pro Homepage, https://www.adacore.com/sparkpro
31. The KeY project Homepage, https://www.key-project.org/
32. Dafny Homepage, https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/.
33. *Dijkstra E.W., Scholten C.S.* Predicate Calculus and Program Semantics, Springer-Verlag, 1990.
34. *Nepomniaschy V., Anureev I., Mikhailov I., Promsky A.* Towards verification of C programs. C-light language and its formal semantics // Programming and Computer Science, 2002. V. 28. № 6. P. 314–323.
35. *Nepomniaschy V., Anureev I., Promsky A.* Towards verification of C programs: Axiomatic semantics of the C-kernel language // Programming and Computer Science, 2003. V. 29. № 6. P. 338–350.
36. *Garanina N., Zyubin V., Lyakh V., Gorlatch S.* An Ontology of Specification Patterns for Verification of Concurrent Systems // In: New Trends in Intelligent Software Methodologies, Tools and Techniques // Proceedings of the 17th International Conference SoMeT-18, Series: Frontiers in Artificial Intelligence and Applications, Amsterdam: IOS Press, 2018. P. 515–528.
37. ACL2 Homepage, http://www.cs.utexas.edu/users/moore/acl2/
38. *Anureev I.S.* Operational ontological approach to formal programming language specification // Programming and Computer Software, 2009. V. 35. № 1. P. 35–42.

ПРОГРАММИРОВАНИЕ, 2020, № 4, с. 27–35

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.42

ИСПОЛЬЗОВАНИЕ СИСТЕМЫ РАЗНОРОДНЫХ ПАТТЕРНОВ ОНТОЛОГИЧЕСКОГО ПРОЕКТИРОВАНИЯ ДЛЯ РАЗРАБОТКИ ОНТОЛОГИЙ НАУЧНЫХ ПРЕДМЕТНЫХ ОБЛАСТЕЙ

© 2020 г. Ю. А. Загорулько[a,*], О. И. Боровикова[a,]**

[a] *Институт систем информатики им. А.П. Ершова СО РАН*
630090 Новосибирск, пр. Лаврентьева, 6, Россия

**E-mail: zagor@iis.nsk.su*

***E-mail: olesya@iis.nsk.su*

Поступила в редакцию 18.02.2020 г.
После доработки 10.03.2020 г.
Принята к публикации 21.03.2020 г.

В статье представлен подход к разработке онтологий научных предметных областей, базирующийся на применении системы разнородных паттернов онтологического проектирования, которые представляют собой документально зафиксированные описания проверенных на практике решений типовых проблем онтологического моделирования. В эту систему входят как универсальные паттерны, предназначенные для описания типовых конструкций онтологии, так и паттерны, ориентированные на представление научных знаний. Применение такой системы паттернов позволяет не только обеспечить согласованное представление всех сущностей онтологии научной предметной области и тем самым избежать многих ошибок онтологического моделирования, но и сэкономить ресурсы, затрачиваемые на разработку этой онтологии.

DOI: 10.31857/S0132347420040068

1. ВВЕДЕНИЕ

В связи с проникновением технологий Semantic Web практически во все сферы человеческой деятельности появилась насущная потребность в формализации знаний различных предметных областей в виде онтологий. Особенно остро стоит проблема построения онтологий научных предметных областей (НПО), к которым обычно относят предметные области, охватывающие некоторую научную дисциплину или область научных знаний во всех ее аспектах.

Для разработки онтологий применяются различные методологии и подходы [1–4]. В последнее время интенсивно развивается подход, использующий паттерны онтологического проектирования (Ontology Design Patterns) или паттерны ОП (ODPs) [5–8]. Согласно этому подходу паттерны ОП представляют собой документально зафиксированные описания проверенных на практике решений типовых проблем онтологического моделирования. Несмотря на то, что использование паттернов ОП позволяет сэкономить человеческие ресурсы и повысить качество разрабатываемых онтологий, они пока не нашли широкого практического применения из-за ряда проблем, возникающих при их использовании.

В статье дается краткий обзор паттернов онтологического проектирования, анализируются проблемы их использования, описывается предложенный авторами подход к построению онтологий научных предметных областей, базирующийся на паттернах ОП. Паттерны ОП, используемые в данном подходе, появились в результате решения проблем онтологического моделирования, с которыми авторы статьи столкнулись в процессе разработки онтологий для различных научных предметных областей [9, 10].

2. КРАТКИЙ ОБЗОР ПАТТЕРНОВ ОНТОЛОГИЧЕСКОГО ПРОЕКТИРОВАНИЯ

Паттерны онтологического проектирования имеют в качестве своих прародителей шаблоны проектирования (design pattern), широко используемые в разработке программного обеспечения [11]. Аналогично шаблонам проектирования, паттерны ОП предназначены для описания решений типичных проблем, возникающих при разработке онтологий [7].

В зависимости от проблем, для решения которых предназначены паттерны онтологического проектирования, различают структурные паттерны (Structural ODPs), паттерны соответствия

(Correspondence ODPs), паттерны содержания (Content ODPs), паттерны логического вывода (Reasoning ODPs), паттерны представления (Presentation ODPs) и лексико-синтаксические паттерны (Lexico-Syntactic ODPs). (Заметим, что эта типология паттернов была предложена в рамках проекта NeOn [12].)

Из всех перечисленных типов паттернов при разработке онтологий инженерами знаний используются в основном структурные паттерны, паттерны содержания и паттерны представления. Рассмотрим их подробнее.

Существует два типа структурных паттернов: логические и архитектурные. Логические паттерны (Logical ODPs) фиксируют способы решения проблем, вызванных ограничениями выразительных возможностей языков описания онтологий. Архитектурные паттерны (Architectural ODPs) задают общий вид онтологии: набор логических паттернов, из которых может строиться онтология, или ее модульную архитектуру, заданную в виде сети онтологий, каждая из которых играет роль модуля.

Паттерны содержания задают способы представления типовых фрагментов онтологий, на основе которых могут строиться онтологии различных предметных областей.

Паттерны представления определяют правила (рекомендации) именования и аннотирования различных сущностей онтологии. Применение этих правил должно повысить читаемость онтологии, а также удобство и простоту ее использования.

В настоящее время создано и развивается несколько каталогов паттернов ОП [13–15]. Наиболее представительный из них размещен на портале Ассоциации ODPA (Association for Ontology Design & Patterns) [13], созданном в рамках проекта NeOn [12].

Паттерны ОП чаще всего описываются в формате, предложенном на портале ассоциации ODPA [13]. Согласно этому формату описание паттерна включает сведения о его авторе и области применения, его графическое представление, текстовое описание, набор сценариев, примеры использования и ссылки на другие паттерны. Паттерны содержания также могут снабжаться набором вопросов компетенции (Competency questions) [6, 7], которые могут использоваться как при разработке паттернов, так и при разработке конкретной онтологии для поиска нужных паттернов.

3. ПРОБЛЕМЫ ИСПОЛЬЗОВАНИЯ ПАТТЕРНОВ ОНТОЛОГИЧЕСКОГО ПРОЕКТИРОВАНИЯ

Первая проблема повторного использования паттернов обусловлена их сложностью – зачастую разработчику новой онтологии трудно понять семантику, которую заложили в паттерн его авторы. В последнее время наблюдается тенденция к упрощению паттернов [16]. Появились даже, так называемые, метапаттерны, которые описывают очень простые сущности [17]. Однако такие простые паттерны не могут существенно облегчить построение онтологий НПО.

Другая проблема вызвана отсутствием удобных инструментов разработки онтологий, поддерживающих использование паттернов ОП. Здесь можно отметить плагины для инструмента разработки онтологий проекта NeOn [12] и редактора онтологий WebProtégé [18]. Однако первый плагин доступен только для участников проекта NeOn, а второй может использоваться только в редакторе WebProtégé, который не очень популярен среди разработчиков онтологий из-за его ограниченной функциональности (по сравнению с настольной версией).

Третья проблема состоит в том, что паттерны описываются и применяются отдельно и не составляют единой системы.

С этой проблемой перекликается четвертая проблема, связанная с отсутствием систематизированных наборов паттернов, ориентированных на специалистов в предметной области. Существующие каталоги паттернов не отвечают этому требованию.

Что касается наличия в широком доступе паттернов, которые можно было бы использовать при разработке онтологий НПО, то упомянутые выше каталоги лишь частично покрывают потребности построения онтологий научных областей.

4. ПОДХОД К РАЗРАБОТКЕ ОНТОЛОГИЙ НАУЧНЫХ ПРЕДМЕТНЫХ ОБЛАСТЕЙ

Рассмотрим подход, обеспечивающий повторное использование паттернов ОП при построении онтологий НПО. Данный подход предлагает систему разнородных паттернов ОП [19], реализованных на языке OWL [20], а также методы и средства, поддерживающие их совместное использование при разработке онтологий НПО.

Заметим, что разнородность паттернов ОП выражается не только в их разнообразии по типам, но и по назначению и предметным областям, для которых они создаются.

На данный момент используются четыре типа паттернов: структурные паттерны, паттерны содержания, паттерны представления и лексико-синтаксические паттерны. При этом одна часть этих паттернов является универсальной, другая часть ориентирована на представление научных знаний.

Важной особенностью данного подхода является использование базовых (ядерных) онтологий, включающих только самые общие сущности, не зависящие от конкретной НПО. Большая

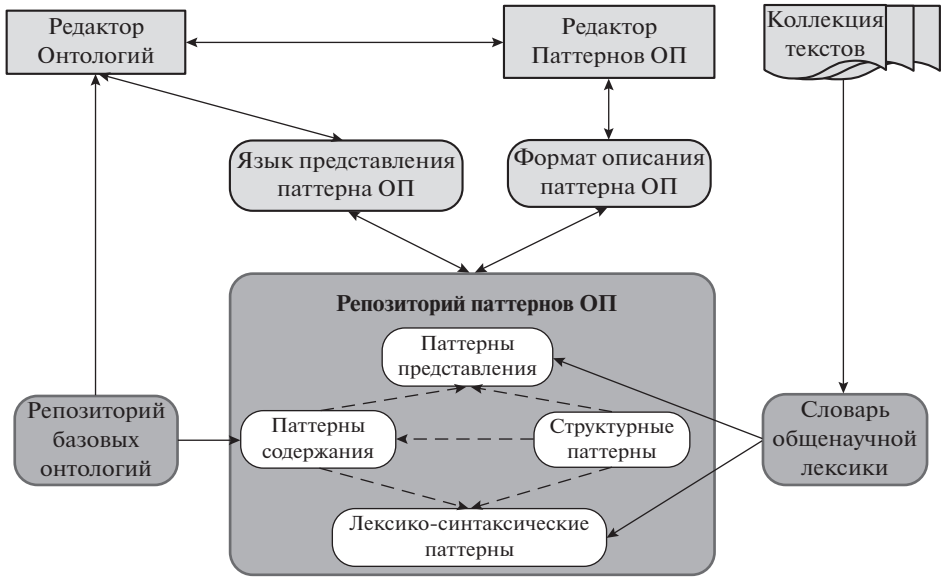


Рис. 1. Система автоматизированного построения онтологий НПО на основе разнородных паттернов онтологического проектирования.

часть таких сущностей представлена паттернами содержания.

Перечисленные выше средства образуют систему автоматизированного построения онтологий НПО, которая состоит из следующих компонент (рис. 1):

• *репозитория паттернов ОП*, включающего паттерны ОП, представленные на одном из языков описания паттернов или в соответствующем формате;

• *форматов (способов) представления* паттернов ОП всех типов, имеющихся в системе;

• *набора языков для описания паттернов ОП разных типов*;

• *репозитория базовых онтологий*, на основе которых строятся онтологии конкретных НПО;

• *словаря* общенаучной лексики, содержащего лексику, характерную для большинства научных предметных областей;

• *редакторов онтологий и паттернов ОП*, служащих для построения онтологии НПО и специализации паттернов.

4.1. Структура и содержание онтологии НПО и базовых онтологий

Онтология любой НПО содержит не только описания присущих ей системы понятий, задач и методов обработки и анализа информации, но и описания релевантных ей информационных ресурсов. В связи с этим онтологию НПО можно представить в виде системы взаимосвязанных онтологий, отвечающих за представление указанных выше трех компонентов знаний, а именно: онтологии области знаний, онтологии задач и методов, а также онтологии научных интернет-ресурсов.

Онтология области знаний задает систему понятий и отношений, предназначенных для детального описания моделируемой НПО и выполняемой в ее рамках научной и исследовательской деятельности. Онтология задач и методов описывает задачи, решаемые в данной НПО, и методы их решения. Онтология научных интернет-ресурсов служит для описания, представленных в сети Интернет информационных ресурсов, релевантных данной НПО.

Так как разработка онтологии НПО “с нуля” является непростой задачей, предложен метод ее построения на основе небольшого, но представительного набора базовых онтологий, в который входят: (1) онтология научного знания, (2) онтология научной деятельности, (3) базовая онтология задач и методов, (4) базовая онтология информационных ресурсов.

Как было сказано выше, все базовые онтологии имеют спецификации на языке OWL [20].

Онтология научного знания содержит классы, задающие структуры для описания понятий, входящих в любую НПО. Такими понятиями являются Раздел науки, Объект исследования, Предмет исследования, Метод исследования, Научный результат и др.

Онтология научной деятельности включает классы понятий, относящиеся к организации научно-исследовательской деятельности, такие как Персона, Организация, Событие, Научная деятельность, Проект, Публикация и др.

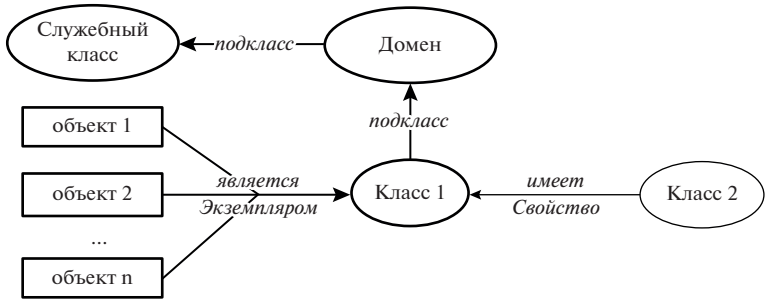


Рис. 2. Структурный паттерн представления области допустимых значений.

Базовая онтология информационных ресурсов включает класс Информационный ресурс в качестве основного класса. Набор атрибутов и связей этого класса основан на стандарте Dublin core [21].

С помощью понятий и отношений базовой онтологии задач и методов могут быть описаны задачи, решаемые в данной НПО, методы их решения и реализующие их программные компоненты и алгоритмы.

4.2. Система паттернов онтологического проектирования

В состав репозитория паттернов ОП включены паттерны четырех типов: структурные логические паттерны, паттерны содержания, паттерны представления и лексико-синтаксические паттерны.

Паттерны представления задают принятые в рассматриваемом подходе правила именования и аннотирования элементов онтологии, близкие к правилам, принятым в сообществе онтологического моделирования [22]. Кроме того, с помощью этих паттернов могут задаваться способы визуализации основных сущностей онтологии (расширенного описания объектов онтологии) [23].

Необходимость в использовании структурных логических паттернов возникла из-за отсутствия в языке OWL выразительных средств для представления сложных сущностей и конструкций, актуальных при построении онтологий НПО, в частности, областей допустимых значений, многоместных и атрибутированных отношений (бинарных отношений с атрибутами).

Структурный паттерн для представления области допустимых значений предназначен для задания таких конструкций, которые в реляционной модели данных [24] называются доменами и характеризуются названием и множеством элементарных значений. Домены удобно использовать при описании возможных значений свойств класса, когда весь набор таких значений известен заранее. В этом паттерне домен задается перечислимым классом, который является наследником специально введенного служебного класса *Домен* и состоит из конечного набора различных индивидов (объектов), определяющих возможные значения некоторого свойства (см. рис. 2).

Примерами таких доменов являются “Географический тип”, “Должность”, “Тип организации”, “Тип публикации”, которые включают соответственно виды населенных пунктов, виды должностей, типы организаций и публикаций.

Заметим, что на приведенных в статье рисунках паттернов классы обозначаются в виде эллипсов, а индивиды и атрибуты – в виде прямоугольников. Связь типа ObjectProperty (отношение) показывается сплошной прямой линией, а связь типа DataProperty (атрибут) – прерывистой. При этом классы, индивиды и атрибуты, которые должны обязательно присутствовать в паттерне, представлены заштрихованными и/или обведенными жирной линией фигурами. Жирной линией также представляются обязательные связи.

Для представления атрибутированных отношений предложен структурный логический паттерн, показанный в левой части рис. 3.

Центральное место в данном паттерне занимает служебный класс *Отношение с атрибутами*, с которым связываются базовые классы, моделирующие аргументы бинарного отношения, посредством связей *являетсяАргументом1* и *имеетАргумент2*. При этом атрибуты бинарного отношения моделируются свойствами данного класса (в нотации языка OWL либо *DataProperty*, либо *ObjectProperty*) *имеетАтрибут* и *имеетАтрибутИзДомена*. Для представления конкретного типа отношения вводится новый класс, являющийся наследником класса *Отношение с атрибутами*.

На основе паттерна атрибутированного отношения строятся путем его специализации паттерны для представления отношений, служащих для описания участия персон или организаций в каких-то событиях и деятельностях, для описания факта работы персоны в организации на определенной должности и в определенный период времени и т.п.

В правой части рис. 3 приведен пример специализации паттерна для описания участия персо-

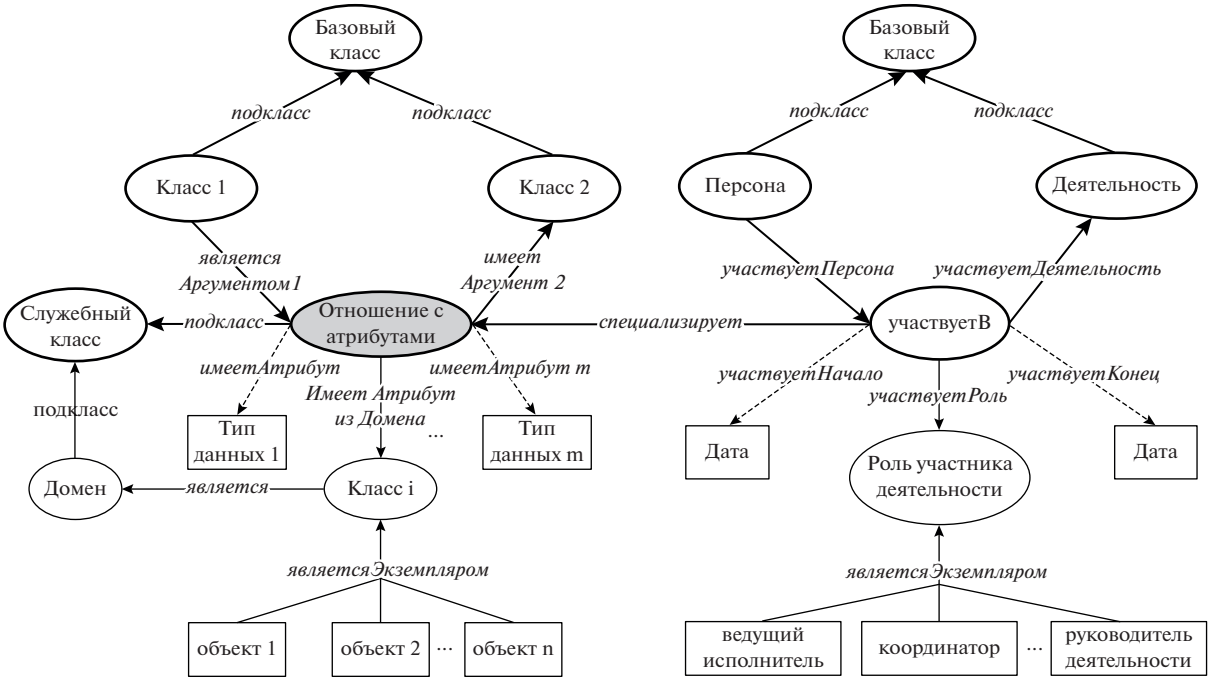


Рис. 3. Структурный паттерн направленного бинарного атрибутированного отношения и пример его специализации.

ны в научной деятельности. Класс *Персона* выступает в качестве первого аргумента, класс *Деятельность* является вторым аргументом. Для каждого экземпляра класса *участвуетВ* с помощью свойства *FunctionalProperty* накладываются требования к указанию только одной даты начала участия и одной даты завершения участия, а путем накладывания ограничений на свойства *участвуетПерсона, участвуетДеятельность, являетсяАргументом1* и *имеетАргумент2* описываются требования на количество и тип аргументов отношения.

Подобным образом строится паттерн для представления многоместного отношения.

Для единообразного и непротиворечивого представления используемых в НПО понятий и их свойств были разработаны паттерны содержания, описывающие основные понятия базовых онтологий. Благодаря этому, разработка онтологии конкретной НПО в основном состоит в специализации паттернов содержания и построении на их основе фрагментов целевой онтологии.

В качестве примера приведем паттерн, предназначенный для описания прикладных задач, решаемых в рамках научной предметной области (см. рис. 4).

С этим паттерном связывается следующий набор вопросов компетенции:

Какими методами решается задача?

Какие данные используются для решения задачи?

Что является результатом решения задачи?

К какому разделу науки относится задача?

Кто формулирует задачу?

и др.

Следует заметить, что входящие в предлагаемый набор паттерны содержания связаны между собой через общие понятия и отношения и тем самым образуют единую семантическую сеть содержательных паттернов.

Например, представленные на рис. 5 паттерны содержания, описывающие понятия *Деятельность* и *Персона*, связаны между собой не только атрибутированным отношением *участвуетВ*, но и через понятия *Организация, Публикация, Метод исследования и Научный результат*. (Заметим, что атрибутированные отношения *участвуетВ* и *работаетВ* показаны на рисунке пунктирной линией.)

Лексико-синтаксические паттерны [25, 26] задают отображения между языковыми структурами (фрагментами текста) и элементами онтологии и применяются для автоматизации построения и пополнения онтологий на основе текстов на естественном языке (ЕЯ) [27]. Элементами лексико-синтаксических паттернов являются группы слов и словосочетаний ЕЯ, которые представлены в словаре общенаучной лексики и соответствуют описаниям онтологических конструкций, заданным как в языке описания онтологий, так и в структурных логических паттернах, паттернах содержания и представления.

Для построения и пополнения онтологий для каждого паттерна содержания и структурного паттерна разрабатывается или автоматически ге-

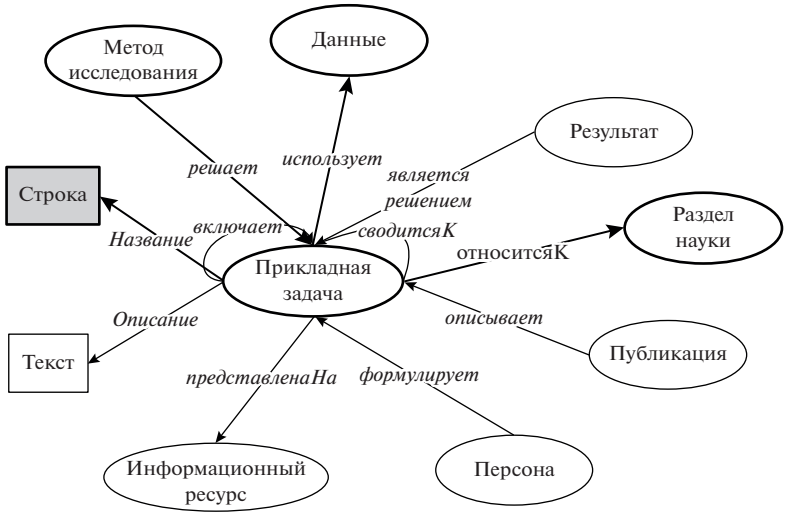


Рис. 4. Паттерн для описания прикладной задачи.

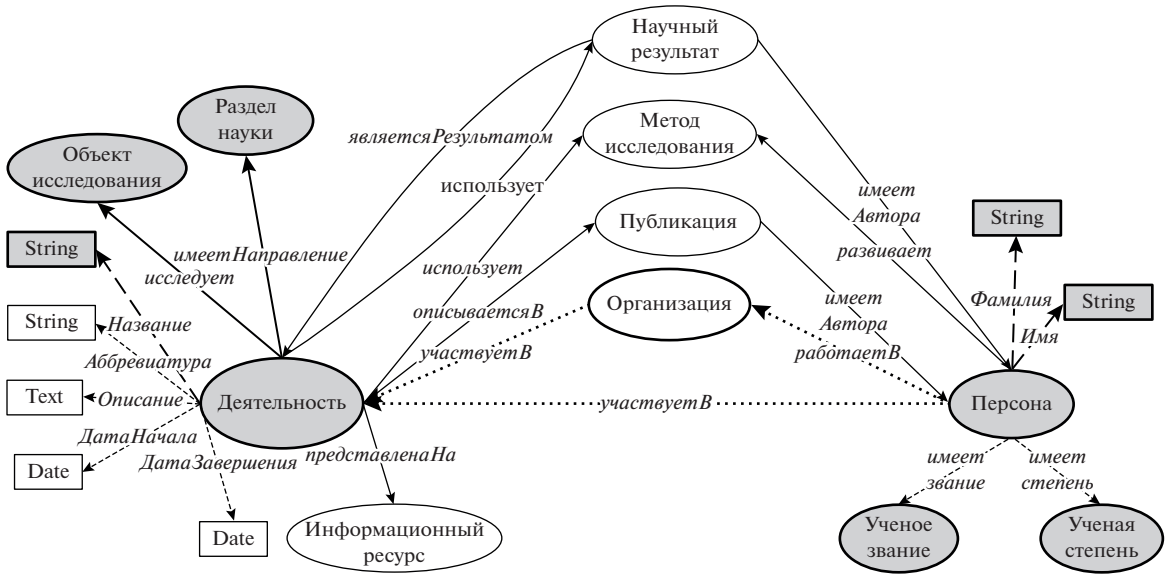


Рис. 5. Фрагмент сети паттернов.

нерируется свой набор лексико-синтаксических паттернов.

Так для паттерна содержания, описывающего *Метод исследования*, разработаны следующие лексико-синтаксические паттерны, предназначенные для извлечения из текста информации о свойстве *название* экземпляра класса *Метод исследования* [28]:

```
NMi = NP1< метод, c=acc> [.] VPa <называть> [.] NP2 <c=ins> <NP1.n = NP2.n>
```

```
NMj = VPaP <предлагать> [.] NP1< метод > [.] VPa <называть> [.] NP2 <c=ins> <NP1.n = NP2.n>
```

Эти паттерны представлены на модифицированной версии языка, предложенного в работе [29], и используют предварительно заданные шаблоны именной группы (*NP*), глагольной лексемы, включающей личные формы глагола и причастия (*VPa=V|Pa*), и глагольной группы (*VP* и *VPaP*). Здесь *PnP* обозначает личное местоимение, точка обозначает разрыв в фрагменте текста, в квадратные скобки помещается факультативный элемент.

4.3. Метод построения онтологии НПО

Построение онтологии конкретной НПО включает три основных этапа:

1. Построение компонентов онтологии НПО на основе базовых онтологий путем их достройки и развития. Этот этап включает специализацию представленных в базовых онтологиях паттернов содержания на конкретную НПО.

2. Пополнение онтологии НПО путем конкретизации (означивания) структурных паттернов и паттернов содержания, представленных в базовых онтологиях или полученных из таких паттернов путем их специализации на конкретную НПО.

3. Построение и пополнение онтологии НПО на основе лексико-синтаксических паттернов.

При этом составная часть онтологии НПО – онтология области знаний – строится на основе онтологий научного знания и научной деятельности, онтология задач и методов – на основе базовой онтологии задач и методов, онтология научных интернет-ресурсов – на основе базовой онтологии интернет-ресурсов.

Использование паттернов содержания для пополнения онтологии НПО поддерживается специальным редактором данных [30], который позволяет специалистам в предметной области пополнять онтологию фактическими данными – объектами классов и их свойствами. При пополнении онтологии пользователь из представленной ему иерархии классов онтологии выбирает нужный класс, редактор по имени класса находит соответствующий паттерн и на его основе строит форму, содержащую поля для заполнения свойств объекта этого класса пользователем. При этом редактор может интерпретировать отношения с атрибутами, описанные паттерном на рис. 2. Благодаря этому пользователь может работать с задаваемыми с помощью таких отношений свойствами объекта как с “обычными” object properties. Отличие будет состоять в необходимости задания в отдельном окне значений атрибутов такого отношения.

5. БЛИЗКИЕ РАБОТЫ

Хотя исследования по разработке паттернов ОП ведутся уже много лет [5], но задумываться о систематизации паттернов ОП и их эффективном практическом применении (например, как лучше приспособить паттерны к использованию в научных и индустриальных проектах) стали не так давно [31]. Каталоги паттернов ОП начали создавать, когда накопилась их критическая масса. Вскоре выяснилось, что одного только формата описания паттернов для ведения каталога недостаточно, поэтому появились работы по систематизации паттернов и организации их совместного использования [32–34].

В работе [32] вводится так называемый язык онтологических паттернов (Ontology Pattern Language или OPL) в виде сети взаимосвязанных паттернов содержания, ориентированных на определенную предметную область, вместе с процедурными правилами, описывающими порядок, в котором они могут комбинироваться при построении онтологии. Паттерны должны быть закодированы на языке представления знаний высокого уровня. В данном подходе предлагается использовать для этого язык OntoUML [35].

Второй подход [33] предлагает простой язык для описания паттернов ОП, который совместим с существующими стандартами и инструментами разработки онтологий (в частности, с языком OWL и редактором Protégé) и может быть расширен до более сложной парадигмы представления паттернов ОП.

Главной идеей подхода является использование аннотационных свойств языка OWL (OWL Annotation Properties) для идентификации паттернов и описания отношений между ними и элементами и модулями онтологии, в которых они используются. Средства для такого аннотирования представляются в онтологии, разработанной в рамках данного подхода и получившей название OPLa. С помощью описанных такими средствами аннотаций обеспечивается документирование использования паттернов и модулей не только во время разработки онтологии, но и после нее. Помимо того, что такие аннотации хранят записи о происхождении паттернов, они также облегчают поиск конкретных паттернов, модулей и онтологий и упрощают процесс выравнивания онтологий.

В следующем подходе предлагается библиотека OTTR (Reasonable Ontology Templates) [34], которая предоставляет разработчикам онтологий паттерны ОП в виде высокоуровневых OWL-макросов. В сущности, такие паттерны представляют собой n-арные отношения, которые связывают простой табличный формат ввода, определяемый заголовком паттерна, с богатой онтологической структурой, представленной в его теле, в том числе, с помощью композиции других паттернов. На основе заголовка паттерна могут быть созданы различные форматы ввода табличных данных, что позволяет инженерам знаний разрабатывать шаблоны, ориентированные на сбор знаний от экспертов заданной предметной. При описании паттернов на языке OWL используется специальный словарь OTTR OWL.

Что касается работ по созданию наборов или систем паттернов, ориентированных на определенный класс предметных областей, то их не так много [36]. Наибольший прогресс наблюдается в создании систем паттернов ОП, предназначенных для построения онтологий в области биоинформатики, биологии и медицины [37–39].

6. ЗАКЛЮЧЕНИЕ И БУДУЩИЕ РАБОТЫ

В статье рассмотрены проблемы применения паттернов онтологического проектирования для разработки онтологий научных предметных областей. Представлен подход к разработке онтологий НПО, решающий большинство из этих проблем. Данный подход поддерживается системой разнородных паттернов онтологического проектирования, описывающих основные конструкции и сущности, необходимые для описания научных предметных областей, и редактором данных, позволяющим пополнять онтологию фактическими данными путем означивания паттернов содержания. Благодаря простоте и понятности системы паттернов и редактора данных этим подходом могут пользоваться не только инженеры знаний, но и специалисты в моделируемой области знаний.

Данный подход показал свою практическую полезность при разработке онтологий различных научных предметных областей (“Поддержка принятия решений” [40], “Активная сейсмология” [41] и др.).

В ближайшее время планируется реализация лексико-синтаксических паттернов онтологического проектирования для поддержки автоматизированного построения и пополнения онтологий НПО. Сами лексико-синтаксические паттерны предполагается автоматически генерировать на основе входящих в систему паттернов содержания и структурных паттернов с использованием словаря синонимов и тезауруса предметной области.

7. БЛАГОДАРНОСТИ

Работа выполнена при частичной финансовой поддержке РФФИ (проект № 19-07-00762) и Министерства образования и науки Республики Казахстан (проект № AP 05133546).

СПИСОК ЛИТЕРАТУРЫ

1. *Fernández-López M., Gómez-Pérez A., Pazos A., Pazos J.* Building a Chemical Ontology Using Methontology and the Ontology Design Environment // IEEE Intelligent Systems & their applications. 1999. V. 4. № 1. P. 37–46.
2. *Sure Y., Staab S., Studer R.* Ontology Engineering Methodology // Handbook on Ontologies, Eds., *Staab S., Studer R.* Berlin: Springer Verlag, 2009. P. 135–152.
3. *Pinto H., Staab S., Tempich C.* DILIGENT: Towards a fine-grained methodology for DIstributed, Loosely-controlled and evolvInG Engineering of oNTologies // Proceedings of the 16th European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications. IOS Press. 2004. V. 110. P. 393–397.
4. *De Nicola A., Missikoff M., Navigli R.A.* Proposal for a Unified Process for Ontology Building: UPON. In: Database and Expert Systems Applications, Eds., Andersen, K.V., J. Debenham, R. Wagner DEXA 2005 // Lecture Notes in Computer Science. 2005. V. 3588. P. 655–664.
5. *Gangemi A., Presutti V.* Ontology Design Patterns // Handbook on Ontologies, Eds., *Staab, S. and R. Studer.* Berlin: Springer Verlag, 2009. P. 221–243.
6. *Blomqvist E., Hammar K., Presutti V.* Engineering Ontologies with Patterns: The eXtreme Design Methodology // Ontology Engineering with Ontology Design Patterns. Studies on the Semantic Web, Eds., *Hitzler, P., and A. Gangemi, K. Janowicz, A. Krisnadhi, V. Presutti, IOS Press.* 2016. P. 23–50.
7. *Karima N., Hammar K., Hitzler P.* How to Document Ontology Design Patterns // Advances in Ontology Design and Patterns. IOS Press, Kobe, Japan. 2017. V. 32. P. 15–27.
8. *Ломов П.А.* Применение паттернов онтологического проектирования для создания и использования онтологий в рамках интегрированного пространства знаний // Онтология проектирования. 2015. Т. 5. № 2(16). С. 233–245.
9. *Анureев И.С., Батура Т.В., Боровикова О.И., Загорулько Ю.А., Кононенко И.С., Марчук А.Г., Марчук П.А., Мурзин Ф.А., Сидорова Е.А., Шилов Н.В.* Модели и методы построения информационных систем, основанных на формальных, логических и лингвистических подходах / Отв. ред. А.Г. Марчук; Рос. акад. наук, Сиб. отд-ние, Ин-т систем информатики им. А.П. Ершова. Новосибирск: Изд-во СО РАН, 2009. 330 с.
10. *Zagorulko Yu., Borovikova O.* Technology of Ontology Building for Knowledge Portals on Humanities // Knowledge Processing and Data Analysis / K.E. Wolf et al.(Eds): KONT/KPP 2007. Lecture Notes in Artificial Intelligence. Springer-Verlag Berlin Heidelberg, 2011. V. 6581. P. 203–216.
11. *Johnson R., Vlissides J., Helm R.* Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma. Addison-Wesley Professional, Boston, 1994. 395 p.
12. Проект NeOn. http://www.neon-project.org.
13. Портал ассоциации ODPA (Association for Ontology Design & Patterns). http://ontologydesignpatterns.org.
14. Ontology Design Patterns (ODPs) Public Catalog. 2009. http://odps.sourceforge.net.
15. *Dodds L., Davis I.* Linked Data Patterns. 2012. http://patterns.dataincubator.org/book.
16. *Krisnadhi A., Hitzler P.* A Core Pattern for Events // Advances in Ontology Design and Patterns. IOS Press, Kobe, Japan, 2017. P. 29–37.
17. *Krisnadhi A., Hitzler P.* The Stub Metapattern // Advances in Ontology Design and Patterns. IOS Press, Kobe, Japan, 2017. V. 32. P. 39–45.
18. *Hammar K.* Ontology Design Patterns in WebProtégé // Proceedings of 14th International Semantic Web Conference (ISWC-2015). Posters & Demonstrations Track. CEUR Workshop Proceedings. 2015. V. 1486. http://ceur-ws.org/Vol-1486/paper_50.pdf
19. *Zagorulko Y., Borovikova O., Zagorulko G.* Development of Ontologies of Scientific Subject Domains Using Ontology Design Patterns // Communications in Computer and Information Science. 2018. V. 822. P. 141–156.

20. *Antoniou G., Harmelen F.* Web Ontology Language: OWL // Handbook on Ontologies, Eds., *Staab, S. and R. Studer*. Berlin: Springer Verlag, 2009. P. 91–110.

21. DCMI Metadata Terms. http://dublincore.org/documents/dcmi-terms.

22. *Noy N., McGuinness D.* Ontology Development 101: A Guide to Creating Your First Ontology // Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.

23. *Загорулько Ю.А., Боровикова О.И., Загорулько Г.Б.* Использование паттернов онтологического проектирования для разработки онтологий предметных областей // Материалы Всероссийской конф. с международным участием "Знания–Онтологии–Теории" (ЗОНТ-2017), Новосибирск: Институт математики им. С.Л. Соболева СО РАН, 2017. Т. 1. С. 139–148.

24. *Дейт К.Дж.* Введение в системы баз данных = Introduction to Database Systems. 8-е изд. М.: "Вильямс", 2006. 1328 с.

25. *Maynard D., Funk A., Peters W.* Using lexico-syntactic ontology design patterns for ontology creation and population // Proceedings of WOP2009, vol. 516. P. 39–52. CEUR Workshop Proceedings, http://ceur-ws.org/Vol-516/pap08.pdf

26. *Ломов П.А.* Программная реализация технологии генерации лексико-синтаксических паттернов для поддержки решения задач обучения онтологий // Труды Кольского научного центра РАН. 2018. № 10. С. 120–128.

27. *Гаранина Н.О., Сидорова Е.А.* Пополнение онтологий на основе алгебраического формализма информационных систем и мультиагентных алгоритмов анализа текста // Программирование. МАИК "Наука/Интерпериодика". 2015. № 3. С. 32–43.

28. *Боровикова О.И., Загорулько Ю.А., Кононенко И.С.* О применении паттернов онтологического проектирования для извлечения информации из научных текстов // Информационные и математические технологии в науке и управлении. 2018. № 4 (12). С. 18–29.

29. *Большакова Е.И., Баева Н.В., Бордаченкова Е.А., Васильева Н.Э., Морозов С.С.* Лексико-синтаксические шаблоны в задачах автоматической обработки текстов // Компьютерная лингвистика и интеллектуальные технологии: Труды Международной конференции Диалог'2007. М.: Издательский центр РГГУ, 2007. С. 70–75.

30. *Zagorulko Yu., Borovikova O., Zagorulko G.* Pattern-Based Methodology for Building the Ontologies of Scientific Subject Domains // New Trends in Intelligent Software Methodologies, Tools and Techniques. Proceedings of the 17th International Conference SoMeT_18. H. Fujita and E. Herrera-Viedma (Eds.). Series: Frontiers in Artificial Intelligence and Applications. Amsterdam: IOS Press, 2018. V. 303. P. 529–542.

31. *Hammar K., Blomqvist E., Carral D., Van Erp M., Fokkens A. et al.* Collected Research Questions Concerning Ontology Design Patterns // Pascal Hitzler, Aldo Gangemi, Krzysztof Janowicz, Adila Krisnadhi, Valentina Presutti (ed.), Ontology Engineering with Ontology Design Patterns. Vol. 25. Series: Studies on the Semantic Web. IOS Press, 2016. P. 189–198. https://doi.org/10.3233/978-1-61499-676-7-189

32. *de Almeida Falbo R., Barcellos M.P., Nardi J.C., Guizzardi G.* Organizing Ontology Design Patterns as Ontology Pattern Languages // Cimiano P., Corcho O., Presutti V., Hollink L., Rudolph S. (eds) The Semantic Web: Semantics and Big Data. ESWC 2013. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2013. V. 7882. P. 61–75.

33. *Hitzler P., Gangemi A., Janowicz K., Krisnadhi A., Presutti V.* Towards a simple but useful ontology design pattern representation language // Blomqvist E., et al. (eds.) Proceedings of WOP 2017. Vienna, Austria, October 21. CEUR Workshop Proceedings, 2017. V. 2043. http://ceur-ws.org/Vol-2043/paper-09.pdf

34. *Skjæveland M.G., Forssell H., Klüwer J.W., Lupp D., Thorstensen E., Waaler A.* Pattern-Based Ontology Design and Instantiation with Reasonable Ontology Templates // Blomqvist E., et al. (eds.) Proceedings of the WOP 2017. Vienna, Austria, October 21. CEUR Workshop Proceedings, 2017. V. 2043. http://ceur-ws.org/Vol-2043/paper-04.pdf

35. *Guizzardi G.* Ontological Foundations for Structural Conceptual Models, Universal Press, The Netherlands, 2005.

36. *Lawrynowicz A., Esteves D., Panov P., Soru T., Dzeroski S., Vanschoren J.* An algorithm, implementation and execution ontology design pattern // ISWC Workshop on Ontology and Semantic Web Patterns, 2016. https://pure.tue.nl/ws/files/53360523/WOP2016_paper_07.pdf

37. *Mortensen J., Horridge M., Musen M., Noy N.* Modest Use of Ontology Design Patterns in a Repository of Biomedical Ontologies // Proceedings of the WOP 2012, Boston, USA, November 12, 2012. V. 929. CEUR Workshop Proceedings, 2012. http://ceur-ws.org/Vol-929/paper4.pdf

38. *Osumi-Sutherland D., Courtot M., Balhoff J. et al.* Dead simple OWL design patterns // Journal of Biomedical Semantics. 2017. V. 18. № 8.

39. *Brochhausen M., Burgun A., Ceusters W., Hasman A., Leong T.Y., Musen M., Oli-veira J.L., Peleg M., Rector A., Schulz S.* Discussion of "biomedicalontologies: toward scientific debate" // Methods of Information in Medicine. 2011. V. 50. № 3. P. 217–236.

40. *Загорулько Г.Б.* Разработка онтологии для интернет-ресурса поддержки принятия решений в слабоформализованных областях // Онтология проектирования. 2016. Т. 6. № 4 (22). С. 485–500.

41. *Braginskaya L., Kovalevsky V., Grigoryuk A., Zagorulko G.* Ontological approach to information support of investigations in active seismology // Proceedings of the 2nd Russian-Pacific Conference on Computer Technology and Applications (RPC), Vladivostok, Russky Island, Russia, 25–29 September, 2017. P. 27–29.

ПРОГРАММИРОВАНИЕ, 2020, № 4, с. 36–45

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 519.7

НЕКОТОРЫЕ НЕДОСТАТКИ ВХОДНОГО СИНТАКСИСА KeYmaera

© 2020 г. Т. Баар

Hochschule für Technik und Wirtschaft (HTW) Berlin, Department of Engineering I, Wilhelminenhofstraße 75A, 12459 Berlin, Germany

E-mail: thomas.baar@htw-berlin.de

Поступила в редакцию 10.02.2020 г.
После доработки 20.02.2020 г.
Принята к публикации 15.03.2020 г.

Автоматическое средство доказательства теорем KeYmaera позволяет (1) описать киберфизические системы в виде гибридных программ, (2) специфицировать свойства для определенной системы, и (3) формально верифицировать эти свойства с использованием специальной логики, называемой дифференциальной динамической логикой. Синтаксис гибридных программ достаточно примитивен и охватывает только самые основные операторы, такие как *присваивание* (*assignment*), *условная инструкция* (*test*), *последовательность инструкций* (*sequential composition*), *недетерминированный выбор* (*nondeterministic choice*) и *итерация* (*iteration*). Решение сохранить синтаксис гибридных программ очень простым имеет различные последствия: преимущество заключается в том, что логическое исчисление для верификации также остается относительно простым; недостатком является то, что даже маленькие программы сложно понимать, и разработчик вынужден программировать, используя метод копирования и вставки, что значительно затрудняет сопровождение. Однако самый существенный недостаток – это отсутствие модуляризации и концепции библиотек, что сильно затрудняет разработку и проверку крупных систем. В данной работе мы выделяем несколько проблем входного синтаксиса KeYmaera и иллюстрируем их на примерах. Для преодоления этих проблем мы сначала создаем метамодель для оригинального синтаксиса. Затем мы предлагаем расширить метамодель с помощью устоявшихся концепций программирования, таких, например, как подпрограмма и внезапное завершение. Мы иллюстрируем наши расширения с помощью нового графического синтаксиса. Примеры из недавно появившегося учебника KeYmaera используются в нашей статьи в качестве иллюстраций.

DOI: 10.31857/S0132347420040032

1. МОТИВАЦИЯ

Киберфизическая система – это система реального мира, которая обычно состоит как из кибер-, так и из физических компонентов. Поведение киберкомпонентов задается (компьютерной) программой, в то время как поведение физического компонента следует законам физики, например, для описания крутящего момента, ускорения, скорости и т.д. Важным подмножеством киберфизических систем являются системы управления, состоящие из датчиков, процессоров и приводов, правильное функционирование которых имеет первостепенное значение и должно быть проверено с использованием методов формальной верификации.

Гибридная система является формальной моделью киберфизической системы. Для определения поведения киберкомпонентов гибридная система должна быть выражена в нотации программ. Поведение физических составляющих моделируется по законам физики, которые формулируются в терминах обыкновенных дифференциальных уравнений. Средство доказательства теорем KeYmaera способно формально верифицировать свойства гибридных систем, выраженных в виде дифференциальной динамической логики [13, 18]. В настоящей статье мы анализируем эту логику, используемую KeYmaera в качестве входного синтаксиса. Мы указываем на некоторые сложности в данном входном синтаксисе и вносим предложения для их устранения.

Одна из главных проблем используемой логики состоит в том, что она представляет собой единственный формализм, который используется для трех разных целей, а именно: i) описать систему, подлежащую анализу (*описание системы*), ii) формализовать свойства, которые должны быть выполнимыми для системы (*спецификация системы*) и iii) сформулировать доказательства (*верификация системы*). Следует обратить внимание на то, что доказательство является деревом

формул дифференциальной динамической логики, где каждая связь между узлами дерева доказательств должна быть обоснована одним из правил используемого логического исчисления.

Таким образом, один и тот же формализм служит совершенно разным целям и бывают случаи, когда трудно сказать, каково же назначение данного артефакта на самом деле. Например, пользователь KeYmaera иногда вынужден формулировать описание системы в неинтуитивном виде, просто для того, чтобы свойство этой системы можно было верифицировать. Другими словами, требование к системе, которое нужно доказать, оказывает сильное влияние на то, как описывается сама система! Обратите внимание, что – в идеале – необходимо уметь формулировать описание системы совершенно независимо от таких свойств системы, которые нужно доказать – они определяются, как правило, позже. Как мы проиллюстрируем на примере очень простой модели прыгающего мяча, такая независимость не всегда возможна. Это делает использование KeYmaera скорее искусством, чем инженерией.

Синтаксис KeYmaera достаточно примитивен и заставляет пользователя описывать всю систему как огромный объект (Big Blob), поскольку модуляризация, например, в виде подсистем или подпрограмм, синтаксически просто невозможна. При проведении нашего анализа мы выявили также и другие недостатки, например то, что корректное поведение непрерывных состояний зависит от выполнения правильных инструкций перед входом в состояние, или то, что в разных состояниях, как правило, высокая доля похожести обыкновенных дифференциальных уравнений системы. К сожалению, текущий синтаксис не позволяет непрерывному состоянию “наследоваться” от уже определенного непрерывного состояния, чтобы избежать использования метода копирования-вставки при описании системы.

В дополнение к анализу проблем входного синтаксиса KeYmaera, в настоящей работе мы также делаем предложения по устранению этих проблем. Для того, чтобы описать наши решения на нужном уровне абстракции, мы вместо текстового синтаксиса переходим к абстрактному синтаксису, который мы определяем в виде метамодели. Для того чтобы подчеркнуть независимость наших решений от конкретного синтаксиса, мы также будем использовать графический синтаксис, близкий к абстрактному синтаксическому дереву (AST).

2. ОСНОВНЫЕ ПОНЯТИЯ

Сначала мы рассмотрим логическую основу средства доказательства теорем KeYmaera.

2.1. Динамическая логика

Термин *динамическая логика* впервые был использован Харелом и др. в работе [7], которая в свою очередь основывается на работах Пратта [16] и Хоара/Флойда [4, 8]. Пратт недавно опубликовал обзор по истории динамической логики в [17].

Динамическая логика (первого порядка) традиционно используется в анализе компьютерных программ и позволяет для программ α сформулировать свойства, описывающие предусловия/постусловия их выполнения. Синтаксически формулы динамической логики строятся на основе арифметических выражений и атомарных формул, таких как $x < 5 + 3$.

Множество формул замкнуто относительно логических операций $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, кванторов $\forall$, $\exists$ и параметризованных модальностей $[\alpha]$ (*box*), $\langle\alpha\rangle$ (*diamond*), где α – это программа.

Программа синтаксически определяется как дерево операторов. Они включают *присваивание* (:=), *условный оператор* (?), *пустой оператор* (*skip*[1]) как атомарные операторы и *недетерминированный выбор* ($\cup$), *последовательность операторов* (;), и *итерация* (*) как составные операторы. Кроме того, разрешены некоторые производные инструкции (известные как *синтаксический сахар*). Например, программа

$$\mathit{if}\ c\ \mathit{then}\ s1\ \mathit{else}\ s2\ \mathit{endif}$$

определяется как синоним для

$$(?c; s1) \cup (?\neg c; s2)$$

В версии динамической логики, поддерживаемой KeYmaera, все условия (например, 3 + 8), включая переменные, имеют тип Real, поэтому нет поддержки сложной системы типов. Для подробного ознакомления с синтаксисом и семантикой динамической логики, читатель может обратиться к [6].

Семантически, формула в виде $\phi \rightarrow [\alpha]\psi$ декларирует, что программа α, при запуске в состоянии, в котором ϕ выполнена, может либо не завершиться либо, в случае фактического завершения, всегда приведет к состоянию, в котором выполнена ψ. Другая модальность динамической логики $\langle\,\rangle$ (diamond), имеет следующую семантику: $\langle\alpha\rangle\psi$ декларирует, что программа α завершается и для хотя бы одного состояния формула ψ выполнена (обратите внимание, что α может вести себя недетерминированно).

В качестве примера рассмотрим формулу

[1] Так как *skip* можно смоделировать с помощью ?*true*, он не поддерживается всеми версиями KeYmaera.

$$x > 0 \rightarrow [\mathit{if}\ x > 0\ \mathit{then}\ x := x - 1 \\ \mathit{else}\ x := -25\ \mathit{ednif};\ x := x + 2]x > 1 \quad (2.1)$$

Программа α внутри []-модальности представляет собой последовательную композицию (оператор ;) условного выражения и присваивания (оператор :=). Требование, сформулированное в (2.1) к программе α читается следующим образом: *Всякий раз, когда* α *начинается в состоянии, в котором* $x > 0$ *выполнено,* $x > 1$ *также должен выполняться и после завершения* α (обратите внимание, что завершаемость α не является частью требования). Формула (2.1) действительно корректна, т.е. во всех случаях формула оценивается как истинная (см. [6] для формального определения корректности).

В неформальной обстановке довольно легко спорить о корректности (2.1): импликация принимает значение *false*, если его предпосылка оценивается как *true*, а его следствие – *false*. Предпосылка здесь – $x > 0$. Согласно этому, при выполнении программы α, then ветвь первого оператора (if) всегда выполняется и уменьшает переменную x на два. Во второй инструкции значение x увеличивается опять на один, так что значение x в постусловии – давайте обозначим его x_{post} – это $x_{post} = x_{pre} - 1 + 2$, где x_{pre} обозначает значение переменной x в предусловии. Таким образом, (2.1) может быть сведено к условию корректности $x_{pre} - 1 + 2 > 1$, которое никогда не сможет обратиться в *false*, если мы предположим, что $x_{pre} > 0$. К счастью, нам не нужно полагаться далее на неформальные способы, чтобы показать корректность (2.1), ведь можно использовать средство KeYmaera, которое доказывает (2.1) полностью автоматически.

Обратите внимание, что формулы динамической логики ничего не говорят о времени исполнения программы α, а формулируют только свойства по отношениям α к предусловию/постусловию. Можно просто считать, что все операторы в программе α исполняются мгновенно, т.е. их исполнение не занимает никакого времени. Это важное отличие от расширения динамической логики, называемого *дифференциальной динамической логикой*, которое мы рассмотрим ниже.

2.2. Дифференциальная динамическая логика

Дифференциальная динамическая логика [12] является расширением динамической логики, что означает, что каждая формула динамической логики также является и формулой дифференциальной динамической логики, и она таким же образом задает предположения о программе α.

Однако, так как формулы дифференциальной динамической логики в основном используются для описания поведения киберфизических систем, мы говорим, что программа α есть *код поведения киберфизической системы*, а не α *выполняется на машине*, как мы делаем для программ α чистых формул динамической логики.

Единственная разница между динамической логикой и дифференциальной динамической логикой состоит в добавлении нового вида операторов, называемым *оператор непрерывного состояния* ({...}) (или просто *оператор эволюции*), который допускается в программах α. Когда во время выполнения α достигается оператор эволюции, выполнение этого оператора *занимает время* и система остается в соответствующем *непрерывном состоянии* на некоторое время. Обратите внимание, что речь идет о новом семантическом концепте дифференциальной динамической логики, который знаменует собой важнейшее отличие от чистой динамической логики!

Выполнение оператора эволюции для модели киберфизической системы означает для системы оставаться в непрерывном состоянии столько, сколько нужно (время пребывания, как правило, выбирается недетерминированно). Тем не менее, разработчик модели имеет две возможности ограничить период времени, в течение которого система остается в таком состоянии. Первая возможность состоит в том, чтобы добавить так называемое *ограничение домена* к оператору эволюции, которое является формулой первого порядка и отделено от остальной части утверждения с помощью & (амперсанд). Ограничение домена семантически означает, что система не может оставаться в непрерывном состоянии дольше, чем на период времени, в котором ограничение оценивается как *true*. Другими словами: как только текущее значение ограничения домена переключается с *true* на *false*, система должна покинуть состояние эволюции.

Вторая возможность ограничить период времени – это задать последовательную композицию оператора эволюции с последующей условной инструкцией. Теоретически, машина может выйти из состояния эволюции в любое время, но если последующее условие оценивается как *false*, тогда эта ветвь выполнения отстраняется от логического анализа поведения системы. Таким образом, оператор эволюции, за которым сразу следует условный оператор, является общей техникой, чтобы заставить систему оставаться в непрерывном состоянии до тех пор, пока условие теста оценивается как *false*.

2.2.1. Прыгающий мяч. Мы проиллюстрируем как использование оператора эволюции, так и два упомянутых метода ограничения времени, в течение которого система будет оставаться в непрерывном состоянии, с помощью следующего примера прыгающего мяча:

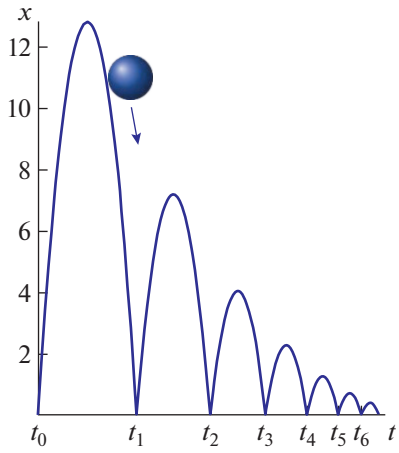


Рис. 1. Пример траектории прыгающего мяча [13, стр. 98].

$$\alpha_{BB} \equiv (\{x' = v, v' = -g \;\&\; x \geq 0\}; \\ ?\, x = 0;\, v := -cv)^* \qquad (2.2)$$

Поведение прыгающего мяча описывается с помощью нового типа переменных, называемых *непрерывные переменные*. Например, переменная x всегда неотрицательна и хранит положение мяча, а переменная v – хранит скорость, которая может быть как положительной (мяч поднимается), так и отрицательной (мяч опускается). Константа g является ускорением свободного падения, она больше 0. Константа c – это коэффициент демпфирования, число от 0 до 1.

Структура α_{BB} такова, что эта программа представляет собой оператор итерации (оператор *) над последовательностью (оператор ;), состоящей из оператора эволюции (заключенного в фигурные скобки {..}), сопровождаемого проверкой условия (оператор ?), а также присваиванием (оператор :=). Программа α_{BB} читается следующим образом: система запускается в состоянии с заданными значениями для переменных x и v (эти значения пока не указаны, но позже мы заставим начальную позицию x_0 быть положительным числом, в то время как начальная скорость v_0 может быть положительной, нулевой или отрицательной) и, пока система остается в первом непрерывном состоянии, значения x, v будут изменяться непрерывно с течением времени в соответствии с законами физики. Таким образом, непрерывные переменные x, v представляют собой скорее функции $x(t), v(t)$ от времени t. Соответствующие физические законы для x, v выражаются двумя обыкновенными дифференциальными уравнениями:

$$x' = v$$
$$v' = -g$$

Последнее означает, что скорость постоянно уменьшается со временем из-за гравитационной силы Земли. К счастью, это дифференциальное уравнение имеет простое полиноминальное решение, которое значительно облегчает анализ всей системы:

$$v(t) = v_0 + -g * t$$

Аналогично, в зависимости от изменяющейся скорости v, положение x прыгающего мяча изменяется как

$$x(t) = x_0 + v_0 * t + \frac{-g}{2} * t^2$$

Ограничение домена $x \geq 0$, упомянутое в операторе эволюции, позволяет системе оставаться в непрерывном состоянии только до тех пор, пока x неотрицательно. Теоретически, система может в любой момент покинуть это состояние, но следующим оператором является проверка условия $? = 0$. Таким образом, если система выходит из непрерывного состояния с $x > 0$, то эта вычислительная ветвь будет отбрасываться.

При верификации свойств системы мы можем полагаться на то, что система покидает непрерывное состояние только тогда, когда $x = 0$, то есть когда мяч касается земли. Следующее присваивание $v := -cv$ определяет, что мяч отпрыгивает вверх: отрицательное значение v падения мяча мгновенно изменится до положительного значения (умножение на $-c$), а абсолютное значение v уменьшается, поскольку мяч теряет энергию при касании земли и изменении направления движения. На рис. 1 показано, как меняется позиция x прыгающего мяча с течением времени (пример траектории).

3. ПРОБЛЕМЫ ВХОДНОГО СИНТАКСИСА KeYmaera

Дифференциальная динамическая логика, представленная выше, поддерживается средством KeYmaera которое позволяет формально верифицировать важные свойства технической системы, что было продемонстрировано в многочисленных тематических исследованиях в разных доменах, например, для самолетов [9, 14], поездов [15] и роботов [11].

Тем не менее, входной синтаксис, используемый для формализации свойств в виде формул дифференциальной динамической логики, страдает от многочисленных проблем, которые описаны ниже. Решения, которые мы предлагаем для преодоления этих проблем, представлены далее в разделе 4.

(1) Инвариантные спецификации не поддерживаются явно. Помимо описания поведения гибридных систем, как показано на примере программы α_{BB} для прыгающего мяча, основное назначение диф-

ференциальной динамической логики – определить также требования для таких систем. Типичными и очень важными на практике требованиями являются так называемые *свойства безопасности*, сообщающие, что система никогда не попадает в "плохую ситуацию". Давайте зададим "плохую ситуацию" с помощью $\neg\psi$. Мы можем показать отсутствие $\neg\psi$, доказав, что во всех достижимых состояниях системы формула ψ выполнима, то есть ψ является инвариантом. Если мы допустили, что все операторы, кроме операторов эволюции, выполнены мгновенно, тогда представленный инвариант на самом деле означает, что ψ выполняется в то время, когда система находится в любом из своих непрерывных состояний. Тем не менее, модальные операторы позволяют описать состояние только *после того*, как программа завершилась. Например, для системы прыгающего мяча α_{BB}, определенной в (2.2), мы можем очень легко доказать

$$x = 0 \to [\alpha_{BB}]x = 0 \tag{3.1}$$

Заметьте, однако, что не было доказано, что $x = 0$ является инвариантом! Если мы хотим выразить такой инвариант, что позиция x остается все время в интервале $[0, H]$, в то время как H задает начальную позицию системы, и если скорость v изначально равна 0, мы должны признать, что формула

$$\begin{aligned} H > 0 \land v = 0 \land x = H \\ \land 0 < c \land c < 1 \to [\alpha_{BB}]x \le H \end{aligned} \tag{3.2}$$

доказуема, но НЕ определяет $x \le H$ как инвариант, потому что эта формула не говорит ничего о x и H, *пока* система остается в непрерывном состоянии $\{x' = v,\ v' = -g\ \&\ x \ge 0\}$, которое является частью α_{BB}. Чтобы доказать, что $x \le H$ является инвариантом, пользователь вынужден переформулировать α_{BB} в

$$\begin{aligned} \alpha'_{BB} \equiv (\{x' = v, v' = -g\ \&\ x \ge 0\}; \\ (skip \cup (?\, x = 0; v := -cv)))* \end{aligned} \tag{3.3}$$

Это, однако, было бы примером описания системы исходя из требования, которое мы хотели бы доказать! Мы считаем это плохим стилем.

(2) Определение оператора эволюции не может быть использовано повторно. Оператор эволюции должен содержать все обыкновенные дифференциальные уравнения, которые выполняются в соответствующих состояниях. Если программа содержит несколько операторов эволюции, то все уравнения, как правило, приходится копировать для всех таких операторов, так как обыкновенное дифференциальное уравнение обычно моделирует физический закон, который выполняется в каждом из непрерывных состояний. В настоящее время, синтаксис KeYmaera не позволяет определить

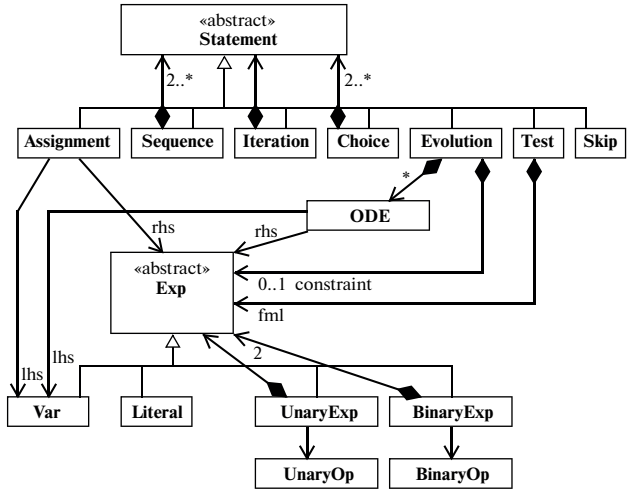


Рис. 2. Метамодель для входного синтаксиса KeYmaera.

все обыкновенные дифференциальные уравнения один раз и затем повторно использовать эти определения для всех встречающихся операторов эволюции. Этот недостаток повторного использования проявляется в стиле "копирование и вставка" при описании системы. В качестве примера, обратимся к пункту 3a из руководства по KeYmaera [18], стр. 10, уравнение (20):

$$\left\{p' = v, v' = -a\ \&\ v \ge 0 \land p + \frac{v^2}{2B} \le S\right\} \cup$$
$$\cup \left\{p' = v, v' = -a\ \&\ v \ge 0 \land p + \frac{v^2}{2B} \ge S\right\}$$

Здесь приведено определение двух эволюционных операторов (в фигурных скобках), которые очень похожи и созданы копированием и вставкой.

(3) Определение непрерывного состояния не инкапсулировано. В руководстве по KeYmaera [12, 18] есть часто применяемый шаблон, чтобы гарантировать, что система остается в непрерывном состоянии $ev \equiv \{\ldots\ \&\ \ldots\}$ не дольше, чем на время ε. Это достигается путем расширения определения ev до $ev' \equiv \{\ldots, t' = 1\ \&\ \ldots \land t \le \varepsilon\}$, где t – свежая непрерывная переменная. Вместе с $t' = 1$ дополнительное доменное ограничение $t \le \varepsilon$ вынуждает систему выходить из ev' не позднее, чем через время ε. Однако это уточненное определение ev работает только в том случае, если значение t заранее установлено на 0. Чтобы достичь этого, выражение ev обычно заменяется на $t := 0; ev'$. Хотя этот шаблон на практике обычно и работает, определение ev' не инкапсулируется и мешает целостности программ.

(4) Отсутствие понятия подпрограммы (или вызова функции в целом). Когда примеры из руководства по KeYmaera [12, 18] становятся немного сложнее, они описываются в составном виде, например, пункт 3a

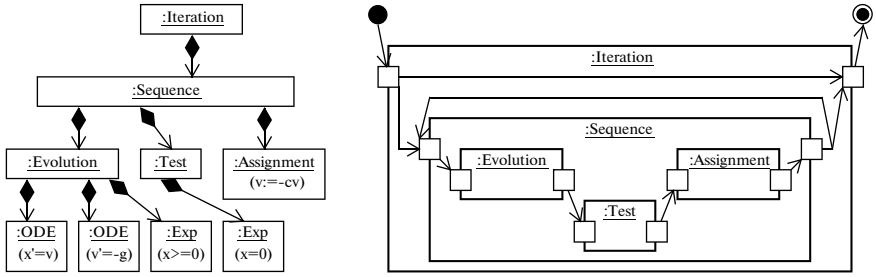


Рис. 3. Экземпляр метамодели (слева) и графический синтаксис, созданный под влиянием потока управления (справа) для программы прыгающего мяча (α_{BB}).

из [18, p. 10]: $init \rightarrow [(ctrl; plant)*]req$, где $init \equiv ...$, $ctrl \equiv ...$, $plant \equiv ...$, $req \equiv ...$. Представление проблемы в такой декомпозированной форме значительно улучшает читабельность. Тем не менее, использование составной записи невозможно для входного файла KeYmaera. В то время как можно задать новые связанные символы *init*, *req* и ограничения их интерпретации с помощью подформул $init \leftrightarrow ...$, $req \leftrightarrow ...$, в настоящее время невозможно определить подпрограммы *ctrl* и *plant* и составить результирующую программу из этих подпрограмм.

4. ПОДХОД НА ОСНОВЕ МЕТАМОДЕЛИ ДЛЯ РЕШЕНИЯ ВЫЯВЛЕННЫХ ПРОБЛЕМ

Указанные выше проблемы могут быть преодолены путем включения языковых концепций из объектно-ориентированных языков программирования и диаграмм состояний во входной синтаксис KeYmaera. Для того, чтобы обсудить включение новых языковых концепций на правильном уровне абстракции, мы формулируем наше предложение в виде измененной метамодели для входного синтаксиса KeYmaera. В качестве отправной точки мы представляем метамодель для текущего синтаксиса.

4.1. Метамодель для текущего синтаксиса KeYmaera

Метамоделирование [5] – это широко распространенный метод определения абстрактного синтаксиса языков моделирования и программирования. Одним известным способом определения языка является применение языка унифицированного моделирования (UML) [19].

Рисунок 2 показывает скетч метамодели для текущего входного синтаксиса KeYmaera с акцентом на операторы программы. Все метаассоциации с кратностью больше 1 предполагаются упорядоченными. Если кратность метаассоциации отсутствует, тогда 1 является ее значением по умолчанию. Метакласс Exp представляет выражения обоих типов *Real* (например, $5 + x$) и *Boolean* (например, $x < 10$).

Конкретная программа α для KeYmaera может быть представлена экземпляром метамодели. Этот экземпляр эквивалентен результату, полученному при синтаксическом разборе такой программы, т.е. абстрактному синтаксическому дереву (AST).

Левая часть рис. 3 показывает метамодель для экземпляра программы прыгающего мяча $\alpha_{BB} \equiv (\{x' = v, v' = -g \;\&\; x \geq 0\}; ?x = 0; v := -cv)*$, как определено в (2.2). В правой части мы видим выровненное графическое представление абстрактного синтаксическое дерева той же программы. Каждый тип оператора представлен блоком с входными и выходными пинами. Поток управления визуализируется направленными ребрами, соединяющими два пина. Предварительные и заключительное состояния выполнения программы представлены символом начальное/конечное состояние, знакомое из машин состояний UML [19].

4.2. Решения выявленных проблем

Основываясь на введенных выше графических обозначениях, перейдем к обсуждению решений проблем, перечисленных в разделе 3.

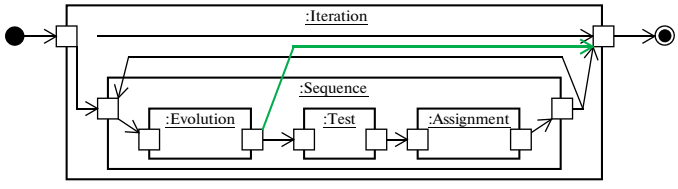


Рис. 4. Решение проблемы инвариантной спецификации.

(1) Инвариантные спецификации не поддерживаются явно. Как описано в разделе 3, модальный оператор [α] всегда относится к пост-состоянию, представленному узлом заключительного состояния на рис. 3, правая часть. Однако для проверки инварианта нам нужна ссылка на состояние после завершения каждого оператора эволюции. Этот момент в решении представлен выходным пином состояния `Evolution`. То, что требуется в семантике программы – это прямое ребро от каждого выходного пина каждого состояния `Evolution` до заключительного состояния, как показано на рис. 4, зеленое ребро. Эта концепция известна как *внезапное завершение* (*abrupt termination*).

Обратите внимание, что внезапное завершение может быть реализовано без каких-либо изменений входного синтаксиса KeYmaera, поскольку оно требует лишь изменения потока управления для существующих операторов.

(2) Определение оператора эволюции не может быть использовано повторно. Часто одни и те же обыкновенные дифференциальные уравнения и ограничения появляются в нескольких непрерывных состояниях снова и снова, что ухудшает читабельность. Чтобы предотвратить это, мы предлагаем ввести *объявление* именованных операторов эволюции, на которые может ссылаться, например, другой оператор эволюции, и наследовать от них дифференциальные уравнения и ограничения. Соответствующее изменение метамодели показано на рис. 5.

Еще одна проблема, которую предстоит обсудить, заключается в том, может ли описание непрерывного состояния производиться в произвольном месте в программе или оно должно быть сделано до программы в качестве глобального описания. Этот вопрос затрагивает важную проблему о том, какую область видимости должен иметь на самом деле идентификатор, введенный таким описанием (см. метаатрибут `name`). Так как разрешение области действия идентификатора является скорее проблемой при разборе программы, эта проблема выходит за рамки этой статьи.

(3) Определение непрерывного состояния не инкапсулировано. Как было показано при определении проблем, оператор эволюции иногда работает как задумано, только когда переменная была заранее установлена правильным значением. Практически, это означает, что непрерывное состояние *EV* всегда предшествует присвоению *ASGN*, поэтому для правильности (*ASGN*; *EV*) должен всегда быть определен. Чтобы избавиться от зависимости непрерывного состояния от присваиваний по контексту (который предотвращает простое повторное использование *EV* в другом контексте), мы предлагаем расширить непрерывное состояние с помощью дополнительных утверждений, которые всегда выполняются при входе или выходе из состояния. Это расширение состояния хорошо известно как входные/выходные действия для машин состояний UML. Соответствующее изменение метамодели показано на рис. 6.

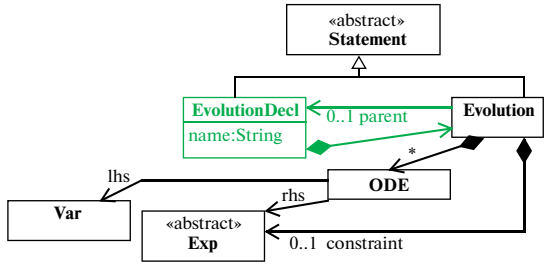


Рис. 5. Решение проблемы повторного использования непрерывных состояний.

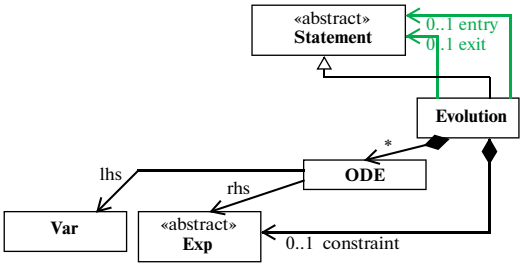


Рис. 6. Решение проблемы инкапсуляции непрерывного состояния.

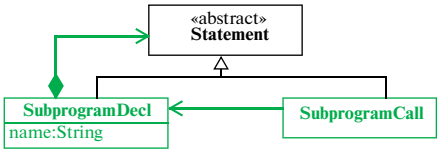


Рис. 7. Решение проблемы с отсутствующими подпрограммами.

(4) Отсутствие понятия подпрограммы. Одно из основополагающих понятий в программировании – это возможность инкапсулировать инструкции с данным именем и повторно использовать эти инструкции в различных местах программы. Эта концепция обычно называется *подпрограмма*, *процедура* или *метод*; в зависимости от того, используются ли параметры или нет. В целом, это очень старая, проверенная и хорошо понятная концепция, так что мы представляем в настоящем предложении решения только самый простой вариант (см. рис. 7).

5. НА ПУТИ К РЕАЛИЗАЦИИ ПРЕДЛОЖЕНИЙ ПО РЕШЕНИЮ

В этом разделе мы рассмотрим возможные варианты реализации предложенного решения и дадим рекомендации по одному из вариантов реализации.

5.1. Реализация путем расширения средства доказательства теорем KeYmaera

Средство доказательства KeYmaera состоит в основном из синтаксического анализатора для входного синтаксиса и исчисления в форме правил доказательств, которые даже могут быть изменены пользователем. Кроме того, существуют некоторые технические компоненты, такие как (i) движок для применения правил доказательств для создания формального доказательства, (ii) адаптеры для подключения внешних систем доказательств, такие как *Z3* или *Mathematica*, и (iii) графический интерфейс для контроля процесса редактирования доказательства. Однако все эти технические компоненты выходят за рамки этой статьи.

Чтобы понять наши предложения, необходимо отличать чисто синтаксические изменения от тех, которые влияют на логическое исчисление, используемое KeYmaera. К последним относятся поддержка внезапного завершения (проблема (1)) и возможность вызова подпрограмм (проблема (4)). Эти изменения потребуют значительного расширения исчисления KeYmaera. Хотя такое расширение требует глубокого знания основного механизма доказательств, тем не менее, оно возможно, как демонстрирует средство доказательства KeY [1][2]. KeY – это интерактивный инструмент формальной верификации программ, реализованных на языке Java, и его исчисление охватывает все тонкости реального языка программирования, включая *вызов функции*, *стек вызовов*, *область видимости переменных*, *внезапное завершение*, *выбрасывание исключения*, *анализ кучи* и т.д.

Чисто синтаксические изменения среди наших предложений, то есть решение проблем (2), (3), могут быть реализованы в KeYmaera просто путем расширения синтаксического парсера. Обратите внимание, что создание альтернативного входного синтаксиса также является и темой текущего проекта под названием Sphinx [10], осуществляемого авторами KeYmaera. Sphinx преследует цель добавить средству доказательства графический интерфейс и позволит пользователю создавать программу в чисто графическом синтаксисе (аналогично нашей графической нотации, предложенной на рис. 3, правая часть).

[2] Средство KeY является предшественником KeYmaera.

Общая проблема, при любых глубоких изменениях средства доказательства KeYmaera заключается в необходимости технических знаний. Кроме того, существуют веские причины для сохранения версии KeYmaera с оригинальным синтаксисом из-за его простоты, благодаря которой KeYmaera гораздо проще использовать для обучения, чем, например, его предшественника KeY. Новую версию KeYmaera с глубокими изменениями сложно поддерживать, так как оригинальная KeYmaera может в будущем также развиваться. По этим причинам глубокие изменения могут сделать только первоначальные авторы KeYmaera сами, и вряд ли кто-то другой.

5.2. Реализация путем создания DSL-фронтэнда

Альтернативным и гибким подходом является разработка DSL-фронтэнда (внешнего интерфейса в виде проблемно-ориентированного языка) с целью включения новых концепций языка, представленных в разделе 4.2. Основная идея заключается в разработке нового предметно-ориентированного языка в соответствии с данной метамоделью. Обратите внимание, что метамодель охватывает только абстрактный синтаксис и сохраняет некоторую гибкость для конкретного синтаксиса. Современные фреймворки для создания DSL языков, такие как Xtext и Sirius, даже позволяют иметь для одного DSL *несколько* представлений (т.е. *конкретных синтаксисов*), поддерживаемых соответствующими редакторами, например, для текстового и графического синтаксиса.

Рисунок 8 показывает общую архитектуру такого инструмента. Следует обратить внимание, что новый инструмент позволит пользователю для создания модели синхронно работать и с текстовым, и с графическим представлением. Однако, такие модели нельзя просто преобразовать во входные файлы для KeYmaera, потому что новый синтаксис поддерживает некоторые семантически-новые понятия, такие как внезапное завершение или стек вызова подпрограммы. Задача компонента *ProofManagement* – разделить задачи, например, для доказательства инварианта – на меньшие условия корректности, которые могут быть сформулированы как формулы дифференциальной динамической логики, и передать эти условия оригинальному средству KeYmaera в виде проверочного бэкэнда. Как инвариантная задача может быть разбита на небольшие условия корректности, показано на конкретном примере в [2].

6. ОБЗОР ЛИТЕРАТУРЫ ПО ТЕМЕ ИССЛЕДОВАНИЯ

Разработка DSL может быть проведена с помощью многочисленных технологий, например Xtext, Spoofax, Metaedit, MPS. Для реализации DSL как с

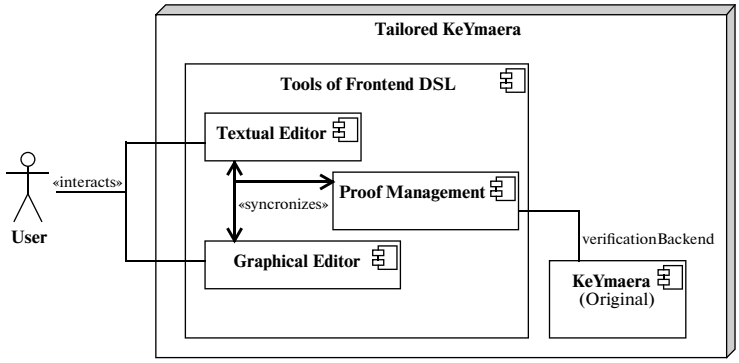


Рис. 8. Архитектура решения с использованием DSL-фронтэнда.

текстовым, так и с графическим конкретным синтаксисом, очень интересно сочетание Xtext и Sirius.

Расширение средства доказательства KeYmaera графическим синтаксисом для программ дифференциальной динамической логики (DDL) проделано в проекте Sphinx [10]. Архитектура этого инструмента очень похожа на наше предложение, представленное на рис. 8, но акцент – в отличии от нашего подхода – лежит не на улучшении читабельности и модульности, делая входной синтаксис более богатым, а чтобы позволить пользователю графически построить программу для DDL.

Обогащение простого императивного языка понятиями из объектно-ориентированного программирования – не редкость в истории компьютерной науки (например, переход от C к C++ или от Modula к Oberon), однако это все еще считается вызовом. В [3] указан отличный учебник Bettini на тему, как включить в простой последовательный язык на основе простых выражений дополнительные понятия из объектно-ориентированного программирования (например, *класс*, *поле*, *метод*, *область видимости*). Полученный язык называется в этом учебнике *SmallJava* и иллюстрирует почти все технические трудности при реализации Java-подобного языка программирования на основе DSL.

7. ЗАКЛЮЧЕНИЕ И ДАЛЬНЕЙШАЯ РАБОТА

Синтаксис программ дифференциальной динамической логики, поддерживаемый средством доказательства теорем KeYmaera, очень прост и низкоуровнен. Преимущество этого решения заключается в том, что даже логическое исчисление для доказательства правильности таких программ является относительно простым, доказательства могут быть легко построены и поняты. С другой стороны, как только примеры становятся немного сложнее – программы становится трудно читать, они плохо структурированы, и их невозможно использовать повторно в другом контексте.

В данной статье мы определили четыре общие проблемы при применении текущего синтаксиса программ на практике. Кроме того, мы внесли предложения по преодолению выявленных проблем путем включения во входной синтаксис KeYmaera проверенных концепций из языков программирования и машин состояний UML. Эти концепции могут сделать программы масштабируемыми и более понятны, так как они способствуют удобочитаемости и модульности.

Наши предложения были сформулированы в форме измененной метамодели, представляющей абстрактный синтаксис программ. Ее преимущество для формулирования предложений заключается в их высокой точности, при этом вопрос, как на самом деле должны быть реализованы изменения в данном конкретном синтаксисе, остается открытым. В настоящее время реализация DSL-фронтэнда, основной составляющей набора инструментов *адаптированной KeYmaera*, находится в стадии разработки, которая еще не завершена.

8. БЛАГОДАРНОСТИ

Данная работа была частично поддержана Deutsche Forschungsgemeinschaft (DFG, Немецкий исследовательский фонд) – проект № 415309034.

Автор очень признателен Сергею Старолетову за обстоятельные обсуждения и большую помощь в переводе статьи на русский язык.

СПИСОК ЛИТЕРАТУРЫ

1. *Ahrendt W., Beckert B., Bubel R., Hähnle R., Schmitt P.H., Ulbrich M.* (eds.): Deductive Software Verification – The KeY Book – From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer, 2016.
2. *Baar T., Staroletov S.* A control flow graph based approach to make the verification of cyber-physical systems using KeYmaera easier. Modeling and Analysis of Information Systems. 2018. V. 25 (5). P. 465–480.
3. *Bettini L.* Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publisher, 2nd edn. 2016.

4. *Floyd R.W.* Assigning meanings to programs. In: Schwartz J.T. (ed.) Proceedings of Symposium on Applied Mathematics. pp. 19–32. Mathematical Aspects of Computer Science, American Mathematical Society, 1967.
5. *Gonzalez-Perez C., Henderson-Sellers B.* Metamodelling for software engineering. Wiley, 2008.
6. *Harel D., Kozen D., Tiuryn J.* Dynamic Logic. Foundation of Computing, MIT Press, 2000.
7. *Harel D., Meyer A.R., Pratt V.R.* Computability and completeness in logics of programs (preliminary report). In: Hopcroft J.E., Friedman E.P., Harrison M.A. (eds.) Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA. pp. 261–268. ACM (1977).
8. *Hoare C.A.R.* An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969).
9. *Jeannin J., Ghorbal K., Kouskoulas Y., Gardner R., Schmidt A., Zawadzki E., Platzer A.* A formally verified hybrid system for the nextgeneration airborne collision avoidance system. In: Baier C., Tinelli C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems – 21st International Conference, TACAS 2015. LNCS, vol. 9035, pp. 21–36. Springer, 2015.
10. *Mitsch S.* Modeling and Analyzing Hybrid Systems with Sphinx – A User Manual. Carnegie Mellon University and Johannes Kepler University (2013), available from: http://www.cs.cmu.edu/afs/cs/Web/People/smitsch/pdf/userdoc.pdf.
11. *Mitsch S., Ghorbal K., Platzer A.* On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: Newman P., Fox D., Hsu D. (eds.) Robotics: Science and Systems IX, Technische Universität Berlin, Berlin, Germany, June 24–June 28, 2013.
12. *Platzer A.* Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg, 2010.
13. *Platzer A.* Logical Foundations of Cyber-Physical Systems. Springer, 2018.
14. *Platzer A., Clarke E.M.* Formal verification of curved flight collision avoidance maneuvers: A case study. In: Cavalcanti A., Dams D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2–6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 547–562. Springer, 2009.
15. *Platzer A., Quesel J.* European train control system: A case study in formal verification. In: Breitman K.K., Cavalcanti A. (eds.) Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9–12, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5885, pp. 246–265. Springer, 2009.
16. *Pratt V.R.* Semantical considerations on Floyd-Hoare logic. In: 17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25–27 October 1976. pp. 109–121. IEEE Computer Society, 1976.
17. *Pratt V.R.* Dynamic logic: A personal perspective. In: Madeira A., Benevides M.R.F. (eds.) Dynamic Logic. New Trends and Applications – First International Workshop, DALI 2017, Brasilia, Brazil, September 23–24, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10669, pp. 153–170. Springer, 2017.
18. *Quesel J.D., Mitsch S., Loos S., Arêchiga N., Platzer A.* How to model and prove hybrid systems with KeYmaera: A tutorial on safety. STTT. 2016. V. 18 (1). P. 67–91.
19. *Rumbaugh J.E., Jacobson I., Booch G.* The unified modeling language reference manual – covers UML 2.0, Second Edition. Addison Wesley object technology series, Addison-Wesley, 2005.

ПРОГРАММИРОВАНИЕ, 2020, № 4, с. 46–64

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 004.415.533

СРЕДСТВА ДИНАМИЧЕСКОГО АНАЛИЗА ПРОГРАММ В КОМПИЛЯТОРАХ GCC И CLANG

© 2020 г. Н. И. Вьюкова[a],*, В. А. Галатенко[a],, С. В. Самборский[a],*****

[a]Федеральное государственное учреждение "Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук"
117218 Москва, Нахимовский проспект, д. 36, к. 1, Россия

**E-mail: niva@niisi.ras.ru*

***E-mail: galat@niisi.ras.ru*

****E-mail: sambor@niisi.ras.ru*

Поступила в редакцию 28.07.2019 г.
После доработки 13.09.2019 г.
Принята к публикации 23.09.2019 г.

Усиливающиеся требования безопасности, предъявляемые к программному обеспечению, рост объемов программных проектов и необходимость постоянно сокращать сроки разработки и выпуска новых версий вызвали настоятельную потребность в средствах динамического анализа, которые бы позволяли эффективно выявлять уязвимости программ на языках C и C++ и предотвращать их эксплуатацию. В статье рассмотрены два вида средств динамического анализа, реализованных в компиляторах gcc и clang и получивших название "санитайзеры". Первый вид санитайзеров используется на стадии тестирования и предназначен для обнаружения ошибок работы с памятью, ошибок несоответствия типов и других уязвимостей, часто остающихся не выявленными при традиционном тестировании. Более эффективному выявлению уязвимостей способствует применение санитайзеров в сочетании с тестированием на автоматически генерируемых случайных наборах данных. Другой вид санитайзеров предназначен для противодействия угрозам безопасности программ, выполняющихся в производственном режиме. Эти средства имеют низкие накладные расходы и ориентированы на поддержание целостности потока управления программы. Применение санитайзеров в значительной мере компенсирует недостатки языков C и C++, такие как небезопасность операций с памятью, возможность небезопасной работы с типами и другие. В то же время в этой области остается ряд нерешенных задач, краткий обзор которых приведен в заключении.

DOI: 10.31857/S0132347420010082

1. ВВЕДЕНИЕ

Языки C и C++ проектировались и развивались как инструменты для создания высокоэффективных программ. Платой за эффективность стало отсутствие требований по осуществлению проверок корректности вычислений во время выполнения. Например, от реализаций этих языков не требуются проверки безопасности доступа к объектам в памяти или отсутствия переполнения при операциях над целочисленными данными. Ответственность за обеспечение разнообразных условий корректности возлагается на программиста. В стандартах языков C и C++ это свойство отражено наличием многочисленных ситуаций неопределенного поведения: "при несоблюдении условия A результат операции B не определен". Неопределенное поведение программы, согласно комментариям к стандартам C и C++, может варьироваться от полного игнорирования ситуации с непредсказуемыми последствиями, до некоторого детерминированного поведения, отраженного в документации, включая, например, прекращение трансляции или выполнения с выдачей диагностики.

Наличие неопределенного поведения в программе далеко не всегда приводит к видимым дефектам ее поведения – программа может годами работать без проблем. Тем не менее оно считается ошибкой по следующим причинам.

• Программа может повести себя непредсказуемо при необычных редко встречающихся входных данных.

• Многие виды неопределенного поведения представляют угрозу безопасности, так как являются уязвимостями, которые могут эксплуатироваться вредоносным программным обеспечением (ПО).

• Наличие неопределенного поведения ухудшает портируемость программы и может стать "бомбой замедленного действия" – программа

может перестать работать корректно при переходе на другую инструментальную платформу или даже к другой версии компилятора либо операционной системы.

Компиляторы старого поколения для некоторых видов неопределенного поведения зачастую обеспечивали интуитивно ожидаемое поведение программ. Современные же компиляторы имеют тенденцию максимально использовать предоставляемую стандартами языков C и C++ свободу для проведения агрессивной оптимизации кода, что может приводить к неожиданному поведению программ. Подробное рассмотрение различных аспектов неопределенного поведения программ представлено в публикации [1].

Для выявления неопределенного поведения в программах традиционно применялись средства статического анализа. Методы статического анализа программ постоянно развиваются (см., например, [2]), тем не менее они остаются принципиально ограниченными в силу невозможности промоделировать все возможные варианты выполнения программы. К числу недостатков средств статического анализа относится возможность ложных срабатываний.

В дополнение к статическим анализаторам применяются инструменты динамического анализа программ для выявления ошибок, имеющие давнюю историю [3]. В настоящее время широко применяются свободно доступные инструменты Memcheck на основе инфраструктуры Valgrind, Dr. Memory, а также различные коммерческие продукты, такие как BoundsChecker, IBM Rational PurifyPlus, Intel® Inspector, продукты компании Parasoft.

Инструменты динамического анализа могут быть классифицированы по ряду признаков.

• Языки программирования, для которых предназначен инструмент анализа.

• Набор выявляемых ошибок. Например, ошибки обращения к памяти, ошибки синхронизации многопоточных программ или другие виды ошибок.

• Набор поддерживаемых аппаратно-программных платформ.

• Представление программы, с которым работает инструмент анализа. Это может быть выполняемый модуль или исходный код программы.

• Поведение при обнаружении ошибки. Возможные варианты – аварийное завершение программы с предварительной выдачей диагностики (или без нее), выдача диагностики и продолжение работы, продолжение выполнения с корректировкой ошибочного поведения программы. Пример последнего из перечисленных подходов представлен в [4], где описан метод контекстно-зависимого обхода ошибок переполнения буфера в некоторых стандартных функциях языка C.

• Основное назначение. Инструменты динамического анализа могут использоваться для тестирования и отладки программ либо для предотвращения эксплуатации уязвимостей в программах, выполняющихся в производственном режиме. В последнем случае ключевым требованием к инструменту анализа являются низкие накладные расходы, связанные с его применением.

Эта статья посвящена средствам динамического анализа, появившимся в последнее десятилетние в компиляторах clang и gcc и получившим название “санитайзеры”. Соответствующий англоязычный термин, sanitizer, переводится на русский как “дезинфицирующее средство”, что неплохо отражает назначение этих инструментов. Санитайзеры предназначены для выявления ситуаций неопределенного поведения во время работы программы, таких как ошибки работы с памятью, ошибки синхронизации в многопоточных программах и другие. В сравнении с инструментами на основе Valgrind и Dr. Memory, санитайзеры clang и gcc поддерживают распознавание ряда дополнительных классов ошибок работы с памятью, в частности, переполнение буфера в статически выделяемой памяти и в локальной памяти (стеке) функций, доступ к локальным объектам функции после выхода из нее.

Поддерживается также распознавание широкого класса ошибок, не связанных с доступом к памяти. Коэффициент замедления типичной программы при использовании санитайзера значительно ниже, чем при использовании упомянутых выше инструментов, что позволяет шире применять динамический анализ для тестировании кода в цикле разработки ПО. В частности, высокую результативность показало применение санитайзеров в сочетании с массированным фаззинг-тестированием, когда используется генерация случайных наборов входных данных для тестируемых приложений или библиотечных функций.

В clang и (в меньшей степени) в gcc поддерживается также ряд средств, предотвращающих эксплуатацию уязвимостей программ. Эти средства можно классифицировать как санитайзеры целостности потока управления (Control Flow Integrity – CFI). Они нейтрализуют последствия различных ошибок, таких как переполнение буфера, не позволяя злоумышленнику перенаправить поток управления программы и выполнить нужный ему код.

Дальнейшее содержание статьи построено по следующему плану. В главе 2 рассмотрены санитайзеры, применяемые для тестирования ПО. Они позволяют обнаруживать ошибки доступа к объектам в памяти, ошибки синхронизации многопоточных программ и многие другие виды не-

определенного поведения программ, которые обычно остаются не выявленными при традиционном тестировании. В главе 3 дается представление о фаззинг-тестировании и преимуществах его применения совместно с санитайзерами. Четвертая глава посвящена санитайзерам целостности потока управления, применяемым для отражения угроз безопасности при промышленной эксплуатации программ. В заключение анализируются некоторые из нерешенных пока задач в этой области и рассматриваются направления дальнейших исследований и разработок.

2. САНИТАЙЗЕРЫ ТЕСТИРОВАНИЯ И ОТЛАДКИ

В этом разделе представлены санитайзеры, предназначенные для применения на стадии тестирования ПО: санитайзер ошибок адресации (AddressSanitizer), санитайзер утечек памяти (LeakSanitizer), санитайзер неинициализированной памяти (MemorySanitizer), санитайзер многопоточных программ (ThreadSanitizer) и санитайзер неопределенного поведения (UndefinedBehaviorSanitizer).

2.1. AddressSanitizer и LeakSanitizer

Инструмент AddressSanitizer [5], поддерживаемый компиляторами clang и gcc, предназначен для выявления ошибок работы с памятью. В GCC, начиная с версии 4.9, он заменил имевшийся там ранее инструмент аналогичного назначения Mudflap.

AddressSanitizer выявляет как пространственные, так и темпоральные ошибки работы с памятью. К первому классу относятся ошибки обращения к памяти по адресу, находящемуся за пределами объекта, такие как выход за границу массива. К темпоральным ошибкам относится доступ к объекту, время жизни которого еще не началось или уже закончилось. Ошибки, выявляемые при помощи AddressSanitizer:

• выход за границы массива (буфера) в динамической памяти, стеке и в статически выделяемой памяти;

• использование динамической памяти после ее освобождения;

• использование локального объекта после выхода из функции, где объект был определен;

• использование объекта после выхода из области его определения;

• повторное или некорректное освобождение памяти;

• некорректные операции над указателями, такие как сравнение указателей, не указывающих на один и тот же объект;

• некорректные аргументы стандартных функций `strcat`, `strcpy`, `memcpy` и других;

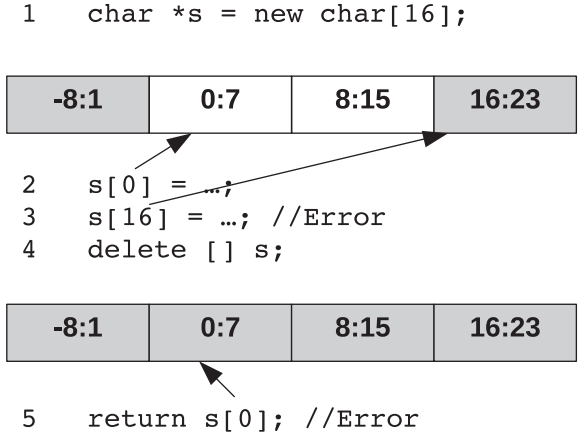


Рис. 1. Принципы работы AddressSanitizer.

• некорректный порядок инициализации глобальных переменных в программах на C++;

• утечки памяти.

AddressSanitizer распознает все основные классы ошибок, выявляемые другими аналогичными средствами, за исключением ошибок использования неинициализированной памяти, для выявления которых предназначен MemorySanitizer. Реализация AddressSanitizer включает модуль компилятора, выполняющий инструментирование программы, и библиотеку времени выполнения. Библиотека времени выполнения включает санитарные версии ряда стандартных функций, таких как `malloc`, `free`, `strcat`, `strcpy`, `memcpy` и других.

Выявление ошибок основано на механизме теневой памяти и выделении “санитарных зон” между объектами в памяти. Рис. 1 иллюстрирует принципы работы AddressSanitizer. Санитарные зоны, выделяемые при создании объекта (строка 1) выделены на рисунке серым фоном. Разметка теневой памяти позволяет определить для каждого байта основной памяти, принадлежит ли он объекту или санитарной зоне. Перед каждым обращением к памяти добавляется проверка ее статуса, и попытка обратиться к памяти санитарной зоны (строка 3) приводит к ошибке. Нетрудно видеть, что, увеличив значение индекса, можно получить доступ к памяти другого объекта, находящегося за санитарной зоной. Такие ошибки AddressSanitizer не обнаруживает.

При уничтожении объекта (строка 4) освобождаемая память помечается как санитарная зона и помещается в карантин, так что ее повторное выделение максимально откладывается; это позволяет идентифицировать ошибки вида “использование после освобождения” (строка 5).

Для активации AddressSanitizer необходимо задать ключ `-fsanitize=address`. Анализатор утечек памяти LeakSanitizer является частью AddressSanitizer и активируется по ключу `-fsani-`

tize=address. Но он также может быть использован самостоятельно при помощи ключа -fsanitize=leak.

Типичное замедление программы, использующей AddressSanitizer, составляет 2x, что существенно меньше, чем замедление при использовании других инструментов. Более высокая эффективность достигается в какой-то мере за счет того, что AddressSanitizer инструментирует код во время компиляции, что позволяет избежать больших задержек на старте программы. Объем требуемой памяти увеличивается в 2–4 раза. Размер стека может увеличиваться примерно в 3 раза. Подробное сравнение AddressSanitizer с аналогичными средствами приведено в [5] и [6].

2.2. MemorySanitizer

Инструмент MemorySanitizer [7] предназначен для отслеживания ситуаций, когда в программе на C или C++ используются неинициализированные данные, то есть когда чтение из стека или из динамически выделенной памяти происходит до записи в эту память. В настоящее время он реализован только в компиляторе clang и активируется ключом -fsanitize=memory.

Отслеживание инициализации данных ведется на уровне отдельных бит памяти, то есть MemorySanitizer способен диагностировать ошибку вплоть до битовых полей в структурах. Копирование неинициализированных данных и простейшие логические или арифметические операции над ними не приводят к ошибке, поскольку это не запрещено стандартами. MemorySanitizer молча отслеживает распространение неинициализированных данных и выдает сообщение об ошибке лишь тогда, когда выполнение программы достигает условного перехода, системного вызова или разыменования указателя, зависящего от неинициализированного значения. MemorySanitizer также включает экспериментальную реализацию проверок использования объектов после выполнения деструкторов.

Стандартная диагностика показывает место, где произошло некорректное использование неопределенного значения, но этого может быть недостаточно для локализации ошибки. Источник ошибки использования неопределенного значения может находиться далеко от места ее проявления, как текстуально, так и по потоку выполнения. Опция -fsanitize-memory-track-origins, аналогичная опции –track-origins=yes инструмента Memcheck, позволяет отслеживать источник ошибки; при этом в диагностическую выдачу включается место создания объекта и все события записи в память неинициализированного значения. В листинге 1 приведен пример программы и диагностики с отслеживанием источника.

```
$ cat -n msan.C
 1  #include <stdio.h>
 2  int x[1] = {9};
 3  int main(int argc, char** argv) {
 4    typedef int* intp;
 5    intp *a = new intp [10];
 6    a[argc] = x;
 7    int *b = a [2];
 8
 9    printf ("*b = %d\n", *b);
10    return 0;
11  }
$ clang++ msan.C -g -Wall -fsanitize=memory \
-fsanitize-memory-track-origins
$ ./a.out
==27687==WARNING: MemorySanitizer:
use-of-uninitialized-value
  #0 0x4a205d in main /home/user/memsan/msan.C:9:24
  #1 0x7fb87418a430 in __libc_start_main
(/lib64/libc.so.6+0x20430)
  #2 0x41b929 in _start (/home/user/memsan/a.out+0x41b929)
Uninitialized value was stored to memory at
  #0 0x4a1fe2 in main /home/user/memsan/msan.C:7:8
```

```
    #1 0x7fb87418a430 in __libc_start_main
(/lib64/libc.so.6+0x20430)
Uninitialized value was created by a heap allocation
    #0 0x49f104 in operator new[](unsigned long)
/home/user/llvm-project-8.0.0/compiler-rt/lib/
msan/msan_new_delete.cc:48
    #1 0x4a1dec in main /home/user/memsan/msan.C:5:13
    #2 0x7fb87418a430 in __libc_start_main
(/lib64/libc.so.6+0x20430)
SUMMARY: MemorySanitizer: use-of-uninitialized-value
/home/user/memsan/msan.C:9:24 in main
```

Листинг 1: Пример программы и диагностики MemorySanitizer с отслеживанием источника.

Условием корректной работы MemorySanitizer является компиляция всего приложения, включая стандартные библиотеки, с ключом `-fsanitize=memory`. Несоблюдение этого условия может приводить к ложным срабатываниям. Поэтому вам придется самостоятельно собрать требуемую версию стандартной библиотеки C++ согласно процедуре, приведенной в документации. Необходимо будет также заменить в программе ассемблерные модули и ассемблерные вставки на код на языке C. Для того чтобы упростить использование MemorySanitizer, в его библиотеку времени выполнения включены “санитарные” версии около 300 наиболее употребительных функций библиотеки языка C, что позволяет применять его с неинструментированной библиотекой libc.

Использование памяти при работе с MemorySanitizer увеличивается в два раза, а при отслеживании источника – в 3 раза. Выполнение типичной программы, использующей MemorySanitizer, замедляется примерно в 3 раза, что значительно меньше, чем при использовании Memcheck или Dr. Memory. Более высокая эффективность MemorySanitizer по сравнению с другими средствами объясняется, во-первых, тем, что инструментирование выполняется на стадии компиляции, и тем самым исключается длительная задержка при старте программы; вторая причина заключается в том, что MemorySanitizer выполняет только проверку использования неинициализированной памяти, в то время как Memcheck и Dr. Memory совмещают функциональность MemorySanitizer и AddressSanitizer. Хотя совместное использование MemorySanitizer и AddressSanitizer не поддерживается, но их последовательное применение занимает, как правило, значительно меньше времени, чем тестирование с помощью Memcheck или Dr. Memory. Сравнение MemorySanitizer с аналогичными инструментами, а также вопросы реализации представлены в [7].

Низкие накладные расходы позволяют применять AddressSanitizer и MemorySanitizer на регулярной основе в цикле разработки ПО как для модульного и регрессивного тестирования, так и в сочетании с фаззинг-тестированием. MemorySanitizer был опробован для тестирования ряда больших проектов, включая сам компилятор clang, компилятор gcc, а также различные серверные приложения Google, где с его помощью было найдено более 500 ошибок.

2.3. ThreadSanitizer

ThreadSanitizer [8] представляет инструмент для выявления ситуаций гонки данных (data races), возникающих в результате ошибок синхронизации многопоточных программ. Реализация включает модуль инструментирования программы в компиляторе и библиотеку времени выполнения.

```
$ cat -n race1.cc
 1 #include <pthread.h>
 2 #include <stdio.h>
 3
 4 int Global;
 5
 6 void *Thread1(void *x) {
 7   Global++;
 8   return NULL;
```

```
 9 }
10
11 void *Thread2(void *x) {
12   Global--;
13   return NULL;
14 }
15
16 int main() {
17   pthread_t t[2];
18   pthread_create(&t[0], NULL, Thread1, NULL);
19   pthread_create(&t[1], NULL, Thread2, NULL);
20   pthread_join(t[0], NULL);
21   pthread_join(t[1], NULL);
22 }
$ clang++ race1.cc -fsanitize=thread -g
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=10915)
 Write of size 4 at 0x0000011a78c8 by thread T2:
  #0 Thread2(void*) /home/user/tsan/race1.cc:12:9
(a.out+0x4c711e)
Previous write of size 4 at 0x0000011a78c8 by thread T1:
  #0 Thread1(void*) /home/user/tsan/race1.cc:7:9
(a.out+0x4c70be)
Location is global 'Global' of size 4 at 0x0000011a78c8
(a.out+0x0000011a78c8)
Thread T2 (tid=10918, running) created by main thread at:
  #0 pthread_create /home/user/llvm-project/compiler-rt
/lib/tsan/rtl/tsan_interceptors.cc:975 (a.out+0x429596)
  #1 main /home/user/tsan/race1.cc:19:3 (a.out+0x4c718a)
Thread T1 (tid=10917, finished) created by main thread at:
  #0 pthread_create /home/user/llvm-project/compiler-rt/
lib/tsan/rtl/tsan_interceptors.cc:975 (a.out+0x429596)
  #1 main /home/user/tsan/race1.cc:18:3 (a.out+0x4c7171)
SUMMARY: ThreadSanitizer: data race /home/user/tsan/
race1.cc:12:9 in Thread2(void*)
```

Листинг 2: Пример программы и диагностики ThreadSanitizer.

Во время работы программы регистрируются события доступа к памяти и события синхронизации. Под управлением наблюдаемых событий и некоторого конечного автомата изменяется текущее состояние программы, которое включает глобальное состояние и состояния отдельных потоков. Состояния, соответствующие некорректным последовательностям событий, трактуются как ошибки. В листинге 2 показан пример программы и диагностика, выданная ThreadSanitizer. Другие примеры типичных ошибок представлены в [9].

Замедление программы в результате использования ThreadSanitizer составляет от 5 до 15 раз. Расход памяти может возрастать в 5 – 10 раз. ThreadSanitizer проверялся преимущественно на программах, использующих библиотеку pthread; работа с библиотекой потоков C++11 проверялась пока недостаточно.

Весь код должен быть скомпилирован с опцией `-fsanitize=thread`, включая стандартные библиотеки C и C++, в противном случае возможны как ложноположительные, так и ложноотрицательные срабатывания, а также могут быть показаны не все кадры стека. Не поддерживается статическая компоновка с libc, libstdc++. Диагностируются только ошибки, фактически произошедшие при данном выполнении, поэтому

при тестировании программы желательно обеспечить реалистичную нагрузку.

2.4. UndefinedBehaviorSanitizer

Санитайзер UndefinedBehaviorSanitizer позволяет динамически выявлять в программе разнообразные виды неопределенного поведения, помимо рассмотренных в предыдущих подразделах. Вся совокупность проверок активируется ключом `-fsanitize=undefined`. Ниже перечислены отдельные виды проверок, которые можно задавать ключами вида `-fsanitize=`*проверка*.

`signed-integer-overflow` – переполнение в операциях знаковой целочисленной арифметики. Ошибки этого типа трудно поддаются обнаружению и представляют собой уязвимости, создающие угрозу безопасности ПО [10]. В списке известных уязвимостей от 2011 г. они фигурируют в числе 25 наиболее опасных [11].

`float-cast-overflow` – переполнение при преобразовании из вещественного типа в целочисленный или обратно, а также при преобразовании между двумя вещественными типами.

`bounds` – выход за границы массива при индексной адресации в случаях, когда границы массива могут быть вычислены статически. В листинге 3 приведен пример программы и диагностики UndefinedBehaviorSanitizer.

```
$ cat -n bounds.c
 1 int ops [13] = {11, 12, 46, 3, 2, 2, 3, 2, 1, 3, 2, 1, 2};
 2 int num = 13;
 3
 4 int main()
 5 {
 6   int i;
 7   for (i = 0; i < num; i++)
 8     {
 9        int j;
10        for (j = num - 1; j >= i; j--)
11          {
12             if (ops[j-1] < ops[j])
13              {
14                 int op = ops[j];
15                 ops[j] = ops[j-1];
16                 ops[j-1] = op;
17              }
18          }
19     }
20   return 0;
21 }
clang -g -O1 bounds.c -fsanitize=bounds -g -Wall
$ ./a.out
bounds.c:12:15: runtime error: index -1 out of bounds
 for type 'int [13]'
```

Листинг 3: Пример программы и диагностики UndefinedBehaviorSanitizer.

`shift` – некорректные операнды операторов сдвига, например, когда величина сдвига отрицательна или превышает разрядность сдвигаемого значения.

`alignment` – использование невыровненного значения указателя или создание невыровненной ссылки. Проверяется также выравнивание значений в соответствии с атрибутами `assume_aligned` и `align_value`, а также выравнивание в соответствии с параметром aligned в директивах OpenMP.

`bool` – проверка считываемых из памяти значений типа `bool`. Ошибка диагностируется, если значение не является ни `true`, ни `false`.

`enum` – проверка считываемых из памяти значений типа `enum`. Полезность этой проверки ограничена, поскольку проверяется лишь то, что значение находится в диапазоне разрядности, со-

ответствующей данному типу. Например, если тип содержит значения 1, 5, 6, 7, 99, то ошибкой будет значение вне диапазона 0..127.

`float-divide-by-zero`, `integer-divide-by-zero` – вещественное и целочисленное деление на ноль.

`null` – разыменование нулевого указателя, создание нулевой ссылки. Проверка указателей, отмеченных атрибутами `nonnul` (аргументы функций), `returns_nonnul` (возвращаемое значение функции).

`object-size` – попытка использовать байты, не являющиеся частью объекта, к которому осуществляется доступ. Выявление различных видов ошибок доступа к объектам по указателям. Например, ошибочное приведение к типу-наследнику или вызов методов по некорректному указателю.

`return` – в программах на C++ достижение конца функции, возвращающей значение, без возврата значения.

`unreachable` – достижение вызова встроенной функции `__builtin_unreachable()`, что является неопределенным поведением. Вызов указанной функции заменяется на вызов диагностического сообщения.

`vla-bound` – создание массива переменного размера, где размер не является положительным значением.

`vptr` – проверка указателей на таблицу виртуальных функций. Проверяются ситуации использования объектов с некорректным динамическим типом, а также объектов, время жизни которых уже закончилось или еще не началось.

Только в clang (но не в gcc) поддерживаются следующие проверки.

`function` – косвенный вызов функции не соответствующего типа по указателю (только для C++ на платформах x86/x86_64 под Darwin/Linux).

`builtin` – передача некорректных аргументов встроенным функциям компилятора.

`pointer-overflow` – арифметические действия над указателями, приводящие к переполнению.

Clang поддерживает, в дополнение к перечисленным выше, проверки ряда ситуаций, которые не относятся к категории неопределенного поведения, но зачастую не соответствуют ожиданиям программиста:

`unsigned-integer-overflow` – переполнение в операциях беззнаковой целочисленной арифметики.

`implicit-unsigned-integer-truncation`, `implicit-signed-integer-truncation` – неявное преобразование целочисленного значения к целочисленному типу с меньшей разрядностью с потерей данных.

`implicit-integer-sign-change` – неявное преобразование между целочисленными типами, при котором происходит смена знака.

`nullability-assign`, `nullability-arg`, `nullability-return` – проверка указателей, отмеченных спецификатором `_Nonnull`.

В отличие от других рассмотренных выше санитайзеров, UndefinedBehaviorSanitizer может быть применен на любой платформе, поддерживаемой компилятором, если задан ограниченный режим, не требующий использования библиотеки времени выполнения `libubsan`. Ограниченный режим активируется ключом `-fsanitize-undefined-trap-on-error` и подразумевает, что при первой же ошибке выполнение программы будет аварийно завершено вызовом встроенной функции `__builtin_trap` без выдачи диагностического сообщения от санитайзера.

2.5. Настройки санитайзеров

Эксперименты показывают, что наиболее точную информацию о локализации ошибки можно получить при компиляции без оптимизации (хотя в документации рекомендуется использовать `-O1 -fno-optimize-sibling-calls -fno-omit-frame-pointer`). Опция отладки `-g` нужна для отображения места возникновения ошибки с указанием имен файлов и номеров строк исходного кода.

Дополнительное управление работой санитайзеров осуществляется при помощи переменных окружения `ASAN_OPTIONS`, `MSAN_OPTIONS`, `TSAN_OPTIONS`, `LSAN_OPTIONS`, `UBSAN_OPTIONS`. В частности, с их помощью можно подавлять известные ошибки. Например: `UBSAN_OPTIONS=suppressions=`*файл*, где *файл* содержит список директив, специфицирующих список игнорируемых ошибок:

```
signed-integer-overflow:module.cpp
alignment:function
vptr:shared_object.so
```

Поддерживается также атрибут функций, позволяющий отменять для них заданные проверки, например, `__attribute__((no_sanitize("null")))`. Это полезно для игнорирования известных ошибок, для функций, выполняющих низкоуровневые манипуляции с системными данными или повышения производительности заведомо корректно реализованных функций.

В clang (но не в gcc) отмена проверок на стадии компиляции поддерживается при помощи ключа `-fsanitize-blacklist=`*файл*, где *файл* – имя

файла с директивами отмены проверок в заданных файлах или функциях. Еще одна полезная возможность в clang – условная компиляция в зависимости от использования санитайзеров. Она управляется директивами препроцессора вида #if `__has_feature`(*санитайзер*).

3. САНИТАЙЗЕРЫ И ФАЗЗИНГ-ТЕСТИРОВАНИЕ

Фаззингом (англ. fuzzing) называется метод тестирования программ, как правило, автоматизированного, с использованием случайных входных данных, возможно неправильных и не ожидаемых тестируемой программой. Фаззинг применяется для тестирования программ или библиотек, работающих с достаточно сложными по структуре входными данными, такими как программы сериализации/десериализации, программы сжатия/разжатия данных, медиа кодеки, криптографические программы, текстовые процессоры, компиляторы, ассемблеры, интерпретаторы и другие.

Растущий интерес к фаззинг-тестированию вызван в значительной степени повышением требований к безопасности ПО. В частности, это относится к применению фаззинга в сочетании с санитайзерами или другими средствами динамического анализа программ. В ряде компаний фаззинг-тестирование является обязательным звеном в цикле разработки ПО, имеющего требования по безопасности. Известно, что хакеры также применяют фаззинг для выявления еще неизвестных им уязвимостей программ. Соответственно, производители ПО, подвергнув свои разработки фаззингу, должны избавиться от ошибок безопасности до того, как у хакеров появится шанс воспользоваться ими [12].

Обстоятельный обзор методов и систем фаззинг-тестирования с обширной библиографией представлен в [13]. Введением в фаззинг-тестирование, хотя и несколько устаревшим, может послужить вышедшая в 2009 году книга на русском языке [14]. Не претендуя на полноту, мы лишь кратко остановимся здесь на основных признаках, по которым обычно классифицируют системы фаззинг-тестирования.

Степень использования информации о тестируемой программе. По этому признаку выделяются 3 категории систем. К первой категории относятся системы, тестирующие целевую программу по принципу “белого ящика” и требующие полной информации о ней, включая исходные тексты и спецификации различных аспектов ее функционирования. Ко второй категории относятся системы, не имеющие никакой информации о тестируемой программе (кроме способа ее запуска). Промежуточное положение занимают системы, использующие частичную информацию о тестируемой программе, которую они могут собирать динамически в ходе выполнения программы (например, покрытие кода) или получать путем статического анализа ее исходного кода.

Объекты, варьируемые при фаззинг-тестировании. Это могут быть аргументы командной строки, значения переменных окружения, входные файлы, содержимое области оперативной памяти, пакеты сетевых протоколов, данные, вводимые интерактивно, и другие.

Степень автоматизации. Система фаззинг-тестирования как минимум должна автоматически создавать входные данные (Csmith [16]). Далее, она может автоматически в цикле запускать тестируемую программу, анализировать результат запуска и сохранять входные данные, которые приводят к ошибкам (AFL [17], libFuzzer [19]). Дополнительный сервис может включать минимизацию набора входных данных, на котором воспроизводится ошибка, автоматическую генерацию и отправку отчета об ошибке с проверкой того, что эта ошибка не была зафиксирована ранее. Пример среды фаззинг-тестирования с полным циклом автоматизации – общедоступный сервис OSS-fuzz [18] для тестирования проектов с открытыми исходными текстами на серверах Google.

Способ порождения входных данных. Фаззеры можно разделить на две группы: генерирующие и мутационные. Первые генерируют каждый очередной тест “с нуля”. Они могут использовать набор правил или грамматику, описывающую синтаксис входных данных. Тесты могут порождаться как строго по правилам, так и с отклонениями от них. Такой подход может быть хорош для тестирования компиляторов, интерпретаторов, ассемблеров и других подобных программ (Csmith). Мутационные фаззеры порождают новые тестовые данные путем мутации тестов из заданного набора. К этому классу относятся фаззеры AFL, libFuzzer. Пример фаззера, поддерживающего оба способа – PEACH [20].

Стратегия порождения данных. Хорошую результативность показали появившиеся в последние годы фаззеры, использующие генетические алгоритмы, управляемые покрытием кода тестируемой программы (coverage-guided fuzzing). Принцип их действия в общем виде описывается следующим алгоритмом.

```
Скомпилировать тестируемую программу с
  инструментацией для измерения покрытия кода.
Сформировать начальный набор тестов ("корпус").
Цикл: {
  Создать новый тест путем случайной мутации
    теста из корпуса
  Выполнить новый тест с измерением покрытия.
  Если новый тест увеличивает суммарное покрытие,
    то добавить его к корпусу.
  }
```

На этом принципе основаны системы AFL, libFuzzer, gofuzz [21] и другие. Покрытие кода здесь детализируется с учетом порядка и количества прохождения линейных участков. Например, в AFL различаются покрытия $A \to B \to C \to D \to E$ и $A \to B \to D \to C \to E$; различаются также покрытия с разным числом счетчиков выполнения блоков с точностью до нескольких диапазонов: 1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+. В libFuzzer помимо этого поддерживается фаззинг, управляемый потоком данных (data-flow-guided fuzzing): инструментирование операций сравнения во время компиляции и целенаправленные мутации входных данных для изменения результата сравнения. Это позволяет быстрее достигать новых покрытий кода за счет изменения направления переходов, зависящих от результатов сравнений.

Рассмотрим подробнее преимущества фаззинга в сравнении с традиционными видами тестирования. Например, при модульном тестировании обычно также стремятся достичь максимального полного покрытия кода тестами. Однако фаззинг-тестирование способно улучшить покрытие кода, понимаемое в более широком смысле, чем просто число строк кода в процентах. Помимо собственно исходного кода есть данные, обрабатываемые этим кодом, например, значения индексов в обращениях к массивам. Возможны также многочисленные вариации потока управления: например, для того чтобы воспроизвести ошибку, может быть необходимо зайти в `then`-часть одного оператора `if`, но не зайти в `then`-часть другого, затем выполнить такой-то цикл определенное число раз и т.д. Еще один важный пример, иллюстрирующий расширенное понимание термина “покрытие кода” – накопленное состояние программы, от которого может зависеть ход ее выполнения. Преимуществом фаззинг-тестирования является его способность обеспечить множество различных вариаций покрытия кода.

Другая сильная сторона фаззинга – беспристрастность. Программист-тестировщик может быть подвержен различным иллюзиям, например, иллюзии “простоты” и “очевидности” тестируемых модулей. Или он может положиться на общепризнанную высокую квалификацию программиста-разработчика ПО, и так далее. Фаззинг-тестирование исключает влияние подобных факторов.

Немаловажным преимуществом фаззинг-тестирования является дешевизна его использования в силу высокой степени автоматизации.

Наконец, отметим преимущества совместного применения фаззинга и санитайзеров. Санитайзеры способны выявлять лишь те ошибки, которые произошли при данном пуске программы. Фаззинг позволяет получить максимальный эффект от применения санитайзеров, предоставляя обширную тестовую базу, включающую как корректные, так и некорректные данные, не ожидаемые тестируемой программой. Применение санитайзеров, в свою очередь, позволяет повысить результативность фаззинг-тестирования за счет расширения класса фиксируемых ошибок.

К слабым сторонам фаззинг-тестирования можно отнести то, что оно позволяет, как правило, фиксировать лишь ошибки, приводящие к аварийному завершению (ошибки сегментирования, вызовы assert(), неперехватываемые исключения C++ и другие) либо к зависанию программы. Но, поскольку используются случайные входные данные, обычно невозможно бывает проверить корректность вычисленных результатов. Тем не менее, выявление логических ошибок в тестируемых программах при фаззинг-тестировании вполне возможно. Способы зависят от специфики тестируемых программ, и требуют творческого подхода. Простой пример – выполнение прямой и обратной функции (сжатие-расжатие, шифрование-дешифрование и т. п.) с проверкой того, что результат совпадает с исходными данными: $f^{-1}(f(x)) = x$. Другие методы обсуждаются, например, в [15].

В заключение рассмотрим пример использования системы AFL для тестирования утилиты gawk-5.0.0, скомпилированной при помощи компилятора clang с применением MemorySanitizer. Процедура фаззинг-тестирования при помощи AFL достаточно проста. После сборки са-

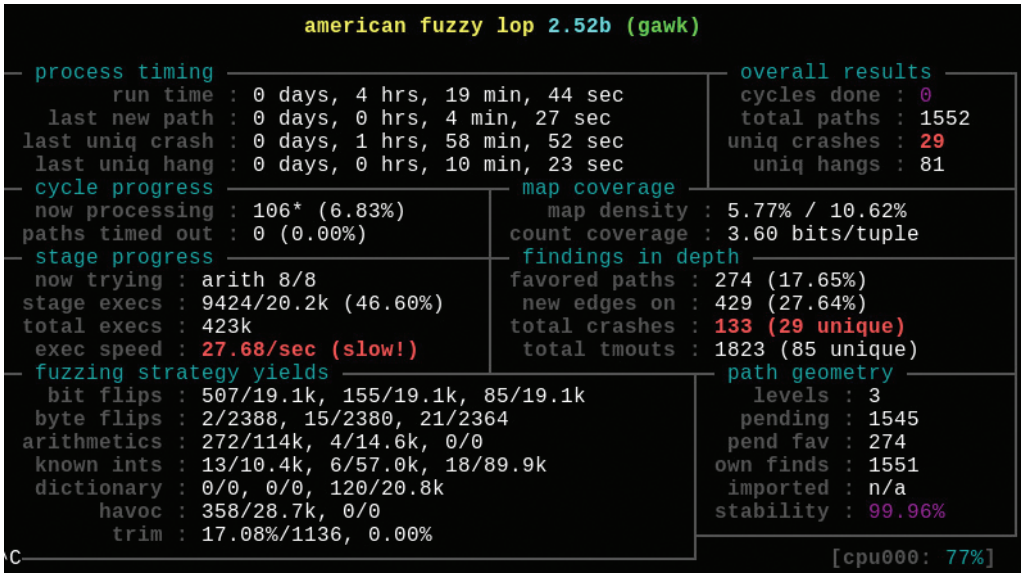


Рис. 2. Экран состояния фаззинг-тестирования при помощи AFL.

мого фаззера согласно описанию нужно определенным образом собрать тестируемое приложение. В данном случае сборка gawk проводилась с использованием переменных окружения `CFLAGS="-g -O2 -fsanitize=memory"`, `CC=~/AFL_PROJECTS/afl-2.52b/afl-clang-fast`. Здесь `afl-clang-fast` – обертка для вызова компилятора clang с определенными ключами, в частности, с ключами сбора тестового покрытия.

Затем необходимо создать подкаталог с корпусом входных данных. Мы поместили в этот каталог одну awk-программу из набора тестов gawk-5.0.0. Запуск фаззера AFL:

```
AFL_USE_MSAN=1 ~/AFL_PROJECTS/afl-2.52b/afl-fuzz -m none -i ./inputs \-o ./out ~/AFL_PROJECTS/local/bin/gawk -f @@ ./fpat1.in
```

Здесь `-m none` означает отсутствие ограничений по использованию памяти, `-i ./inputs` задает каталог с начальным корпусом входных данных, `-o ./out` задает выходной каталог, где AFL будет накапливать корпус сгенерированных входных данных и сохранять файлы данных, вызвавшие аварийное завершение или зависание тестируемой программы. Далее следует имя тестируемой программы и ее аргументы. Опция `-f` задает файл с awk-программой, а символы `@@` обозначают место подстановки имени очередного сгенерированного фаззером файла. Последний аргумент `./fpat1.in` – файл с данными, обрабатываемыми утилитой gawk. Этот файл также взят из набора тестов gawk-5.0.0 и он в ходе тестирования изменяться не будет.

AFL поддерживает фаззинг только одного входного файла, то есть в данном случае можно применить фаззинг к файлу с awk-программой либо к файлу с обрабатываемыми данными, но не к тому и другому сразу.

Фаззинг-тестирование будет продолжаться, пока вы не нажмете `Ctrl-C`. На экране будет отображаться таблица с текущим состоянием тестирования (рис. 2).

В правой верхней части таблицы показано число случаев аварийного завершения (`uniq crashes: 29`) и зависания (`uniq hangs: 81`). Наборы данных для воспроизведения этих ситуаций сохраняются в подкаталогах `out/crashes` и `out/hangs`. Запуск `gawk` с awk-программи, сохраненными в `out/crashes` показал, что все они вызывают одну и ту же ошибку использования неинициализированных данных в лексическом анализаторе. Эта ошибка была исправлена разработчиками программы `gawk`.

4. САНИТАЙЗЕРЫ ЦЕЛОСТНОСТИ ПОТОКА УПРАВЛЕНИЯ

В разделе 2 были рассмотрены инструменты динамического анализа, предназначенные для тестирования ПО. Настоящий раздел посвящен реализованным в компиляторах clang и gcc средствам динамического анализа, предназначенным для обеспечения безопасности выполнения приложений в производственном режиме. Эти средства позволяют исключить или существенно затруднить эксплуатацию уязвимостей, возможно присутствующих в программе. Санитайзеры этого вида могут применяться и на стадии тестирования ПО, но важно понимать, что их срабатывание не всегда соответствует месту возникновения ошибки. Их основное назначение – предотвра-

тить опасные последствия ошибки, а не указать ее местоположение и характер.

Наличие в программе косвенных переходов создает потенциальную возможность выполнить переход на произвольный адрес. Если злоумышленник сумеет модифицировать значение адреса, по которому осуществляется переход, то он сможет использовать существующий код для своих целей. Термин "целостность потока управления" (Control Flow Integrity – CFI) обозначает совокупность методов безопасности, направленных на то, чтобы ограничить возможные пути исполнения программы в рамках графа потока управления, определяемого семантикой программы [22].

Для обеспечения широкого применения CFI важно, чтобы соответствующие механизмы защиты были интегрированы непосредственно в промышленные компиляторы и были совместимы с другими технологиями разработки, такими как разделяемые библиотеки и инкрементальная компиляция. Поскольку средства CFI должны встраиваться в промышленно поставляемое ПО, накладные расходы по памяти и производительности должны быть приемлемыми.

Принято выделять прямые и обратные косвенные переходы. Прямые переходы происходят, например, при вызовах функций по указателю или при вызовах виртуальных методов в языке C++. На графе потока управления они обозначаются прямыми дугами (forward edges). Обратные переходы соответствуют возвратам из функций; они обозначаются обратными дугами (backward edges). Далее будут рассмотрены средства защиты прямого и обратного потоков управления, поддерживаемые компиляторами clang и gcc.

4.1. Защита обратного потока управления

Для защиты обратного потока управления, то есть адресов возврата из функций, в gcc поддерживается ключ `-fstack-protector`, затрудняющий эксплуатацию уязвимостей в стеке (stack smashing). В стек записывается дополнительная переменная-маркер, отделяющая локальные данные функции от сохраненных значений регистров. Перед возвратом из функции значение маркера сравнивается с эталонным и при несовпадении происходит аварийное завершение программы с выдачей диагностики.

Опция `-fstack-protector` применяется только к уязвимым функциям, а именно, к функциям, содержащим вызовы alloca или буферы размером более 8 байт (этот размер буферов регулируется параметром компилятора `ssp-buffer-size`). Поддерживается также тотальная защита всех функций (`-fstack-protector-all`), но она может приводить к ощутимой деградации производительности. Начиная с версии 4.9 в gcc реализован усиленный режим защиты `-fstack-protector-strong`), который защищает функции, содержащие любые локальные массивы, даже внутри структур или объединений, или использующие адреса локальных переменных как аргументы функций либо в правой части присваиваний. Это обеспечивает более сильную защиту в сравнении с `-fstack-protector` без чрезмерной потери производительности.

Компилятор gcc поддерживает также режим выборочной защиты (`-fstack-protector-explicit`), применяемой только к функциям с атрибутом `stack_protect`. Этот атрибут действует и при наличии любой из перечисленных выше опций защиты стека. Таким образом, компилятор предоставляет гибкие возможности для настройки защиты адресов возврата в стеке.

Заметим, что компилятор gcc для ОС Ubuntu использовал `-fstack-protector` по умолчанию с момента реализации данной опции; в компиляторе, поставляемом с последними версиями ОС Ubuntu, по умолчанию действует `-fstack-protector-strong` и `ssp-buffer-size=4`.

Описанный способ, хотя и охватывает большинство уязвимостей переполнения буферов в стеке, встречающихся на практике, не обеспечивает абсолютной защиты. Он защищает при переполнениях в результате циклической записи в непрерывный диапазон адресов, но в других ситуациях может не сработать.

В clang, в дополнение к описанному выше методу, поддерживается санитайзер безопасного стека (ключ `-fsanitize=safe-stack`), являющийся частью проекта CPI (Code Pointer Integrity) [23]. Идея метода заключается в том, что приложение использует два стека вместо одного: безопасный и небезопасный. В безопасном стеке хранятся данные, доступ к которым не может привести к перезаписи других значений в стеке: адрес возврата, регистры, вытолкнутые в память, скалярные локальные переменные. В небезопасном стеке размещается все остальное, в частности, локальные массивы и переменные, от которых берутся указатели. Наборы переменных, сохраняемых в каждом из стеков, определяются путем статического анализа кода. Для защиты безопасного стека применяются различные методы изоляции памяти, которые могут быть специфическими для разных архитектур, см. [23].

Накладные расходы, связанные с использованием санитайзера `safe-stack`, значительно ниже, чем для метода `-fstack-protector`, и составляют не более 0.1%. Это связано с тем, что дополнительный, небезопасный, стек требуется в среднем лишь примерно для 25% функций. Иногда использование второго стека приводит даже к ускорению программы за счет более эффективного кеширования данных: поскольку массивы пе-

реносятся в небезопасный стек, то часто используемые небольшие локальные переменные располагаются более компактно. Еще одно преимущество санитайзера `safe-stack` в сравнении с опцией `-fstack-protector` заключается в том, что он не приводит к аварийным завершениям приложений. Правда, `safe-stack` не всегда применим, поскольку он не поддерживается для разделяемых библиотек и реализован не для всех операционных систем.

Упомянем также санитайзер теневого стека (`-fsanitize=shadow-call-stack`), поддерживаемый компилятором clang. Метод защиты, реализуемый этим санитайзером, заключается в том, что адреса возврата из функций сохраняются в отдельном, теневом стеке и, следовательно, не могут быть переписаны в ситуации переполнения буфера в стеке. Для совместимости с существующими ABI, адрес возврата размещается и в обычном стеке, но его значение там не используется. Санитайзер теневого стека задуман как более сильный вариант защиты адреса возврата в сравнении с методом `-fstack-protector`, поскольку обеспечивает защиту от произвольных, а не только циклических, записей в буфер. Однако в настоящее время он реализован только для архитектуры aarch64. Недостатком этого метода в сравнении с санитайзером `safe-stack` является то, что он защищает только адреса возврата, в то время как `safe-stack` защищает все данные, хранимые в безопасном стеке.

4.2. Защита прямого (восходящего) потока управления

Способы защиты адресов возврата из функций и другой критической информации, размещаемой в стеке, разработаны и поддерживаются в компиляторах довольно давно. В связи с этим злоумышленники переключились на создание альтернативных подходов к эксплуатации уязвимостей с задействованием восходящего потока управления. Например, они могут попытаться перезаписать хранящиеся в динамической памяти указатели на функции или таблицы виртуальных функций. Это может стать возможным при наличии в программе таких уязвимостей, как переполнение буфера либо использование объекта после освобождения. Для противодействия угрозам такого рода в gcc и clang были реализованы механизмы динамического анализа, предотвращающие нарушения целостности восходящего потока управления [24].

В gcc при компиляции программ на C++ поддерживается опция `-fvtable-verify=`, обеспечивающая для каждого виртуального вызова проверку того, что используемая таблица виртуальных методов (virtual method table, VMT) соответствует типу объекта, для которого делается вызов, и что эта таблица не была испорчена или перезаписана. Если в результате такой проверки выявлен некорректный указатель VMT, то выдается диагностическое сообщение и выполнение программы аварийно завершается. Срабатывание может происходить также в результате некорректного приведения типов в программе.

При использовании этой опции перед каждым вызовом виртуального метода вставляется вызов функции, которая проверяет корректность указателя на VMT. Эти проверочные функции используют служебные vtable-map-переменные, указывающие на наборы допустимых VMT для каждого полиморфного класса. Наборы допустимых указателей формируются всегда до входа в функцию main. Опция `-fvtable-verify=` имеет аргумент, который уточняет, когда именно происходит формирование этих наборов: до загрузки и инициализации разделяемых библиотек (аргумент `preinit`) или после (аргумент `std`).

Для корректной верификации необходимо, чтобы весь проект был скомпилирован с ключом `-fvtable-verify=`, иначе наборы допустимых указателей на VMT могут оказаться неполными, что приведет к ложным срабатываниям. В [24] описывается подход, позволяющий обойти эту проблему, если, например, в проекте используются сторонние библиотеки, поставляемые без исходных текстов.

Другое затруднение, с которым вы можете столкнуться, попытавшись воспользоваться этой функциональностью, заключается в том, что предустановленный в системе компилятор, скорее всего, не поддерживает ее. Для этого при конфигурировании gcc должна быть указана опция `--enable-vtable-verify`, которая по умолчанию не активна в силу следующих причин. Обозначим для краткости через vtv-gcc компилятор, сконфигурированный с опцией `--enable-vtable-verify`. Стандартная библиотека C++ в vtv-gcc собирается с ключом `-fvtable-verify=`, чтобы обеспечить корректность верификации. Программа на C++, компилируемая при помощи vtv-gcc даже без `-fvtable-verify=`, все равно будет работать медленнее, чем при сборке стандартным gcc, так как функции библиотеки C++ содержат вызовы верификации. Хотя в этом случае вызываются лишь заглушки, эти дополнительные вызовы замедляют выполнение программ. Для тестов на C++ из SPEC CPU2006, согласно [24], замедление составляет до 4,7%. Но, как показывают эксперименты, замедление может быть и более значительным. Поэтому компилятор vtv-gcc целесообразно применять только для сборки ПО, использующего динамическую верификацию виртуальных вызовов, а в остальных случаях пользоваться стандартным gcc.

В компиляторе clang санитайзер целостности потока управления доступен начиная с версии 3.7. Поддерживается несколько схем динамического анализа, которые могут быть включены по отдельности либо все вместе при помощи ключа `-fsanitize=cfi`.

В clang реализация этой функциональности основывается на доступности графа потока управления для всей программы. Поскольку обычно программа собирается из множества модулей, полный граф становится доступен только на стадии компоновки. Поэтому вместе с `-fsanitize-cfi` необходимо использовать ключ `-flto` для включения оптимизаций времени компоновки (Link Time Optimization, LTO). В рамках прохода LTO выполняется анализ программы и ее трансформация с генерацией заданных видов динамических проверок.

Рассмотрим теперь возможности различных динамических проверок CFI. Опция `-fsanitize=cfi-icall` включает верификацию косвенных вызовов функций. Для каждого косвенного вызова функции по указателю добавляется проверка двух условий: (1) адрес вызова соответствует началу некоторой функции в программе и (2) сигнатура вызываемой функции соответствует сигнатуре указуемой функции, определенной во время компиляции. Данная проверка реализована только для платформ x86 и x86_64.

Упомянутые условия могут нарушаться при эксплуатации уязвимостей, связанных с перезаписью содержимого памяти. Злоумышленник таким образом может попытаться передать управление на фрагменты существующего в программе кода, который реализует нужные ему действия. Эти фрагменты кода (называемые гаджетами) часто не соответствуют началу какой-либо функции; такая подмена указателя не пройдет проверку `cfi-icall`. Не сработает также попытка передать управление на функцию с несоответствующей сигнатурой. Однако эта проверка не спасет от подмены корректного указателя на указатель функции с такой же сигнатурой (например, `delete_user(const char *user)` на `make_admin(const char *user)`).

Опция `-fsanitize=cfi-mfcall` активирует верификацию косвенных вызовов по указателю на метод класса. Проверяется, что метод применяется к объекту подходящего динамического типа и что указуемая функция имеет соответствующий тип.

Опция `-fsanitize=cfi-vcall` включает верификацию вызовов виртуальных методов. Виртуальные методы класса могут быть специализированы в его производных классах, и для них применяется динамическое связывание, то есть конкретный метод определяется во время выполнения в зависимости от типа объекта. Поэтому виртуальные вызовы реализуются как косвенные. При задании `cfi-vcall` проверяется, что вызываемый метод относится к классу из иерархии базовых для объекта, к которому он применяется. Эта проверка выявляет, в частности, ошибки несоответствия типов (type confusion), являющиеся уязвимостями, типичными для программ со сложными иерархиями классов.

Опция `-fsanitize=cfi-nvcall` защищает от вызовов невиртуальных методов для объектов, не относящихся к классам, для которых данные методы были определены. Эта опция подобна `cfi-vcall`, но применяется к невиртуальным вызовам. Поскольку адреса невиртуальных методов известны во время компиляции, то, строго говоря, данный механизм защиты не имеет отношения к CFI. Для каждого невиртуального вызова осуществляется динамическая проверка типа объекта, к которому применяется метод. Динамический тип объекта должен быть производным от типа, известного во время компиляции, или совпадать с ним.

Срабатывания этой проверки могут возникать в результате перезаписи содержимого памяти, ошибок несоответствия типов или ошибок десериализации. Перенаправления потока управления при этом не происходит; опасность заключается в том, что метод может быть применен к данным, для которых он не предназначен.

Опции `-fsanitize=cfi-unrelated-cast`, `-fsanitize=cfi-derived-cast` позволяют динамически проверять и отвергать некорректные приведения типов объектов. Эти проверки также не связаны с целостностью потока управления, а направлены на предотвращение эксплуатации ошибок несоответствия типов. Причинами их срабатывания могут быть также порча содержимого памяти, ошибки десериализации.

Опция `cfi-unrelated-cast` отвергает приведение типов между объектами, типы которых не связаны друг с другом. Такие ошибки часто возникают из-за того, что адреса объектов передаются между разными частями программы как указатели типа `void *`. При приведении от типа `void *` к типу класса будет проверяться, что объект действительно имеет указанный тип. При приведении от одного типа класса к другому проверяется, что эти типы связаны отношением наследования. Опция `cfi-derived-cast` запрещает приведение от базового типа к производному, если объект в действительности не имеет указанный производный тип.

Проверка `cfi-derived-cast` не отвергает приведение от базового типа к производному типу, если производный класс имеет единственный базовый, не вводит своих виртуальных методов и не переопределяет никаких виртуальных методов, за исключением виртуального деструктора.

В этом случае раскладка памяти объектов совпадает, и проблем безопасности не возникает. С точки зрения стандарта языка C++, такое приведение является неопределенным поведением, но этот прием используется во многих проектах. Для того чтобы подобные приведения типов отвергались, необходимо дополнительно использовать опцию `-fsanitize=cfi-cast-strict`.

В заключение рассмотрим пример программы, нарушающей условия проверки `-fsanitize=cfi-nvcall`, который показан на листинге 4. Это сокращенный вариант примера из публикации [25], где можно найти примеры срабатывания и других проверок CFI.

```
01 #include <iostream>
02 #include <string>
03
04 struct Account {
05   Account(const std::string &s) : name(s) {}
06   virtual ~Account() {}
07   void showName() {
08     std::cout << "Account name is: "
09               << name << std::endl;
10   }
11   void adminStuff() { std::cout
12         << "Not Implemented" << std::endl; }
13   std::string name;
14 };
15 struct UserAccount : Account {
16   UserAccount(const std::string &s) : Account(s) {}
17   virtual ~UserAccount() {}
18   void adminStuff() {
19     std::cout
20     << "Admin Work not permitted for a user account!"
21     << std::endl;
22   }
23 };
24 struct AdminAccount : Account {
25   AdminAccount(const std::string &s) : Account(s) {}
26   virtual ~AdminAccount() {}
27   void adminStuff() {
28     std::cout << "Would do admin work in context of: "
29      << this->name << std::endl;
30   }
31 };
32 int main(int argc, const char *argv[]) {
33   UserAccount* user = new UserAccount("user");
34   AdminAccount* admin = new AdminAccount("admin");
35   admin->showName();
36   admin->adminStuff();
37   user->showName();
38   user->adminStuff();
39
```

```
40   Account *account = static_cast<Account*>(user);
41   AdminAccount *admin_it =
42        static_cast<AdminAccount*>(account);
43   admin_it->showName();
44   std::cout << "CFI Should prevent the actions below:"
45             << std::endl;
46   admin_it->adminStuff();
47   return 0;
48 }
```

Листинг 4: Пример программы нарушающей условия проверки `-fsanitise=cfi-nvcall`.

На листинге 5 показаны выдачи этой программы, скомпилированной без `-fsanitise=cfi-nvcall`, и той же программы, скопилированной с `-fsanitise=cfi-nvcall`. Строки 40–42 программы эмулируют ситуацию подмены данных. В результате по указателю `AdminAccount *admin_it` оказывается объект типа `UserAccount`. Ошибка обнаруживается при вызове метода `adminStuff()` в строке 46.

```
$ ./no-cfi-nvcall
Account name is: admin
Would do admin work in context of: admin
Account name is: user
Admin Work not permitted for a user account!
Account name is: user
CFI Should prevent the actions below:
Would do admin work in context of: user
$ ./cfi-nvcall
Account name is: admin
Would do admin work in context of: admin
Account name is: user
Admin Work not permitted for a user account!
Account name is: user
CFI Should prevent the actions below:
nvcall.cpp:46:3: runtime error: control flow integrity
check for type 'AdminAccount' failed during non-virtual call
(vtable address 0x000000437c00)
0x000000437c00: note: vtable is of type 'UserAccount'
  00 00 00 00 90 f0 42 00 00 00 00 00 10 f1 42 00
  00 00 00 00 00 00 00 00 00 00 00 00 88 7b 43 00
```

Листинг 5: Выдача программы из листинга 4

5. ЗАКЛЮЧЕНИЕ

В условиях усиливающихся требований безопасности, предъявляемых к ПО, роста объемов разрабатываемых проектов и необходимости постоянно сокращать сроки разработки и выпуска новых версий возникла настоятельная потребность в инструментах динамического анализа, которые бы позволяли эффективно выявлять уязвимости программ на языках C и C++ и могли применяться на регулярной основе в цикле разработки. Санитайзеры AddressSanitizer, MemorySanitizer, ThreadSanitizer, UndefinedBehaviorSanitizer, реализованные в компиляторе clang, а в дальнейшем вошедшие и в gcc, в значительной мере восполнили этот пробел. Эти средства позволяют обнаруживать многие критические с точки зрения безопасности классы ошибок. Инструменты фаззинг-тестирования, рассмотренные в разделе 3, автоматизируют процесс создания тестов и позволяют многократно повысить эффект применения санитайзеров за счет более полного

покрытия множества возможных путей выполнения программы. Сервис OSS-fuzz (доступный, правда, только для проектов с открытыми исходными текстами) добавляет еще один уровень автоматизации, обеспечивая генерацию отчетов об ошибках и минимизацию тестовых данных для воспроизведения ошибки. Можно ожидать, что следующим шагом на пути автоматизации устранения уязвимостей в программах станет автоматическое исправление типичных ошибок ([26]).

Еще одну линию защиты предоставляют санитайзеры целостности потока управления. Их применение в сочетании с системными средствами защиты, такими как рандомизация размещения адресного пространства (ASLR), предотвращение выполнения кода, находящегося в сегментах данных (DEP), существенно затрудняет эксплуатацию уязвимостей программ. Дополнительную защиту могут обеспечить методы диверсификации программного кода [27].

Рассмотренные в работе санитайзеры позволяют значительно повысить безопасность ПО. В то же время, остается ряд нерешенных задач, относящихся к доступности, простоте использования и полноте существующих средств динамического анализа.

Под *доступностью* понимается набор платформ, для которых поддерживаются рассмотренные средства динамического анализа. В компиляторах gcc и clang они реализованы для ряда наиболее употребительных операционных систем общего назначения, таких как ОС Linux, Android, MacOS, MS Windows. Несомненно, подобные средства были бы полезны и при разработке ПО для встроенных и бортовых систем, к которому предъявляются повышенные требования надежности и безопасности. Портирование средств динамического анализа на платформы, используемые во встроенных системах, затрудняется ограничениями по ресурсам, особенностями функционирования ОС реального времени и, возможно, спецификой процесса разработки ПО для этих систем. В связи с этим интерес представляют работы [29], где рассмотрены вопросы портирования AddressSanitizer на платформу Myriad, и [28], где представлен опыт портирования санитайзеров AddressSanitizer, MemorySanitizer, UndefinedBehaviorSanitizer на платформы под управлением ОС реального времени JetOS. В [28] отмечается также, что подобные инструменты могут быть дополнены средствами проверки требований контрактов для кода при сертификации ПО.

С точки зрения *простоты использования*, положительными характеристиками рассмотренных санитайзеров является их доступность непосредственно в компиляторах gcc и clang и, в большинстве случаев, приемлемые накладные расходы. Наличие различных ограничений может усложнять применение санитайзеров. Например, не все они в полной мере поддерживают разделяемые библиотеки. Неудобства создает также невозможность одновременного применения нескольких санитайзеров, в частности, AddressSanitizer, MemorySanitizer, ThreadSanitizer. Из-за этого тестирование с каждым из них должно проводиться раздельно. Согласно [31], невозможность совместного применения – общая беда многих санитайзеров, связанная с тем, что для анализа им требуются различные несовместимые метаданные.

Для использования MemorySanitizer, ThreadSanitizer необходимо, чтобы все приложение, включая стандартные библиотеки, было скомпилировано с соответствующей опцией. Желательно, чтобы компилятор включал версии библиотек C++, подходящие для применения с этими санитайзерами и обеспечивал их автоматическое подключение при компоновке.

Важный фактор, снижающий популярность санитайзеров среди разработчиков, согласно исследованию [31], – наличие ложноположительных срабатываний. Например, при использовании MemorySanitizer причиной ложноположительных срабатываний может стать невозможность инструментировать весь проект, если он содержит внешние библиотеки. Опыт применения UndefinedBehaviorSanitizer, согласно [31], также показывает значительное число ложных срабатываний. Возможность ложноотрицательных срабатываний, согласно тому же источнику, в меньшей степени влияет на популярность инструмента.

Полнота функциональности. Не менее важная проблема заключается в том, что существующие санитайзеры охватывают далеко не все случаи неопределенного поведения программ на языках C и C++. Например, AddressSanitizer обнаруживает не все ошибки адресации. В некоторых случаях обнаружение ошибки санитайзером зависит от заданного уровня оптимизации. Примеры подобных ситуаций, а также полезные рекомендации по использованию санитайзеров и других способов повышения безопасности программ обсуждаются в публикации [30].

Далее, существующие санитайзеры, обеспечивающие безопасное выполнения программ в производственном режиме, ориентированы в основном на поддержание целостности потока управления. Но, во-первых, они не гарантируют абсолютной целостности потока управления, а,во-вторых, существуют атаки, реализуюемые путем подмены данных (data-only attacks). По этой причине некоторые производители ПО пытаются использовать AddressSanitizer в промышленных релизах, что вряд ли можно считать приемлемым решением из-за высоких накладных расходов. К тому же злоумышленник, зная принципы работы AddressSanitizer, может обмануть за-

щиту. В связи с этим сейчас активно ведутся исследования по созданию санитайзера, основанного на аппаратной технологии теггирования памяти (memory tagging, MT) и указателей, [33], [34]. В сравнении с AddressSanitizer этот подход требует существенно меньших накладных расходов, а предоставляемую им защиту труднее обойти. Поэтому MT-санитайзер может применяться не только при обычном и фаззинг-тестировании, но и для противодействия эксплуатации ошибок. Правда, аппаратная поддержка технологии MT имеется пока лишь на двух платформах: SPARC M7/M8 (Application Data Integrity, ADI) и ARM v8.5 (Memory Tagging Extension, MTE).

Наконец, для некоторых видов неопределенного поведения не существует пока надежных методов обнаружения. Например, это относится к ошибкам перекрытия объектов в памяти (strict aliasing rules, см. [35]) и к нарушениям правил модификации объектов между точками следования (sequence points). Современные компиляторы выдают диагностику в относительно простых случаях, но в целом отсутствие диагностики не гарантирует отсутствия ошибок. Помимо неопределенного поведения, стандарты языков C и C++ описывают ситуации неспецифицированного поведения, которые также могут быть источником ошибок. В частности, не определен порядок вызовов функций при вычислении аргументов функций. В [31] также рассматриваются некоторые легальные языковые средства, являющиеся источником распространенных уязвимостей.

Тем не менее, рассмотренные в работе санитайзеры в значительной мере компенсируют негативные свойства языков C и C++, такие как небезопасность работы с памятью, возможность небезопасной работы с типами и другие. Несомненно, дальнейшее развитие и внедрение подобных средств будет способствовать повышению надежности и безопасности ПО. Отметим, что попытки свести к минимуму эффекты неопределенного поведения на уровне стандарта языка C не привели пока к желаемому результату. В C11 были введены необязательные требования, относящиеся к функциям манипуляций со строками с безопасными (bounds checking) интерфейсами (Annex K) и к свойству анализируемости (Analyzability, Annex L). Хотя компиляторы поддерживают опции, позволяющие снимать некоторые виды неопределенного поведения (strict aliasing, переполнение целых), требование анализируемости в целом, по-видимому, не было реализовано ни в одном из них. Что касается функций с безопасными интерфейсами, то в документе [32], где анализируются недостатки этого пункта стандарта и предлагается в конечном счете отказаться от него, в качестве альтернативных решений рассматриваются прежде всего средства динамического анализа.

6. БЛАГОДАРНОСТИ

Исследование выполнено в рамках государственного задания ФГУ ФНЦ НИИСИ РАН (проведение фундаментальных научных исследований 47 ГП) по теме № 0065-2019-0002 “Исследование и реализация программной платформы для перспективных многоядерных процессоров” (рег. № АААА-А19-119012290074-2).

СПИСОК ЛИТЕРАТУРЫ

1. *Латтнер К.* Что каждый программист на C должен знать об Undefined Behavior. https://habr.com/ru/post/341144/.
2. *Дудина И.А., Белеванцев А.А.* Применение статического символьного выполнения для поиска ошибок доступа к буферу // Программирование. 2017. № 5. С. 3–17.
3. *Glenn R. Luecke, Coyle J., Hoekstra J., Kraeva M., Li Y., Taborskaia O., and Yanmei Wang.* A Survey of Systems for Detecting Serial Run-Time Errors // Concurrency and Computation: Practice and Experience, 2006. P. 1885–1907.
4. *Rigger M., Pekarek D., Mossenbock H.* Context-aware Failure-oblivious Computing as a Means of Preventing Buffer Overflows // Proceedings of the 12th International Conference, NSS 2018. P. 376–390.
5. *Serebryany K., Bruening D., Potapenko A., Vyukov D.* AddressSanitizer: a fast address sanity checker. // Proceedings of the 2012 USENIX conference on Annual Technical Conference. Berkeley, CA, USA, 2012, p. 309–318.
6. AddressSanitizerComparisonOfMemoryTools. https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools.
7. *Stepanov E., Serebryany K.* MemorySanitizer: fast detector of uninitialized memory use in C++ // Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2015. P. 46–55.
8. ThreadSanitizerCppManual. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
9. ThreadSanitizerPopularDataRaces. https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces
10. *Dietz W., Li P., Regehr J.* Understanding Integer Overflow in C/C++. http://www.cs.utah.edu/regehr/papers/tosem15.pdf
11. *Christey S., Martin R.A., Brown M., Paller A., Kirby D.* 2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.
12. *Ализар. А.* Разработан универсальный фаззер, объединивший 15 разных фаззинг-приложений. 2011. https://xakep.ru/2011/04/25/55501/
13. *Man'es V.J.M., Han H., Han C., Cha S.K., Egele M., Schwartz E.J., Woo M.* The Art, Science, and Engineering of Fuzzing:*A Survey*. https://arxiv.org/pdf/1812.00140.pdf
14. *Саттон М., Грин А., Амини П.* Fuzzing: исследование уязвимостей методом грубой силы. Пер. с англ. СПб.: Символ Плюс, 2009. 560 с.

15. *Вьюков Д.* C++ Russia 2017: Fuzzing: The New Unit Testing. https://www.youtube.com/watch?v=FD30Qzd6ylk
16. Csmith. https://embed.cs.utah.edu/csmith/
17. American fuzzy lop. http://lcamtuf.coredump.cx/afl
18. OSS-Fuzz – continuous fuzzing of open source software. https://github.com/google/oss-fuzz
19. libFuzzer – a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html.
20. PEACH Fuzzer. https://www.peach.tech/products/peach-fuzzer/
21. Gofuzz. https://github.com/google/gofuzz.
22. *Abadi M., Budiu M., Erlingsson U., Ligatti J.* Control-Flow Integrity Principles, Implementations, and Applications // ACM Conference on Computer and Communication Security (CCS). November, 2005. P. 340–353.
23. *Kuznetsov V., Szekeres L., Payer M., Candea G., Sekar R., Song D.* Code-Pointer Integrity // Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October, 2014. P. 147–163.
24. *Tice C., Roeder T., Collingbourne P., Checkoway S., Erlingsson U., Lozano L., Pike G.* Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM // SEC'14 Proceedings of the 23rd USENIX conference on Security Symposium. August, 2014. P. 941–955.
25. Let's talk about CFI: clang edition. https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/
26. Getafix: How Facebook tools learn to fix bugs automatically. https://engineering.fb.com/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically/
27. *Нурмухаметов А.Р., Курмангалеев Ш.Ф., Каушан В.В., Гайсарян С.С.* Применение компиляторных преобразований для противодействия эксплуатации уязвимостей программного обеспечения // Труды ИСП РАН. 2014. Т. 26. Вып. 3. С. 113–126.
28. *Cheptsov V., Khoroshilov A.* Dynamic Analysis of ARINC 653 RTOS with LLVM // Ivannikov Isp Ras Open Conference, Moscow, 22–23 November 2018. P. 9–15.
29. *Lee W.* Address Sanitizer on Myriad. https://docs.google.com/document/d/1oxmk0xUojyb-DaQDAuTEVpHVMi5xQX74cJPyMJbaSaRM
30. UB-2017. Часть 1. https://habr.com/ru/post/341694/
31. *Song D., Lettner J., Rajasekaran P., Na Y., Volckaert S., Larsen P., Franz M.* SoK: Sanitizing for Security. 2019, https://oaklandsok.github.io/papers/song2019.pdf.
32. Updated Field Experience With Annex K – Bounds Checking Interfaces. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm.
33. *Serebryany K., Stepanov E., Shlyapnikov A., Tsyrklevich V., Vyukov D.* Memory Tagging and how it improves C/C++ memory safety. Google, February 2018. https://arxiv.org/pdf/1802.09517.pdf.
34. Hardware-assisted AddressSanitizer Design Documentation. http://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.
35. *Horgan P.* Understanding C/C++ Strict Aliasing. http://dbp-consulting.com/tutorials/StrictAliasing.html.

ПРОГРАММИРОВАНИЕ, 2020, № 4, с. 65–72

КОМПЬЮТЕРНАЯ ГРАФИКА И ВИЗУАЛИЗАЦИЯ

УДК 004.421.6

ИССЛЕДОВАНИЕ ТЕХНОЛОГИИ Nvidia RTX

© 2020 г. В. В. Санжаров[a,*], В. А. Фролов[b,c,], В. А. Галактионов[b,***]**

[a] *Российский государственный университет нефти и газа (Национальный исследовательский университет) имени И.М. Губкина 119296 Москва, Ленинский пр., д. 65, Россия*

[b] *Институт прикладной математики имени М.В. Келдыша РАН 125047 Москва, Миусская пл., д. 4, Россия*

[c] *Московский государственный университет имени М.В. Ломоносова 119899 Москва, Ленинские горы, д. 1, стр. 8, Россия*

**E-mail: vs@asugubkin.ru*

***E-mail: vfrolov@graphics.cs.msu.ru*

****E-mail: vlgal@gin.keldysh.ru*

Поступила в редакцию 25.12.2019 г.
После доработки 09.01.2020 г.
Принята к публикации 13.01.2020 г.

Nvidia RTX – это закрытая аппаратно-ускоренная технология трассировки лучей от компании Nvidia. Поскольку детали реализации неизвестны, в сообществе разработчиков было много вопросов о том, что на самом деле представляет из себя аппаратная реализация: какие именно части в конвейере трассировки лучей ускорены аппаратно, а что может быть эффективно реализовано программно. В этой статье мы представляем результаты наших экспериментов с RTX, направленные на понимание внутренней работы этой технологии. В нашей работе мы постарались ответить на вопросы, волнующие разработчиков по всему миру: какое ускорение можно на практике получить по сравнению с программной реализацией и в чем его технологическая основа? Насколько трудоемко будет разрабатывать рендер-систему с поддержкой аппаратного ускорения, которая в то же время может работать на GPU и без RTX (т.е. реализуя трассировку лучей программно), или даже производить вычисления на CPU? Насколько эффективна программная эмуляция RTX, доступная на предыдущем поколении видеокарт Nvidia, и насколько возможно приблизить ее эффективность к аппаратной? Какова будет трудоемкость создания аналога RTX при необходимости запуска приложения на видеокартах других производителей?

DOI: 10.31857/S0132347420030061

1. ВВЕДЕНИЕ

Трассировка лучей является базовой операцией не только в реалистичной компьютерной графике, но и во многих других приложениях (в том числе физическая симуляция, обнаружение столкновений, компьютерная геометрия, симуляция переноса нейтронов в реакторах, визуализация медицинских данных, научная визуализация и др.). В прошлом известно множество реализаций аппаратного ускорения трассировки лучей, однако ни одна из них не была массовой в том смысле, что она была бы интегрирована в штатный графический ускоритель. Поэтому важность появления такой технологии как Nvidia RTX трудно переоценить.

Однако для исследователей и разработчиков по всему миру, использующих трассировку лучей в своих решениях уже сегодня, важно понимать целесообразность интеграции технологии аппаратного ускорения (либо полного перехода на нее). Во-первых, трудоемкость разработки приложений на GPU (особенно при использовании специфической функциональности GPU) в 2–5 раз выше, чем на CPU. Во-вторых, Nvidia RTX – закрытая технология, монопольно предоставляемая на сегодняшний день лишь одним производителем графических ускорителей.

2. ОБЗОР СУЩЕСТВУЮЩИХ РАБОТ

Первыми специализированными аппаратными решениями, связанными с трассировкой лучей, были PCI-карты для визуализации объемных данных, в которых реализовано маршрирование по лучу (ray marching) и затенение по фонгу (например, [1, 2]).

Еще одной заметной ранней реализацией была архитектура SaarCOR [3] и ее обновленная версия

в ПЛИС [4]. Чип SaarCOR реализовал весь алгоритм трассировки лучей – данные сцены и камеры помещались в отдельную DRAM память, соединенную с чипом. Как и рассмотренные ранее работы, SaarCOR использовал трассировку пакетов (в группах по 64 луча). SaarCOR использовал глубокую конвейеризацию для сокрытия высокой латентности доступа к памяти по аналогии к современными GPU: пока одни группы лучей загружают данные, другие, для которых данные уже загружены на чип, могут вычислять пересечение с треугольниками. Подобные системы не получили широкого распространения. Их основной недостаток – узкая направленность системы в целом.

Альтернативой узко-специализированному чипу является размещение большого числа обычных процессоров на одной плате с PCI-e интерфейсом [5–8]. Такие решения обладают наибольшей гибкостью и могут быть использованы не только для ускорения трассировки лучей. Но они не стали популярны в основном из-за их высокой стоимости.

Наконец, существует группа работ, направленных на разработку аппаратных расширений для графических процессоров (либо разработку похожих массивно-параллельных программируемых систем). Одно из первых программируемых решений такого типа было представлено в работе [9]. Обход дерева и поиск пересечений был реализован в специальном блоке с фиксированной функциональностью, в то время как пользовательские программы (шейдеры) выполнялись на т. н. Shader Processing Unit (SPU), очень похожих по архитектуре на ранние процессорные ядра GPU. Как и SaarCOR, работа [9] использовала трассировку пакетов, из-за чего скорость сильно падала на расходящихся в разные стороны (т.н. некогерентных) лучах. Такая же проблема наблюдается во многих GPU реализациях трассировки лучей [10, 11].

Одно из решений проблемы случайного доступа к памяти предложено в работе [12]. Этот подход позразумевает разделение потока запросов к памяти как минимум на 2 потока – поток данных для лучей (ray stream), и поток данных для сцены (BVH дерево, scene stream). Можно сказать, что традиционный подход сокрытия латентности памяти при помощи глубокой конвейеризации, широко используемый в GPU, в работе [12] расширяется таким образом, чтобы загруженный один раз в кэш трилет (фрагмент BVH-дерева) был пройден всеми лучами, которые в данный момент обрабатываются на графическом процессоре. Авторы [12] уверяют что таким образом им удается избегать случайного доступа.

Кроме некогерентных лучей для GPU существует еще проблема нерегулярного распределения работы. Когда в SIMD группе потоков (warp) остается мало активных потоков/лучей, эффективность SIMD процессора GPU существенно снижается. Для решения этой проблемы в работах [10, 11] было использовано уплотнение потоков и регенерация путей, а в [13] была предложена техника блочной регенерации.

В [8, 10, 14] была использована идея группировки BVH дерева в т. н. трилетах (treelets) – небольших фрагментах BVH-дерева. Основное отличие работы [14] состоит в том, что в трилетах можно хранить данные об ограничивающих объемах в BVH с пониженной точностью в 5 бит на 1 плоскость (вместо 32 бит для стандартного типа float). Благодаря этому снижается нагрузка на память и улучшается эффективность работы кэша GPU. Кроме того, решение, предложенное в [14], является относительно дешевым в плане занимаемой площади кристалла (то есть по количеству используемых транзисторов).

Некоторые работы были направлены на аппаратную реализацию трассировки лучей для мобильных систем, где важен такой параметр как энергопотребление системы [15, 16]. Эти работы были нацелены в основном на реализацию классической трассировки лучей [17], и в отличие от многих работ рассмотренных выше, используют MIMD архитектуру с VLIW процессорами, чтобы уменьшить потери энергии/эффективности во время вычислений для расходящихся лучей.

Итого, за последнее время было разработано множество аппаратных реализаций трассировки лучей. Более полный обзор можно найти в работе [18]. Кроме того, некоторые коммерческие компании также презентовали свои решения [19], хотя в настоящее время они не доступны публично. Таким образом, RTX является первой технологией, доступной широкой общественности. Но т. к. эта технология закрыта, неясно какие методы ускорения она использует. Чтобы это понять, мы исследовали Nvidia RTX как черный ящик, проводя различные эксперименты и измеряя производительность. Для этой цели мы реализовали базовый интегратор освещенности на основе трассировки путей, используя интерфейс Vulkan.

2.1. Трассировка путей на GPU

Трассировка лучей на GPU сама по себе является ограниченной и компактной задачей, которую можно решать эффективно различными способами. Однако, проблема в корне меняется, когда на основе трассировки лучей необходимо построить расширяемую программную систему с большим количеством различных функциональностей. При этом необходимо хотя бы приблизительно сохранить исходный уровень производительности. Эта задача во-многом нетривиальна даже для CPU реализаций, но на GPU она требует применения особых подходов. На данный момент известно три основных подхода:

1) “Убер-ядро” (uber-kernel) – подход, при котором код организуется вручную или автоматически (обычно последнее) в виде конечного автомата внутри одного вычислительного ядра. Автомат используется для того, чтобы сократить регистровое давление, поскольку каждое состояние в верхнем операторе switch получает в свое распоряжение все доступные для программы (вычислительного ядра) регистры. Основные недостатки этого подхода – существенные потери производительности на ветвлениях (когда разные потоки выполняют разные состояния), и влияние различных состояний на производительность друг друга, т. к. итоговое вычислительное ядро требует столько регистров, сколько нужно самому тяжелому состоянию [20, 21].

2) “Разделенное ядро” (separate kernel) – подход, при котором код организуется (как правило вручную) в виде нескольких вычислительных ядер, общающихся между собой явно через буферы данных в памяти [20]. Этот подход решает основные недостатки убер-ядра, и благодаря явному разделению на ядра позволяет сохранять производительность критичных участков кода. Однако он обладает повышенной трудоемкостью разработки (из-за необходимости явной передачи данных, что особенно заметно при наличии сортировки или уплотнения потоков [13]). Кроме того, повышаются накладные расходы на запуск и ожидание ядер, а также на саму передачу данных. Поэтому такой подход может замедлять работу программы на простых сценах/случаях, когда накладные расходы становятся соизмеримы с полезной работой ядер.

3) “Трассировка путей волновыми фронтами” (wavefront pathtracing) – это сложный подход, в котором работа и данные для лучей группируются в отдельные очереди [21]. Очереди выполняются в разных ядрах, а результат сохраняется в нужное место памяти также путем отдельных вызовов вычислительных ядер. Благодаря группировке по условным шейдерам, wavefront pathtracing меньше теряет на ветвлениях, чем предыдущие подходы. Однако сортировка и уплотнение потоков для лучей в этом подходе строго обязательны, поэтому его накладные расходы еще выше, чем в предыдущем случае.

3. ИЗВЕСТНЫЕ ДЕТАЛИ

В настоящий момент технология RTX доступна в таких программно-аппаратных интерфейсах (Aplication Programming Interface или API) как DirectX12, Vulkan и OptiX. Для нашего исследования наибольший интерес представляет API Vulkan, поскольку он был разработан специально для того чтобы предоставлять разработчикам максимально прозрачный доступ к функциональности графических процессоров на низком уровне. Этот подход отличается, например, от OptiX, в котором компания Nvidia стремится скрыть детали, упростив жизнь разработчику прикладных приложений. Что касается DirectX12, при внимательном анализе можно обнаружить, что Microsoft добавляет некоторую собственную функциональность, реализуемую в их компиляторе HLSL. Среди дополнительных возможностей, доступных в DirectX12, следует отметить появившуюся в новой версии (т.н. DXR Tier 1.1) возможность “inline” трассировки лучей – вызова функций трассировки луча в произвольном шейдере (пиксельном, вычислительном и др.) без создания специального конвейера трассировки лучей [22]. В этом случае вызывающий код берет на себя всю работу по использованию результатов трассировки луча – вычисления в случае найденных пересечений с тем или иным видом примитива, в случае промаха и т.д. По этой причине в качестве основного API мы выбрали Vulkan.

Трассировка лучей в Vulkan используется в виде отдельного вида конвейера наряду с традиционным графическим и вычислительным конвейером общего назначения. Для запуска этого конвейера необходимо заранее строить ускорящую структуру в виде двухуровнего дерева. Нижний уровнень дерева (Bottom Level Acceleration Sructure или BLAS) строится над отдельными объектами (RTX поддерживает возможность объявления своего геометрического примитива) или мешами. Верхний уровень дерева (Top Level Acceleration Sructure или TLAS) строится над множеством инстансов (копий) объектов/мешей нижнего уровня. Что касается построения ускоряющих структур, наиболее свежая информация об этом появилась на конференции SIGGRAPH в 2019 году [23].

Сам конвейер трассировки лучей имеет 5 программируемых стадий: (1) генерация луча, (2) промах, (3) ближайшее пересечение (которая вызывается уже после того как пересечение найдено), (4) специальная стадия для реализации прозрачных теней и аналогов (называемая “any hit”, но это название сбивает с толку) и, наконец, (5) пересечение луча с геометрическим примитивом. Вместо пятой стадии есть встроенная реализация пересечения луча с треугольником, которая повидимому реализована аппаратно [24].

Между стадиями конвейера лучи переносят т. н. полезную нагрузку (ray payload) – некоторые данные, например координаты или цвет. Nvidia рекомендует делать полезную нагрузку как можно меньше, так же как и при передаче данных между стадиями графического конвейера [25]. В [23] можно заметить, что данные между различными стадиями конвейера трассировки лучей передаются не через память, а через внутренние очереди. Однако, чем больше размер полезной нагрузки, тем меньше эти очереди могут помочь.

Интересно отметить, что функциональность под-проходов (subpass) в API Vulkan в принципе

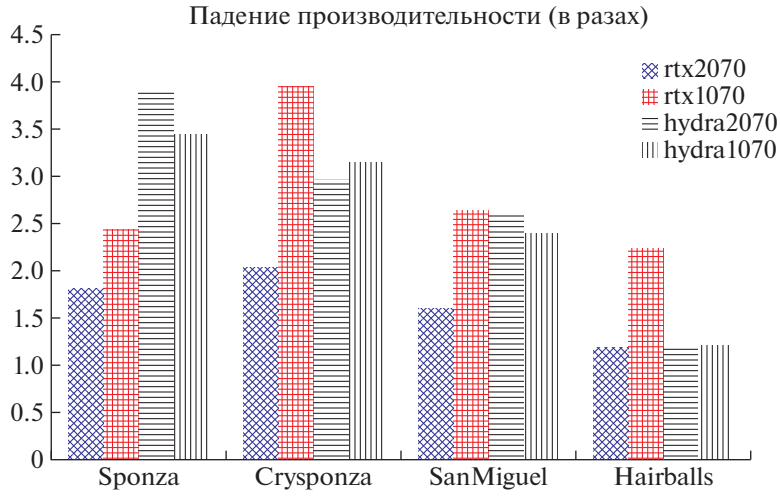


Рис. 1. Падение производительности в разах при переходе от первичных лучей ко вторичным.

позволяет передавать данные между различными вычислительными ядрами не выгружая их в память, и таким образом, симулировать механизм передачи данных, используемый в RTX (в соответствии с [25]). Однако для этого вычисления придется организовывать через графический конвейер специфическим образом, поскольку подпроходы были придуманы для специального алгоритма отложенного затенения. Кроме того, подпроходы предназначены в первую очередь для мобильных графических процессоров, и на настоящий момент нет гарантии того, что на десктопных GPU они не игнорируются полностью (т.е. в реальности все данные могут передаваться через память).

Наконец, стоит сказать, что Nvidia занимается разработкой трассировки лучей в продукте OptiX последние 5–7 лет. Поэтому мы еще раз обращаем внимание на работу [21], в которой предлагается т. н. “wavefront path tracing”.

Таблица 1. Миллионы лучей в секунду для разрешения 1024 × 1024 и 1 сэмпла на пиксел (spp), видеокарта GTX1070 (программная реализация RTX от Nvidia)

Сцена	перв.	втор.	трет.
Sponza, RTX	103	42	38
Sponza, Hydra	**214**	**62**	**59**
Crysponza, RTX	95	24	14
Crysponza, Hydra	**132**	**42**	**37**
San Miguel, RTX	26	10	6
San Miguel, Hydra	**55**	**23**	**20**
Hairballs, RTX	22	9.5	7
Hairballs, Hydra	**27.6**	**22.8**	**25**

4. ЭКСПЕРИМЕНТЫ

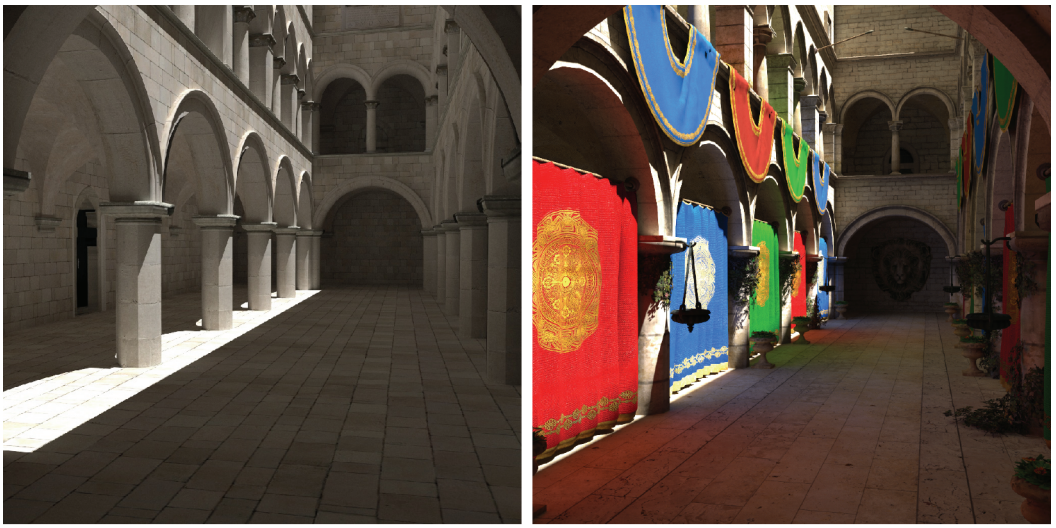
Чтобы проверить наши гипотезы о том как RTX устроен внутри, мы реализовали базовый алгоритм трассировки путей и сравнили его с открытой программной реализацией трассировки путей на OpenCL в Hydra Renderer [26]. Для измерения производительности во всех наших экспериментах мы использовали 2 графические карты: GTX1070 и RTX2070. При этом известно, что в GTX1070 трассировка лучей внутри Vulkan реализована программно, в то время как RTX2070 имеет аппаратное ускорение.

Эксперимент 1. В первом эксперименте мы измеряли время для различной глубины трассировки лучей (из которого последовательным вычитанием мы получали время для разных отскоков), и из него оценивали количество лучей в секунду для каждого отскока по формуле 4.1:

$$rays = \frac{width * height * spp}{t_s} \tag{4.1}$$

При помощи ПО Nvidia Nsight Graphics мы замеряли время вызова “vkCmdTraceRaysNV” в Vulkan и сравнивали его с временем, потраченным на выполнение вычислительного ядра OpenCL “BVH4TraversalInstKernel” в Hydra Renderer [26]. В этом эксперименте нас интересовала производительность для различных случаев: когерентные (первичные) и расходящиеся лучи (вторичные и третичные, рис. 1), а также зависимость скорости от сложности геометрии (рис. 2, таблицы 1, 2).

Эксперимент 2. Наш следующий эксперимент ставил своей целью проверить наличие внутреннего механизма распределения нерегулярной работы, когда некоторые лучи порождают много дочерних лучей, а некоторые – мало, либо не порождают их вовсе. Такая рекурсивная трассировка путей



Sponza (66 K треугольников) Cry Sponza (262 K треугольников)

Sun Miguel (11 M треугольников) Hairballs (224 M треугольников)

Рис. 2. Тестовые сцены; Sponza и Crysponza – сцены с низкой детализацией и преимущественно прямоугольной геометрией. San Miguel содержит большое количество непрямоугольных форм и наиболее близка к практике. Hairballs интенсивно использует инстансинг, при этом базовый меш состоит из неудобных для BVH дерева геометрических форм – тонких волосков.

является в некотором смысле традиционным “вызовом” для GPU реализаций, поскольку наивная реализация этого алгоритма на GPU через стэк в единственном вычислительном ядре чрезвычайно неэффективна [20, 21]. Для того чтобы получить наглядные результаты, мы провели эксперимент следующим образом: в raygen-шейдере мы пускали случайно от 10 до 40 лучей и измеряли падение производительности. Далее, мы провели аналогичный эксперимент с обыкновенными вычислениями, когда некоторые тяжелые вычисления (например, шум перлина) мы также выполняли случайно от 10 до 40 раз (рис. 4а).

Эксперимент 3. В этом эксперименте мы поставили своей целью проверить наличие в RTX внутренних очередей, передающих данные между различными стадиями конвейера трассировки лучей. Для этого мы последовательно увеличивали полезную нагрузку (payload) для луча и измеряли падение производительности в процентах, чтобы понять, в какой момент передача данных становится узким местом (рис. 5).

5. ВЫВОДЫ

Вывод 1. Nvidia RTX в первую очередь нацелена на ускорение случайного доступа к памяти во время трассировки большого числа расходящихся лучей. Этот вывод вытекает из рис. 3, справа. На небольшой сцене (Sponza) аппаратная реали-

Таблица 2. Миллионы лучей в секунду для разрешения 1024 × 1024 и 1 сэмпла на пиксел (spp), видеокарта RTX2070 (аппаратно ускоренная реализация RTX)

Сцена	перв.	втор.	трет.
Sponza, RTX	**970**	**534**	**490**
Sponza, Hydra	480	122	130
Crysponza, RTX	**788**	**386**	**337**
Crysponza, Hydra	276	92	80
San Miguel, RTX	**286**	**180**	**151**
San Miguel, Hydra	127	48	42
Hairballs, RTX	**282**	**238**	**289**
Hairballs, Hydra	61	50	56

зация Nvidia RTX на первичных (когерентных) лучах выигрывает у открытой программной реализации из [26] не более чем в 2 раза. Однако, уже на вторичных лучах эта разница достигает 5–6 раз. Кроме того, на тяжелой сцене (Hair Balls) RTX показывает то же преимущество в 4–5 раз, и тот факт, что ускорение сохраняется для сцен, где память является узким местом, подтверждает наше предположение.

Вывод 2. RTX реализует некоторый механизм группировки лучей. Это подтверждается анализом падения производительности трассировки лучей (в процентах или разах), представленном на рис. 1. Можно заметить следующие тенденции: во-первых, аппаратная реализация (rtx2070, первый столбец на рис. 1) существенно лидирует над всеми программными реализациями и ни на одной сцене не замедляется более чем в 2 раза. Во-вторых, на сцене Hairballs, где группировка лучей не может помочь в принципе в силу высокой сложности геометрии, аппаратная реализация и открытые программные реализации (столбцы hydra2070 и hydra1070), не выполняющие группировку лучей, ведут себя одинаково и не теряют в производительности существенно. При этом программная реализация RTX от Nvidia (gtx1070) демонстрирует неожиданное поведение: на простой сцене Sponza она выигрывает, но на остальных, более сложных, существенно проигрывает открытой реализации (то есть имеет существенно больший процент падения производительности при переходе к вторичным лучам). Такой результат может быть вызван одной из двух причин:

1) Если RTX в программной реализации (на GTX1070) выполнен в виде монолитного вычислительного ядра, тогда выигрыш на простой сцене является следствием сниженных накладных расходов, т. к. не нужно передавать данные между разными ядрами; при этом проигрыш на сложных сценах – прямое следствие известных недостатков убер-ядер [20, 21].

2) Если RTX в программной реализации (на GTX1070) выполнен в виде аналога “wavefront pathtracing”, тогда без должной аппаратной поддержки распределения работы этот подход, по-видимому, недостаточно эффективен.

Мы считаем первый сценарий наиболее вероятным, однако, поскольку RTX является закрытой реализацией, исключать второй вариант полностью нельзя.

Вывод 3. RTX реализует некоторый внутренний механизм распределения нерегулярной работы. Этот механизм по-видимому работает по принципу, похожему на “wavefront path tracing” [21]. Этот вывод подтверждается следующим наблюдением во время эксперимента номер 2: когда на каждый пиксел мы сгенерировали случайное (от 10 до 40) количество лучей, мы получили замедление в 2 раза по сравнению с 10 лучами. С другой стороны, когда мы повторили тот же эксперимент для вычисления шума Перлина, мы получили замедление ровно в 4 раза, как и должно быть на GPU из-за того что все потоки в SIMD группе warp должны ожидать завершения самого медленного (рис. 4а).

Вывод 4. RTX в действительности реализует передачу данных между разными стадиями кон-

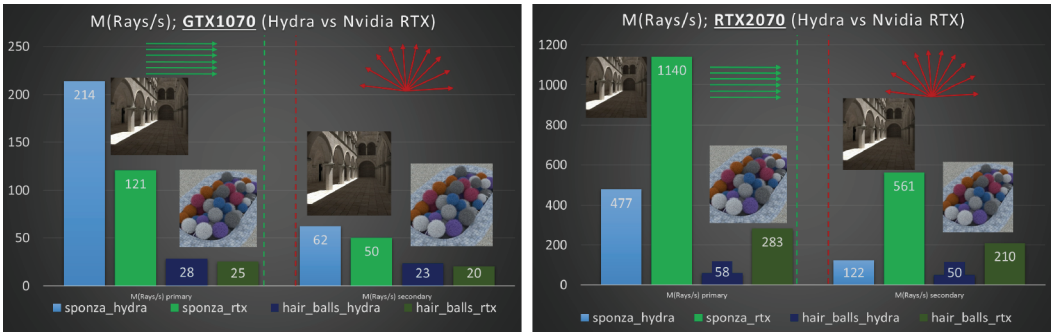
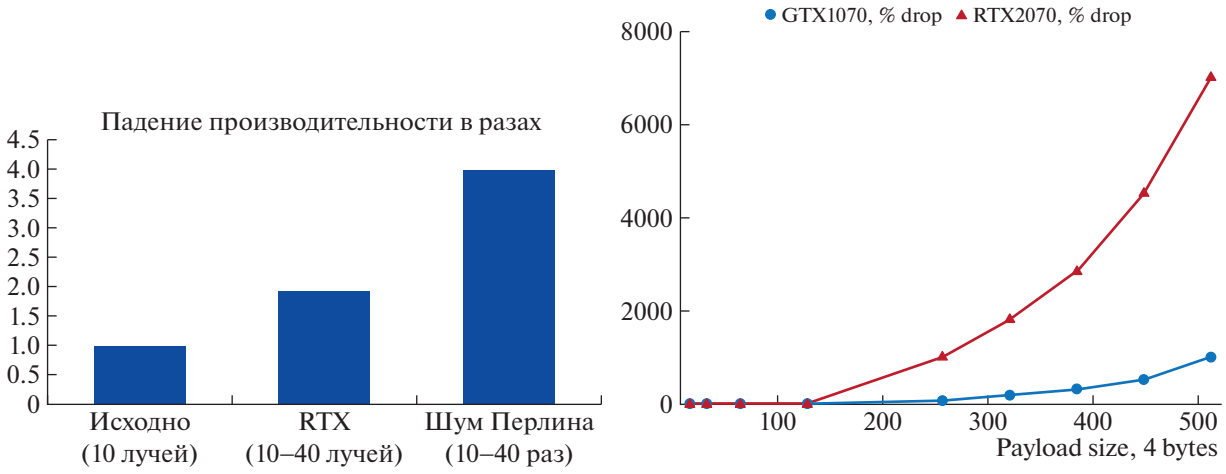


Рис. 3. Сравнение на GTX1070 (слева) и RTX2070 (справа), открытая реализация трассировки путей в HydraRenderer против Nvidia RTX. На каждом изображении его левая часть (слева от двух параллельных пунктирных линий) показывает производительность для первичных (когерентных лучей). Правая часть (справа от двух параллельных пунктирных линий) показывает производительность на вторичных, расходящихся лучах.



(а) Падение производительности для цикла со случайным количеством итерации (от 10 до 40)

(б) Падение производительности (в процентах) в зависимости от полезной нагрузки для луча (в байтах)

Рис. 4. Устойчивость производительности в различных экспериментах.

вейера (т.е. разными пользовательскими программами) через очереди на чипе. Это подтверждается характером падения производительности при увеличении полезной нагрузки на луч (рис. 5). Это дополнительно подтверждается появлением “Mesh шейдеров” в RTX картах.

Вывод 5. Nvidia RTX – это чрезвычайно тяжелая технология, которую трудно эффективно реализовать программно. Этот вывод подтверждается низкой эффективностью программной реализации RTX от самой компании Nvidia на видеокарте GTX1070, которая проигрывает простой открытой программной реализации трассировки лучей в 2–3 раза (таблица 1). Низкая производительность в данном случае, вероятно, является следствием высокой гибкости и стремлением сделать как можно более общую технологию, которая без должной аппаратной поддержки работает медленно.

6. ЗАКЛЮЧЕНИЕ

Технология Nvidia RTX – это довольно общий механизм, сочетающий в себе различную аппаратную функциональность, которая может быть использована не только в трассировке лучей, но и в других приложениях (в качестве примера такого использования можно привести работу [24]). Основные используемые механизмы это: (1) упорядочивание случайного доступа к памяти во время трассировки расходящихся лучей и (2) механизм создания работы на GPU, включающий в себя (3) передачу данных между разными вычислительными ядрами через кэш на чипе. Для пользователя RTX во многом упрощает разработку и предоставляет большую гибкость. С другой стороны, эта технология существенно ограничивает переносимость, поскольку RTX реализован как отдельный тип конвейера в Vulkan, и возможность использовать разработанный под RTX код каким-либо еще образом практически отсутствует. Данная проблема частично решается в DirectX12 (DXR Tier 1.1) за счет “inline” трассировки лучей, позволяющей использовать RTX и в “традиционных” конвейерах, но само по себе использование DirectX12 снижает переносимость еще больше.

СПИСОК ЛИТЕРАТУРЫ

1. *Meibner M. et al.* VIZARD II: a reconfigurable interactive volume rendering system // ACM Eurographics Proceedings of High-Performance Graphics on Graphics hardware, Eurographics Association. 2002. P. 137–146.
2. *Pfister H. et al.* The VolumePro real-time ray-casting system // Computer graphics and interactive techniques. N.Y.: Association for Computing Machinery, 1999. P. 251–260.
3. *Schmittler J., Wald I., Slusallek P.* SaarCOR: a hardware architecture for ray tracing // ACM Special Interest Group on Computer Graphics conference on Graphics hardware, Eurographics Association. 2002. P. 27–36.
4. *Schmittler J. et al.* Realtime ray tracing of dynamic scenes on an FPGA chip // ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware. ACM. 2004. P. 95–106.
5. *Hall D.* The AR350: Today’s ray trace rendering processor // Eurographics/SIGGRAPH workshop on Graphics hardware – Hot 3D Session 1. 2001.
6. *Seiler L.* et al. Larrabee: a many-core x86 architecture for visual computing // ACM Transactions on Graphics. V. 7. № 3, Article 18 (August 2008), 15 pages.

7. TRaX: *Spjut J.* et al. A multi-threaded architecture for real-time ray tracing // Symposium on Application Specific Processors, Institute of Electrical and Electronics Engineers. 2008. P. 108–114.
8. *Kopta D. et al.* An energy and bandwidth efficient ray tracing architecture // High-performance Graphics, ACM. 2013. P. 121–128.
9. *Woop S., Schmittler J., Slusallek P.* RPU: a programmable ray processing unit for realtime ray tracing // ACM Transactions on Graphics (TOG), ACM, 2005. V. 24. № 3. P. 434–444.
10. *Aila T., Karras T.* Architecture considerations for tracing incoherent rays // High-performance Graphics, Eurographics Association. 2010. P. 113–122.
11. *Nocak J., Havran V.* and Daschbacher. Path regeneration for interactive path tracing // EUROGRAPHICS 2010, short papers. P. 61–64.
12. *Shkurko K. et al.* Dual streaming for hardware-accelerated ray tracing // High Performance Graphics, ACM. 2017. P. 12.
13. *Фролов В.А., Галактионов В.А.* Регенерация путей с низкими накладными расходами // Программирование. 2016. № 6. С. 67–74. English translation: V.A. Frolov, V.A. Galaktionov. Low overhead path regeneration // Programming and Computer Software. 2016. V. 42. № 6. P. 382–387.
14. *Keely S.* Reduced precision hardware for ray tracing // Proceedings of High-Performance Graphics. 2014. P. 29–40.
15. *Nah J.H.* et al. RayCore: A ray-tracing hardware architecture for mobile devices // ACM Transactions on Graphics (TOG), ACM. 2014. V. 33. № 5. P. 162.
16. *Lee W. J. et al.* SGRT: A mobile GPU architecture for real-time ray tracing // High-performance graphics Proceedings of High-Performance Graphics, ACM. 2013. P. 109–119.
17. *Whitted T.* An improved illumination model for shaded display // ACM Special Interest Group on Computer Graphics and Interactive Techniques. 1979. V. 13. № 2. P. 14.
18. *Deng Y. et al.* Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques // ACM Computing Surveys (CSUR). 2017. V. 50. № 4. P. 58.
19. Imagination technologies., PowerVR Ray Tracing, 2019, https://www.imgtec.com/graphics-processors/architecture/powervr-ray-tracing/
20. *Фролов В.А., Харламов А.А., Игнатенко А.В.* Смещенное решение интегрального уравнения светопереноса на графических процессорах при помощи трассировки путей и кэша освещенности // Программирование. 2011. Т. 37. № 5. English translation: V. A. Frolov. A. A. KharlamovA. V. Ignatenko. Biased solution of integral illumination equation via irradiance caching and path tracing on GPUs // Programming and Computer Software. 2011. V. 37.
21. *Laine S., Karras T., Aila T.* Megakernels considered harmful: wavefront path tracing on GPUs // Proceedings of the 5th High-Performance Graphics Conference (HPG '13), ACM, New York, NY, USA. P. 137–143.
22. Microsoft. DirectX, спецификация трассировки лучей (DXR), 2019, https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html
23. Viitanen Timo. Acceleration data structure hardware (and software), Jul 27, 2019, Special Interest Group on Computer Graphics and Interactive Techniques, LA, USA.
24. *Wald I.* et al. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. Authors' Preprint – to be presented at High-Performance Graphics 2019.
25. Nvidia. RTX Ray tracing developer resources, 2019, https://developer.nvidia.com/rtx/raytracing
26. Ray Tracing Systems, Keldysh Institute of Applyed Mathematics, Moscow State Uiversity. Hydra Renderer. Open source rendering system, 2019, https://github.com/Ray-Tracing-Systems/HydraAPI