ПРОГРАММИРОВАНИЕ, 2015, № 4, с. 23–39

УДК 004.45

ВЫСОКОУРОВНЕВАЯ МОДЕЛЬ ПАМЯТИ ПРОМЕЖУТОЧНОГО ЯЗЫКА JESSIE С ПОДДЕРЖКОЙ ПРОИЗВОЛЬНОГО ПРИВЕДЕНИЯ ТИПОВ УКАЗАТЕЛЕЙ *

© 2015 г. М.У. Мандрыкин, А.В. Хорошилов
Институт системного программирования РАН
109004 Москва, ул. А.Солженицына, 25
E-mail: khoroshilov@ispras.ru, mandrykin@ispras.ru
Поступила в редакцию 25.11.2014

В статье представлен промежуточный язык, предназначенный для использования в качестве целевого анализируемого языка при верификации промышленного кода на языке GNU C (в частности, модулей ядра Linux). Язык представляет собой расширение существующего промежуточного языка, используемого подключаемым модулем Jessie в системе статического анализа Frama-C. Он имеет семантику, совместимую с семантикой языка Си (в частности, для массивов), изначально поддерживает различаемые объединения и префиксные (иерархические) приведения типов указателей на структуры, и расширен ограниченной поддержкой низкоуровневого приведения типов указателей. Подходы к трансляции исходного Си-кода в промежуточный язык, а также трансляции промежуточного языка во входной язык платформы дедуктивной верификации Why3 рассматриваются на примерах. Эти примеры иллюстрируют выразительность расширенного промежуточного языка и эффективность получаемых аксиоматических спецификаций.

1. ВВЕДЕНИЕ

Существует множество подходов к статической верификации Си-программ. Среди них есть дедуктивная верификация, которая основана на переводе исходного кода на языке Си, аннотированного спецификациями проверяемых свойств, во множество логических формул, общезначимость (или невыполнимость) которых эквивалентна корректности (в смысле отсутствия нарушений спецификации) исходной программы в соответствии с заданными свойствами. Эти логические формулы, также известные как *условия верификации* (УВ, англ. *verification conditions* (VCs), а также *proof obligations*) могут быть проверены на выполнимость (разрешены) с помощью различных инструментов. Эти инструменты могут быть автоматическими, такими как SAT- и SMT-решатели [1, 2] (Z3, CVC3, CVC4, Alt-Ergo и др.), а также доказатели теорем на основе обобщенного резолютивного вывода (Vampire, E-Prover и др.), или требующими участия со стороны пользователя, такими как интерактивные средства доказательства теорем Coq и PVS.

Таким образом, любой инструмент дедуктивной верификации для языка Си, целью которого служит максимальная автоматизация проверки заданных спецификаций, должен каким-либо образом преобразовывать семантику исходного аннотированного кода во множество УВ, которые способны разрешать современные автоматические инструменты проверки выполнимости.

В то же время основные свойства языка Си существенно усложняют подобное преобразование. Во-первых, Си — это императивный язык программирования, и, как следствие, неявное состояние Си-программы должно каким-либо образом моделироваться в получаемых логических

*Исследование проводилось при финансовой поддержке Министерства образования и науки Российской федерации (уникальный идентификатор проекта – RFMEFI60414X0051).

формулах. Его можно моделировать несколькими способами, которые обычно так или иначе предполагают использование теории массивов для представления состояния памяти программы. Во-вторых, Си предполагает ручное управление памятью, что требует генерации дополнительных проверок условий корректности, связанных с возможными ошибками управления памятью (преждевременное очищение, мусор, обращение за пределы выделенной памяти). В третьих, Си — это низкоуровневый язык программирования, имеющий в общем случае нетипизированную семантику, что может существенно влиять на выбор подходящей модели памяти.

Рассмотрим очень простой пример аннотированного фрагмента Си-программы (рис. 1):

```
1   int *a, *b, d[4], e[4];
2   char c[4];
3   int n, m;
4   // ...
5   a = e;
6   b = a;
7   b[n] = 1;
8   d[m] = 2;
9   c[1] = 'a';
10  //@ assert a[n] ≡ 1;
```

Рис. 1. Пример аннотированного фрагмента Си-программы.

Предположим, что требуется получить УВ для проверки условия `a[n]` $\equiv 1$ в строке `10`. Наиболее прямолинейным и одновременно точным решением будет использование для построения соответствующей логической формулы низкоуровневой модели, в которой каждая ячейка памяти программы представлена некоторым количеством последовательных индексов в одном общем логическом байтовом (8-битном) массиве M, индексированном, например, 32-битными адресами. Логические массивы [2] (те, с которыми работают SMT-решатели) — это тотальные отображения из значений типа индекса массива в значения типа элемента массива, для которых определена тернарная операция обновления $\cdot[\cdot \leftarrow \cdot]$ без побочных эффектов для троек (массив, индекс, значение), которая для указанного массива возвращает новый логический массив того же типа, отличающийся от исходного тем и только тем, что он отображает указанный индекс в указанное новое значение (то есть для остальных индексов значения остаются без изменения). В логических формулах задаются только ограничения на значения массивов по некоторым индексам, изначально значения элементов массивов считаются неопределенными (недетерминированными). Взятие значения элемента массива M по индексу i будем обозначать через $M[i]$. Для обозначения изменений массива M в результате выполнения операций записи в память будем использовать (конечную) последовательность логических массивов $M_1, M_2, \ldots, M_n$. Пусть адреса переменных `a`, `b`, `c`, `d`, `e`, `n` и `m`, представлены неинтерпретируемыми константами — 32-битными векторами a, b, c, d, e, n и m. Тогда соответствующее УВ в низкоуровневой модели памяти запишется в виде:

$$
\begin{aligned}
&M_1 = M_0\big[a \leftarrow M_0[e]\big] \wedge \\
&M_2 = M_1\big[b \leftarrow M_1[a]\big] \wedge \\
&M_3 = M_2\big[M_2[b] +_{32} M_2[n] \leftarrow 0_8\big] \wedge \\
&M_4 = M_3\big[M_3[b] +_{32} M_3[n] +_{32} 1_{32} \leftarrow 0_8\big] \wedge \\
&M_5 = M_4\big[M_4[b] +_{32} M_4[n] +_{32} 2_{32} \leftarrow 0_8\big] \wedge \\
&M_6 = M_5\big[M_5[b] +_{32} M_5[n] +_{32} 3_{32} \leftarrow 1_8\big] \wedge \\
&M_7 = M_2\big[M_6[d] +_{32} M_6[m] \leftarrow 0_8\big] \wedge \ldots \wedge \\
&M_{10} = M_5\big[M_9[d] +_{32} M_9[m] + 3_{32} \leftarrow 2_8\big] \wedge \\
&M_{11} = M_{10}\big[M_{10}[c] \leftarrow 97\big] \wedge \\
&\left.\begin{aligned}
&(b \le a - 4 \vee b \ge a + 4) \wedge \ldots \wedge \\
&(e \le a - 16 \vee e \ge a + 4) \wedge \ldots \wedge \\
&(e \le d - 16 \vee e \ge d + 16)
\end{aligned}\right\} \text{42 неравенства}
\end{aligned}
$$

Низкоуровневая модель позволяет моделировать семантику Си-программ с очень высокой побитовой точностью, в перспективе (при еще более точном моделировании памяти, например с учетом регистров и кэшей) достигая практически полного соответствия модели реальному выполнению программы на какой-либо физической машине. Логические формулы, использующие одновременно теории массивов и битовых векторов, в значительном числе случаев поддаются разрешению с помощью современных автоматических инструментов проверки выполнимости (SMT-решателей), хотя существующие алгоритмы для теории массивов в общем случае не полны [2] (то есть соответствующие инструменты могут завершаться, не выдавая вердикт

"*выполнимо*"/"*невыполнимо*" или вовсе не завершаться). Однако даже на приведенном простейшем примере видно, что логические формулы, построенные в низкоуровневой модели памяти, весьма избыточны и неоптимальны. На практике такая модель памяти плохо масштабируется и не используется в инструментах дедуктивной верификации в изначальном виде — при построении формул всегда применяются некоторые существенные оптимизации.

1.1. *Существующая модель памяти* Jessie

Здесь следует упомянуть, что в данной статье речь идет только о построении логических формул в классической логике первого порядка с равенством и с использованием некоторых теорий (таких как теории массивов, неинтерпретируемых функций, битовых векторов и линейной вещественной или целочисленной арифметики), то есть так называемых *SMT-формул* (от англ. Satisfiability Modulo Theories). Альтернативные техники, в первую очередь, основанные на использовании различных модификаций логики разделения (англ. separation logic) [3], отличаются, в целом, менее развитой поддержкой со стороны инструментов автоматического разрешения задач выполнимости соответствующих формул и, соответственно, обычно требуют существенно большего участия со стороны пользователя при проверке получаемых УВ. В области автоматизированной дедуктивной верификации техники, основанные на построении SMT-формул, остаются наиболее часто используемыми на практике, в частности, в инструментах ESC/Java [4], Spec# [5], VCC [6], Frama-C/Jessie [7, 8], Frama-C/WP [7] и др.

Первая, наиболее распространенная и очевидная оптимизация, используемая при построении SMT-формул — это выделение в исходной Си-программе так называемых *чистых переменных*, то есть переменных простых (не составных) типов данных, обращение к которым может происходить только по их именам, в отличие от общего случая, когда обращение к переменной может происходить также, например, по указателю, содержащему адрес этой переменной. В частности, в качестве чистых переменных часто рассматриваются переменные, для которых в исходном коде отсутствуют операции взятия адреса (`&`). Такое предположение является надежным при условии того, что для исходной программы каким-либо образом доказана корректность работы с памятью (в частности, отсутствие обращений за границы выделенных областей). Так как обращение к чистым переменным может происходить только по их именам, обращения к каждой чистой переменной можно легко статически отделить от обращений как к другим чистым переменным, так и к другим объектам в памяти программы. Поэтому для каждой чистой переменной вместо индекса (адреса) в последовательности массивов $M_1, \ldots, M_n$ можно использовать отдельную последовательность переменных $v_1, v_2, \ldots v_k$. Одна только эта оптимизация позволяет уменьшить число неравенств (задающих непересечение (разделение) различных объектов в памяти) в приведенном примере формулы с 42 до 6, а также уменьшить число переменных в последовательности состояний памяти $M_1, \ldots, M_n$ с 11 до 9.

Идею отделения некоторых индексов из общего массива M, примененную для чистых переменных, можно обобщить на произвольные априорно (с точностью до некоторой статической аппроксимации) разделенные области памяти. Для этого можно выделить набор непересекающихся множеств объектов в памяти, таких, что для любого указателя в любой точке программы гарантированно можно указать то единственное множество объектов из этого набора, которое данный указатель может адресовать (то есть указывать на объекты только из этого множества). Такие непересекающиеся множества объектов в памяти будем в дальнейшем называть *регионами*, по непосредственной аналогии с регионами, изначально предложенными в [9, 10] для изложения техники автоматического управления динамической памятью без использования сборщика мусора (так называемого управления памятью на основе регионов, англ. region-based memory managenment). В этих статьях не предложен конкретный алгоритм вывода регионов (хотя и предложены соответствующие не алгоритмические правила их вывода), однако в статье [11] предложен практический алгоритмический метод, позволяющий статически вычислять чувствительные к контексту, но не к потоку, приближен-

ные разделения памяти Си-программ на регионы. Этот метод основан на применении некоторого набора правил объединения регионов к изначальному разбиению, в котором предполагается, что каждая переменная-указатель в исходной программе образует отдельный регион. При этом результатом алгоритма являются метки регионов, приписанные строго по одной к каждому указательному выражению исходной программы. Для применения разделения на регионы к Си-программам в общем случае в статье предлагается использовать предварительную *нормализацию* их исходного кода, которая в основном включает в себя удаление операций взятия адреса с помощью перетипирования адресуемых переменных в указатели и приведение трех операторов косвенного доступа языка Си (`*`, `[]` и `.`) к одному единственному оператору `->` с использованием перетипирования в указатели, адресной арифметики и фиктивных структур с одним полем для представления простых типов данных. Применение нормализации к фрагменту Си-программы на рис. 1 дает нормализованную Си-программу, показанную на рис. 2.

```
1   struct intP { int intM; };
2   struct charP { char charM; };
3   // ...
4   struct intP *a, *b, d[4], e[4];
5   struct charP c[4];
6   int n, m;
7   // ...
8   a = e;
9   b = a;
10  (b + n)->intM = 1;
11  (d + m)->intM = 2;
12  (c + 1)->charM = 'a';
13  //@ assert (a + n)->intM ≡ 1;
```

Рис. 2. Нормализация аннотированного фрагмента Си-программы.

Разбиение нормализованной программы на регионы позволяет значительно упростить (уменьшить) полученное ранее условие верификации. В приведенном примере вся память программы разделяется на три региона — один общий регион для указателей `a`, `b` и `e`, один регион для указателя `d` и один регион — для указателя `c`. Для каждого региона можно использовать отдельный логический массив, получая в результате, с учетом ранее рассмотренной оптимизации чистых переменных, следующее УВ:

$$
\begin{array}{l}
a_1 = e_0 \wedge \\
b_1 = a_1 \wedge \\
M_{a,b,e,1} = M_{a,b,e,0}[b_1 +_{32} n_0 \leftarrow 0_8] \wedge \ldots \wedge \\
M_{a,b,e,4} = M_{a,b,e,3}[b_1 +_{32} n_0 +_{32} 3_{32} \leftarrow 1_8] \wedge
\end{array}
$$

$$
\begin{array}{l}
M_{d,1} = M_{d,0}[d_0 +_{32} m_0 \leftarrow 0_8] \wedge \ldots \wedge \\
M_{d,4} = M_{d,3}[d_0 +_{32} m_0 +_{32} 3_{32} \leftarrow 2_8] \wedge \\
M_{c,1} = M_{c,0}[c_0 \leftarrow 97]
\end{array}
$$

В этом условии верификации всего по 4 переменных-массива в каждой последовательности состояний регионов памяти и вовсе отсутствуют неравенства, обеспечивающие непересечение адресов выделенных объектов в памяти, которые оказались избыточны после разбиения памяти программы на регионы.

Однако простое применение разбиения на регионы к верификации достаточно больших программ на языке Си сталкивается с трудностями масштабирования — в больших программах регионы часто объединяются, в основном за счет большого числа вызовов функций с различными комбинациями параметров-указателей. Решение проблемы масштабируемости было предложено в статье [12], в которой вводится понятие *полиморфного региона*, то есть региона, область видимости которого ограничивается одной компонентой сильной связности в графе вызовов функций (то есть одной функцией или несколькими при наличии между ними косвенной рекурсии). Разделение на полиморфные регионы производится отдельно для каждой компоненты сильной связности, что значительно улучшает масштабируемость всего подхода. Сами полиморфные регионы в дальнейшем трактуются как дополнительные формальные параметры соответствующих функций, в качестве которых при вызовах могут передаваться фактические параметры — регионы из вызывающих функций. При этом, однако, требуется проверка дополнительных предусловий, потому что вызывающая функция может передать в качестве двух разных фактических параметров вызываемой функции один и тот же регион, тем самым нарушив предположения, сделанные при выводе регионов в компоненте сильной связанности вызываемой функции. В приведенном примере, если считать `d`, `e` и

`c` формальными параметрами некоторой функции, соответствующие предусловия принимают следующий вид:

$$(e_0 \le d_0 - 16 \vee e_0 \ge d_0 + 16)\wedge$$
$$(e_0 \le c_0 - 16 \vee a_0 \ge c_0 + 4)\wedge$$
$$(d_0 \le c_0 - 16 \vee d_0 \ge c_0 + 4).$$

Полученные с использованием рассмотренных оптимизаций УВ можно еще больше упростить, введя ограничение на исходный код верифицируемых программ. Если известно, что в исходной программе указатели разных типов никогда не могут быть синонимами, то есть указывать на один и тот же объект в памяти, то разделенность соответствующих регионов для указателей разных типов можно предполагать глобально для всей программы, упрощая тем самым предусловия вызываемых функций. В таких предположениях вполне естественно отказаться от низкоуровнего побайтового представления логических массивов для регионов памяти и перейти к их более эффективному высокоуровневому представлению с использованием массивов логических (то есть неограниченных, математических) целочисленных типов и модульной арифметики:

$a_1 = e_0\wedge$
$b_1 = a_1\wedge$
$M_{a,b,e,1} = M_{a,b,e,0}\big[(b_1 + n_0) \bmod 4\,294\,967\,296 \leftarrow 1\big]\wedge$
$M_{d,1} = M_{d,0}\big[(d_0 + m_0) \bmod 4\,294\,967\,296 \leftarrow 2\big]\wedge$
$M_{c,1} = M_{c,0}\big[c_0 \bmod 4\,294\,967\,296 \leftarrow 97\big]\wedge$
$e_0 \le d_0 - 4 \vee e_0 \ge d_0 + 4$ (предусловие).

В результате получается высокоуровневая модель памяти, которую можно использовать для верификации программ не только с переменными простых типов, но также и программ с массивами, структурами, в том числе с поддержкой префиксного приведения типов (кастирования)[1], и различаемыми объединениями[2]. Соответствующие подходы описаны в статьях [13, 14] и [15].

Все рассмотренные оптимизации применены в инструменте дедуктивной верификации Jessie [16, 8]. Так как на практике в языке Си алиасинг практически всегда ограничен так, что отсутствуют невыровненные указатели (указатели какого-либо типа строго внутрь объекта того же типа) и пересечения объектов в памяти (объекты могут быть встроенными один в другой, как, например, вложенные структуры, но никак по-другому обычно не пересекаются), а префиксные приведения типов указателей (включая как частный случай приведение к `void *` и обратно) и различаемые объединения в совокупности, как показывают некоторые исследования [17], составляют до 99% всех приведений типов указателей в большинстве промышленных программ, модель памяти, реализованная в Jessie, оказывается на практике достаточно удобной и эффективной.

1.2. Проблемы существующей модели памяти

Вместе с этим, некоторые важные участки в исходном коде ядер операционных систем, являющихся одними из наиболее перспективных объектов для практического применения существующих техник дедуктивной верификации, остаются за пределами области применимости модели. В отличии, например, от модели памяти, используемой инструментом VCC [18], в модели памяти Jessie нет поддержки операций над битовыми полями структур и переинтерпретации памяти. В то же время, эти возможности однозначно необходимы для верификации модулей ядра Linux, в частности модулей поддержки файловых систем и сетевых протоколов, где многие низкоуровневые операции кодирования/декодирования и буферизации реализуются с использованием переинтерпретации областей памяти между массивами байтов и структур в обоих направлениях.

В кандидатской работе, описывающей методы, лежащие в основе существующей реализации инструмента Jessie [8], предложено использование уже рассмотренной техники вывода регионов (и вычисления эффектов) [19, 9, 11, 12] вкупе с комбинированием высокоуровневой типизированной

[1] *Восходящее* и *нисходящее* (англ. *upcast* и *downcast*), или, иначе, *префиксное* (англ. *prefix*) или *иерархическое* (англ. *hierarchical*) приведение типа указателя на структуру — это приведение типа указателя на одну структуру к указателю на другую структуру, при котором список полей одной из структур, составляет префикс (не обязательно собственный) списка полей другой структуры.

[2] *Различаемое объединение* (англ. *discriminated unions*) — это объединение, поля которого адресуются только через указатель на содержащее их объединение (то есть для которого отсутствуют взятия адресов от полей или соответствующие им приведения типов указателей), и при этом чтение всегда происходит только из того поля такого объединения, которое было записано в память объединения в самый последний раз.

и низкоуровневой побитовой моделей памяти путем применения к различным подмножествам, полученным в результате вывода регионов. Это позволяет соединять несколько различных моделей памяти в рамках одного инструмента верификации и при этом использовать эти модели независимо в отношении различных регионов при построении соответствующих УВ.

Слабость этого подхода проявляется при использовании современных решателей, прежде всего SMT-решателей, основанных на DPLL [2, 1]. В них обычно каждая поддерживаемая логическая теория реализуется отдельно, и связь между теориями устанавливается за счет обмена дизъюнкциями предикатов, среди которых только равенства интерпретируются всеми участвующими теориями [2]. Из этого следует (и это подтверждается на практике), что, чаще всего, логические предикаты и функции, требующие применения одновременно сразу нескольких теорий, в частности, преобразование целого в битовый вектор конечной длины или обратно, либо вовсе не поддерживаются современными решателями, либо их поддержка крайне ограничена и неэффективна. В такой ситуации все взаимосвязи между используемыми низкоуровневой и высокоуровневой моделями памяти оказывается необходимым выражать в виде сложно определяемых пользовательских предикатов, либо в виде неинтерпретируемых предикатов с соответствующими наборами аксиом. При этом вся работа по формулированию спецификаций этих предикатов и возможному ручному доказательству получаемых в результате УВ (в силу сложности предикатов) ложится на инженера-верификатора. Кроме этого, в присутствии большого количества синонимичных указателей низкоуровневые регионы могут включать в себя довольно значительные области памяти, приводя к тому, что существенная часть данных программы может быть переведена в формулы с использованием теории битовых векторов конечной длины (причем из-за возможной переинтерпретации памяти векторы будут иметь значительную длину, например, при объединении переинтерпретируемого массива структур в один битовый вектор). Это, в свою очередь, может значительно осложнять проверку выполнимости получаемых формул. Другой неудачный, но весьма распространенный случай — это адресная арифметика со смещением, закодированным в виде битового вектора, когда добавление такого смещения к указателю в каком-либо одном месте приводит к необходимости кодирования всех добавляемых к этому указателю (и его синонимам) смещений в виде битовых векторов. Кроме этого, на момент написания статьи последняя версия платформы дедуктивной верификации Why3 [20], на которой базируется инструмент Jessie, не поддерживала все необходимые операции над битовыми векторами, а реализация низкоуровневых регионов в самом Jessie была по большей части не завершена или очень ограничена.

Еще одним недостатком исходной модели памяти Jessie является то, что семантика операции сдвига указателя в ней оказалась несовместимой с аналогичной операцией в языке Си в присутствии префиксных приведений типов указателей. Исходная модель памяти Jessie разрабатывалась в целях одновременной поддержки нескольких анализируемых языков (по крайней мере, Java с JML, Си с ACSL и OCaml с некоторым аналогичным языком спецификации) и, по всей видимости, в целях упрощения представления в Jessie Java- (и OCaml-) массивов, семантика массивов промежуточного языка Jessie была выбрана соответствующей семантике массива объектов Java (или массиву любых упакованных значений в OCaml). В результате, в то время как в Си указатель на элемент массива производных структур (с бóльшим/таким же списком полей) в общем случае становится невыравненным (и, чаще всего, непригодным для дальнейшей адресной арифметики) после приведения его к указателю на базовую структуру (с меньшим/таким же списком полей), семантика исходного промежуточного языка Jessie полностью игнорирует это ограничение (в соответствии с ковариантностью массивов в Java/OCaml) и таким образом существенно ограничивает класс Си-программ, корректно транслируемых в промежуточный язык (без явного добавления значительного числа дополнительных проверок).

В итоге рассмотрения исходных моделей памяти для промежуточного языка Jessie было принято решение изменить и расширить уже

существующую эффективную высокоуровневую модель.

С целью упрощения дальнейшего изложения, в статье не рассматривается настоящий промежуточный язык Jessie, его синтаксис, типизация и семантика, которые довольно сложны, избыточны и совсем не минималистичны. Более подробное описание настоящего промежуточного языка Jessie можно найти в диссертации [8] и статье [16]. В данной статье же представлен промежуточный язык и новая соответствующая ему модель памяти в существенно редуцированном (упрощенном) виде. Этот *упрощенный промежуточный язык* (*УПЯ*) все же позволяет показать основные свойства реального языка Jessie и то, как этот язык может быть расширен частичной поддержкой низкоуровневого доступа к данным в памяти (переинтерпретирующих приведений типов указателей), позволяя относительно эффективно выразить наиболее важные случаи его использования, такие как операции кодирования/декодирования, буферизация и переупорядочение байт.

В статье вначале представлен *упрощенный промежуточный язык* (*УПЯ*), маленький игрушечный аналог настоящего промежуточного языка Jessie, затем представлена его семантика и соответствующая высокоуровневая модель памяти. После этого предложено расширение *упрощенного промежуточного языка — расширенный упрощенный промежуточный язык* (*РУПЯ*), который предоставляет поддержку некоторых низкоуровневых приведений типа указателей, сохраняя все существующие преимущества исходной высокоуровневой модели памяти. Предлагаемое расширение совместимо с битовыми полями и достаточно легко интегрируется с улучшениями, сделанными в существующей реализации инструмента Jessie, такими как ограниченные указатели, поддержка множества корней в иерархии структур и разделением памяти на регионы. Расширенный упрощенный промежуточный язык и его модель памяти по сравнению с исходным упрощенным промежуточным языком содержит изменения, соответствующие изменениям, сделанным в настоящем промежуточном языке Jessie при реализации предложенной в статье модели памяти в соответствующей модификации инструмента. В заключении обсуждаются текущие результаты этой реализации.

2. УПРОЩЕННЫЙ ПРОМЕЖУТОЧНЫЙ ЯЗЫК

2.1. Абстрактный синтаксис

Транслятор промежуточного языка Jessie работает в середине следующей цепочки инструментов верификации: Frama-C — подключаемый модуль Jessie — транслятор промежуточного языка Jessie — интегрированная среда верификации Why3. Транслятор принимает на вход специфицированную программу на промежуточном языке Jessie в виде одного файла, в котором должны содержаться все единицы трансляции исходной программы, объединенные вместе с заданными пользователем спецификациями. Промежуточный язык Jessie довольно сложный и совсем не минималистичный, поскольку он был разработан с целью упростить трансляцию исходной специфицированной программы на языке Си в этот язык, как можно меньше изменяя её структуру, но применяя при этом к ней ряд существенных преобразований, в основном касающихся упрощения (нормализации) структуры памяти (например, устранение операций взятия адреса и встраивания). Поэтому в статье не представлен собственно промежуточный язык Jessie, а использован его радикально упрощенный аналог, сходный с настоящим языком Jessie по большинству основных свойств, относящихся к используемой в языке модели памяти. Основные существенные отличия этого *упрощенного промежуточного языка* (*УПЯ*) от настоящего — отсутствие типизации и вывода регионов памяти. На практике существующий подход к выводу регионов памяти, реализованный в Jessie, может быть непосредственно применен и к УПЯ. Еще одно важное отличие УПЯ от языка Jessie состоит в уровне поддержки различаемых объединений. В УПЯ отсутствует поддержка непосредственного использования структуры в качестве одного из полей объединения, в то время как в Jessie поля объединения могут быть структурами. Это ограничение УПЯ легко преодолевается за счет некоторого усложнения (расширения) семантики операции *сильного обновления*, то есть присваива-

Целочисленный терм:

$$
\begin{array}{rll}
t_n ::= & v & \text{цел. переменная} \\
| & n & \text{цел. значение} \\
| & * & \text{недет. цел. значение} \\
| & t_n \star t_n & \text{бин. операция над цел.} \\
| & \mathbf{acc}(t_p, f) & \textit{разыменование} \\
| & \mathbf{psub}(t_p, t_p) & \textit{разность указателей}
\end{array}
$$

указательный терм:

$$
\begin{array}{rll}
t_p ::= & p & \text{переменная-указатель} \\
| & \mathbf{null} & \textit{нулевой указатель} \\
| & \mathbf{alloc}(t, t_n) & \textit{выделение памяти} \\
| & \mathbf{acc}(t_p, f) & \textit{разыменование} \\
| & \mathbf{shift}(t_p, t, t_n) & \textit{сдвиг указателя}
\end{array}
$$

терм:

$$
t_{np} ::= t_n \mid t_p \qquad \text{цел. или указ. терм}
$$

предикат:

$$
\begin{array}{rll}
\bar{p} ::= & t_n \bowtie t_n & \text{бин. отношение на цел.} \\
| & \mathbf{peq}(t_p, t_p) & \textit{равенство указателей} \\
{}^*| & \mathbf{omin}(t_p) \bowtie t_n & \textit{мин. смещение} \\
{}^*| & \mathbf{omax}(t_p) \bowtie t_n & \textit{макс. смещение} \\
{}^*| & \mathbf{tag}(t_p) = t & \textit{тег в точности равен} \\
{}^*| & \mathbf{tag}(t_p) \preceq t & \textit{тег ограничен сверху} \\
| & \bar{p} \lozenge \bar{p} & \text{логическая связка}
\end{array}
$$

операторы:

$$
\begin{array}{rll}
o ::= & v \leftarrow t_n & \text{присваивание} \\
| & p \leftarrow t_p & \text{присваивание указ.} \\
| & \mathbf{upd}(t_p, f, t_{np}) & \textit{запись в поле} \\
| & \mathbf{free}(t_p) & \textit{освобождение памяти} \\
{}^*| & \mathbf{assume}\ \bar{p} & \text{предположение} \\
{}^*| & \mathbf{assert}\ \bar{p} & \text{проверка}
\end{array}
$$

последовательность операторов:

$$
\begin{array}{rll}
s ::= & o & \text{оператор} \\
| & o; s & \text{последовательность опер.}
\end{array}
$$

Рис. 3. Абстрактный синтаксис упрощенного промежуточного языка (УПЯ).

ния в поле объединения. Абстрактный синтаксис упрощенного промежуточного языка приведен на рисунке 3.

На этом рисунке использованы следующие обозначения:

- v обозначает целочисленную переменную;
- $n \in \mathbb{Z}$ обозначает целочисленное значение;
- $*$ обозначает недетерминированное целочисленное значение. Возможность использования недетерминированных значений присутствует в УПЯ с целью имитации вызовов функций (которые отсутствуют в упрощенном языке, но присутствуют в настоящем), в частности, их следов в памяти (англ. memory footprint, то есть верхнего приближения их побочных эффектов, в том числе недоспецифицированных). Недетерминированные целочисленные значения могут заменяться произвольными целочисленными значениями при вычислении, при этом соответствующая семантика естественным образом выразима для языков, предназначенных для анализа с помощью инструментов дедуктивной верификации, основанных, например на исчислении слабейших предусловий [21];
- $\star$ обозначает произвольную бинарную операцию над целочисленными значениями;
- p обозначает переменную-указатель;
- $f \in \mathbb{F}$ используется для обозначения уникальных меток полей структур и объединений из конечного множества $\mathbb{F}$ (метки полей уникальны во всей программе, а не только в рамках одной структуры или объединения);
- $t \in \mathbb{T}$ используется для обозначения уникальных меток тегов[3] структур и объединений, а также меток полей объединений из другого конечного множества $\mathbb{T}$, не имеющего с множеством $\mathbb{F}$ общих элементов (таким образом у полей объединений имеется по две различные метки из множеств $\mathbb{F}$ и $\mathbb{T}$);
- $\bowtie$ обозначает произвольное бинарное отношение на множестве целочисленных значений;
- $\lozenge$ обозначает произвольную логическую связку;
- предикаты $\mathbf{omin}(t_p) \bowtie t_n$, $\mathbf{omax}(t_p) \bowtie t_n$, $\mathbf{tag}(t_p) = t$, $\mathbf{tag}(t_p) \preceq t$ и операторы $\mathbf{assume}\ \bar{p}$ и $\mathbf{assert}\ \bar{p}$ (на рисунке помечены звездочкой) относятся только к спецификационной части языка, поэтому они никак не

[3] При условии явного именования (нет анонимных структур или объединений) и уникальности в качестве меток можно использовать сами теги.

влияют на ход вычислений и, как следствие, не имеют вычислительной семантики.

Для всех термов, предикатов и операторов, имеющих отношение к указателям, структурам или объединениям, на рисунке 3 справа приведены краткие комментарии (выделены курсивом). Предполагается, что на множестве $\mathbb{T}$ введено отношение частичного порядка $\preceq \subseteq \mathbb{T} \times \mathbb{T}$. Это отношение образует на $\mathbb{T}$ ограниченную верхнюю полурешетку с точной верхней гранью $\top$. Отношение $\preceq$ соответствует отношению наследования (или префиксности) для структур, которое распространяется также и на объединения, и поля объединений (точнее, на соответствующие им метки) так, что (1) если список полей какой-либо структуры с меткой t_1 составляет префикс (не обязательно собственный) списка полей какой-либо структуры с меткой t_2, выполнено $t_2 \preceq t_1$, и (2) для меток полей объединений выполнено $t_f \preceq t_u$, где t_f — метка поля объединения с меткой t_u.

Упрощенный промежуточный язык не поддерживает встраивание объектов в памяти. Поэтому во время рассмотренной во введении нормализации Си-программы перед её трансляцией в промежуточный язык помимо рассмотренных преобразований, связанных с устранением операций взятия адреса (`&`) и приведения всех операций косвенного доступа языка Си к единому виду (операции `->`), выполняется устранение встраивания структур, массивов и объединений в другие структуры. В общем случае встраивание устраняется путем автоматического преобразования встроенных структур, массивов и объединений в косвенно адресуемые, явно выделяемые и освобождаемые. Таким образом, все встроенные объекты автоматически перемещаются в динамическую память. Кроме этого, так как УПЯ (как и Jessie) поддерживает композитные типы данных (массивы, структуры и объединения) только через указатели на динамическую память, все автоматические (стековые) выделения переменных композитных типов преобразуются в явные выделения и освобождения объектов в динамической памяти. В качестве особого случая специальным образом преобразуется встраивание одной структуры в другую в качестве первого поля. В этом случае список полей встроенной структуры добавляется к списку полей объемлющей структуры в качестве префикса. Между тегами встроенной и объемлющей структуры устанавливается отношение наследования (префиксности) $\preceq$, что дает возможность транслировать префиксные преобразования типов между указателями на эти структуры. В УПЯ в силу нетипизированности префиксные приведения типов указателей просто опускаются, однако в настоящем языке Jessie необходимо их явное использование.

Смысл предикатов **omin**, **omax** и **tag** объяснен в следующем разделе, рассказывающем о моделях указателей и памяти, используемых при анализе специфицированных программ на УПЯ.

2.2. *Модель памяти*

В соответствии с описанием в диссертации [8], модель памяти Jessie основана на так называемой *блочно-байтовой модели* (англ. *byte-level block memory model*), которая моделирует защиту памяти. В этой модели указатели представляются тройками (α, l, o), где α обозначает абсолютный адрес указателя, то есть значение, которое может быть потенциально получено приведением указателя к соответствующему целочисленному типу, l — это уникальная метка блока памяти, которая приписывается указателю в точке выделения блока памяти и остается неизменной при любых сдвигах этого указателя, а o — это смещение указателя относительно начала соответствующего блока (то есть блока с меткой l). Модель предполагает, что указатели, относящиеся к различным блокам памяти всегда различны, так как это свойство обеспечивается уникальностью меток соответствующих блоков. Однако абсолютные адреса двух указателей, относящихся к различным блокам памяти, могут совпадать, включая случаи с участием указателей, относящихся к ранее освобожденным блокам памяти (как показано на рисунке 4).

Каждому блоку памяти в модели приписывается его текущая длина, обозначаемая $\mathrm{A}[l]$, которая может меняться в ходе вычисления. Указатель считается *действительным* (англ. *valid*), тогда и только тогда, когда его смещение удовлетворят соотношениям $0 \le o < \mathrm{A}[l]$. Корректная (проходящая проверки выполнимости всех УВ) программа может разыменовывать только действительные указатели, и таким образом читать

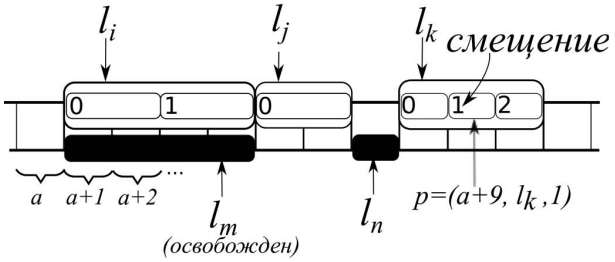

Рис. 4. Блочно-байтовая модель памяти.

либо записывать только из/в память, доступную через эти действительные указатели, которая соответствует памяти, выделенной этой программе операционной системой (либо основной частью ядра, реализующей подсистему управления памятью) через соответствующий интерфейс. В рассматриваемой модели памяти, однако, не делается различий между областями памяти, доступными только на чтение или только на запись (или на то и другое) статически, автоматически или динамически выделенными областями, а также между областями памяти в адресных пространствах пользователя или ядра. Вычитание указателей определено только для указателей, относящихся к одному и тому же блоку памяти, а сравнение на равенство также допускается для любых двух действительных в данный момент выполнения программы указателей (то есть там, где семантика модели совпадает с соответствующей семантикой языка Си).

Однако, как было упомянуто во введении, сама по себе такая модель памяти не совместима с поддержкой префиксных приведений типов указателей на структуры при сохранении семантики арифметики указателей, принятой в языке Си. Эта несовместимость появляется из-за возможного появления невыравненных указателей в результате ненулевого сдвига указателей на элементы массивов после выполнения над ними префиксного приведения типа.

Поэтому в модели памяти дополнительно не допускается появление невыравненных указателей за счет введения еще одного атрибута, относящегося к самим указателям, а именно метки (англ. *tag*) указателя, обозначаемой $\mathrm{T}[(\alpha, l, o)]$. Метка указателя выражает точный динамический тип структуры или поля различаемого объединения, адресуемого данным указателем в данный момент выполнения программы. Метки указателей могут изменяться только в результате выделений или освобождений памяти, а также выполнения операторов присваивания значений полям различаемых объединений (т.н. *усиленные присваивания*, англ.*strong updates*). Таким образом, метки остаются неизменными при выполнении операций префиксного приведения типа указателя на структуру, которые, следовательно, можно исключить из УПЯ. Сдвиги указателей в таком случае, напротив, предваряются проверками соответствующих тегов с целью предотвращения нарушения требования выравненности указателей. Именно по этой причине метки структур и объединений требуется явно указывать в операциях сдвига в УПЯ.

Отсутствие операций приведения типа для указателей также определяет необходимость соответствующих динамических проверок при разыменованиях, однако этих проверок удается избегать в реализации настоящего языка, где явные операции приведения типа сделаны обязательными с помощью соответствующей системы типов, и для этих операций введены соответствующие необходимые проверки.

Отсутствие невыравненных указателей позволяет перейти к эффективному высокоуровневому представлению памяти в виде набора раздельно обновляемых массивов-"памятей" отдельно для каждого поля структуры или объединения [15, 22], но только при условии отказа от соответствия семантике языка Си в отношении не-префиксных приведений типов указателей.

В абстрактном синтаксисе УПЯ, рассмотренном в предыдущем разделе, отсутствуют какие-либо возможности преобразования значений указателей в целочисленные значения (или обратно), поэтому далее при обозначении указателей будем опускать обозначения абсолютных адресов α (т.к. абсолютные адреса не могут, таким образом, существенно повлиять на ход или результаты вычислений).

2.3. *Семантика*

УПЯ (как и язык Jessie) предназначен для мнимого исполнения на воображаемой недетерминированной машине. Все потенциально возможные выполнения программы на такой машине ограничиваются операторами предположе-

ния (**assume**), получаемыми путем перевода аннотаций на языке ACSL, предоставляемых пользователем, в промежуточный язык. Затем полученное множество неявным образом анализируется с помощью генерации и последующего разрешения УВ, соответствующих условиям корректности операторов, выполняющихся в ходе вычислений, а также предусловиям и проверочным условиям, представленным в виде операторов проверки условия (**assert**) и полученными изначально от пользователя в виде аннотаций на языке ACSL.

Промежуточный язык разрабатывался с целью иметь легко анализируемую семантику скорее, чем легко исполнимую, что объясняет преимущество, получаемое от преобразования в него исходной Си-программы. Семантика промежуточного языка предоставляет генератору УВ и, как следствие, в дальнейшем, решателю, эффективное представление большинства условий разделения областей памяти за счет кодирования всех операций чтения и записи в/из памяти соответствующими операциями выборки (select) и сохранения (store) над различными независимыми логическими массивами, по одному массиву на каждое поле структуры или объединения [15, 22].

Вся (динамическая) память программы на промежуточном языке представляется отображением $\mathbf{M} : \mathbb{P} \to \mathbb{M}_n \cup \mathbb{M}_p$, где $\mathbb{M}_n = \{\mathrm{M} : \mathbb{F} \to \mathbb{Z}\}$, а $\mathbb{M}_p = \{\mathrm{M} : \mathbb{F} \to \mathbb{P}\}$, $\mathbb{P} = \{(l, o) \mid l \in \mathbb{L}, o \in \mathbb{Z}\}$ — это множество указателей, а $\mathbb{L} \simeq \mathbb{N} \cup \{0\}$ — это счетное множество различных (уникальных) меток выделяемых блоков памяти, включая специальную метку для нулевого указателя.

Представление рассмотренных отображений A и T изначально недостаточно эффективно для дальнейшего анализа. Чтобы сделать его значительно более эффективным, в реализации инструмента Jessie применяется ряд важных оптимизаций на этапе преобразования специфицированной программы на промежуточном языке в программу на языке Why3ML [20].

Рассмотрим две наиболее важные из этих оптимизаций: локальное кодирование блоков памяти, имеющее цель повысить эффективность кодирования отображения A, и частично аксиоматическое кодирование отображения T для структур.

Во-первых, заметим что мы можем рассматривать значения отображения A только для блоков, доступных в данной точке программы по какому-либо указателю, так как значения для остальных блоков не могут влиять на ход вычисления. В таком случае можно заменить отображение $\mathrm{A} : \mathbb{L} \to \mathbb{Z}$ отображением $\mathrm{A}' : \mathbb{P} \to \mathbb{Z}$, таким, что $\forall (l, o) \in \mathbb{P}.\, \mathrm{A}'[(l, o)] = \mathrm{A}[l]$. Теперь можно заметить, что все три отображения A, T и **M** определены на множестве $\mathbb{P}$. Это позволяет скрыть внутреннюю структуру указателей как пар вида (l, o) и перейти к абстрактному типу указателя, введя функцию $\mathrm{o} : \mathbb{P} \to \mathbb{Z} : \forall (l, o) \in \mathrm{P}.\, \mathrm{o}((l, o)) = o$. Тогда, заменив для любого указателя p функцию $o(p)$ и отображение $A'[p]$ двумя функциями $\mathrm{omin}(\mathcal{A}, p) = -\mathrm{o}(p)$ и $\mathrm{omax}(\mathcal{A}, p) = \mathrm{A}'[p] - \mathrm{o}[p] - 1$, получим локальное кодирование блоков памяти. Кодирование называется локальным, потому что при анализе функций, которые работают с указателями с помощью операций **shift**, **psub**, **acc** и оператора **upd**, эффективнее работать с неравенствами относительно минимального и максимального смещений указателей, чем с аналогичными неравенствами относительно смещений указателей и длин соответствующих блоков памяти (например, $\mathrm{omax}(\mathcal{A}, p) \geq 0$ вместо $\mathrm{A}[p] - \mathrm{o}(p) - 1 \geq 0$). Такое кодирование было ранее применено в инструменте дедуктивной верификации Caduceus [22, 23]. В Jessie функции omin и omax кодируются в Why3ML неявно (аксиоматически) как неинтерпретируемые функции с двумя аргументами и соответствующим набором аксиом.

Во-вторых, заметим, что поскольку в промежуточном языке разрешаются только различаемые объединения, которые являются частным случаем опосредованных объединений[4], указатели на структуры никогда не бывают синонимичны указателям на объединения. Но для указателей на структуры всегда выполнен следующий инвариант: $\forall p \in \mathbb{P}. \forall i \in \mathbb{Z}.\, \mathrm{T}[p] = \mathrm{T}[\mathrm{shift}(p, i)]$ (shift — функций сдвига для абстрактного типа указателей). Отсюда возникает идея замены отображения T на аксиоматически задан-

[4] *Опосредованными* (англ. *moderated*) называют объединение, для которого отсутствует взятие адресов от полей и приведение типа указателя на объединение к указателю на любое его поле, или, иначе, обращение к полям которого всегда опосредуется обращением к самому объединению.

ную функцию $\mathrm{tag}(\mathcal{T}, p) \equiv \mathrm{T}[p]$, используемую для указателей на структуры и отображение T_u, совпадающее с исходным отображением T для объединений. Функция $\mathrm{tag}(\mathcal{T}, p)$ в таком случае кодируется неинтерпретируемой функцией двух аргументов с несколькими соответствующими аксиомами. Такое кодирование существенно упрощает предусловия функций, в которых вместо требования необходимого значения метки для каждого элемента диапазона указателей в массив структур, с которым работает данная функция, становится достаточным требования необходимого значения метки только для одного произвольного указателя в этот массив структур.

3. РАСШИРЕНИЕ УПРОЩЕННОГО ПРОМЕЖУТОЧНОГО ЯЗЫКА

3.1. *Мотивировка и абстрактный синтаксис*

Как было уже упомянуто во введении, многие реальные программы на языке Си, которые выполняют низкоуровневые преобразования над некоторыми областями памяти, не имеют прямого выражения на УПЯ, представленном в предыдущем разделе. Типичные примеры таких преобразований — переинтерпретация структуры или массива структур как массива байт перед передачей его по сети или изменение порядка байт в целочисленном значении типа `int` после приема его по интерфейсу USB или чтения из файла.

Для поддержки подобных случаев в инструменте верификации предлагается расширение УПЯ двумя новыми возможностями — операторами *переинтерпретации* (англ. *reinterpretation)* и *разбиения* (англ. *ripping*) выделенного блока памяти.

Неформально говоря, смысл первого оператора состоит в преобразовании блока памяти, выделенного для одного типа структур или объединений в блок памяти для структур или объединений другого типа, при условиях, что (1) в обоих этих структурах/объединениях отсутствуют поля-указатели и что-либо; (2а) размер первого (исходного) композитного типа кратен размеру второго (целевого) — будем называть этот случай *расщеплением* (англ. *splitting*) внутри блока памяти, либо (2б) помимо обратной кратности (размер целевого типа кратен размеру исходного), размер исходного блока также кратен размеру целевого типа — этот случай назовем *слиянием* (англ. *joining*) внутри блока.

Одного лишь оператора переинтерпретации оказывается недостаточно для представления, к примеру, переинтерпретации массива байт, содержащего две последовательно записанные структуры разных взаимно не кратных размеров, в два соответствующих указателя на структуры (в силу некратности размера массива размеру каждой из структур).

Здесь оказывается уместным применение второго оператора — разбиения, неформальный смысл которого состоит в разбиении исходного блока памяти непосредственно перед выполнением оператора переинтерпретации на две непрерывные части — доступную и теневую (англ. ghost), с целью последующего объединения этих частей вновь в единый блок (склейки, англ. *mending*) после того, как будет выполнен обратный оператор переинтерпретации и типы частей разбитого блока вновь совпадут.

Сущность второго оператора — разбиения — подсказывается возможностью языка спецификаций ACSL добавлять в программу теневые переменные-указатели, которые могут адресовать (указывать на) скрытые блоки памяти, недоступные из исходной Си-программы, однако неотличимые от обычных блоков памяти после перевода этой программы в промежуточный язык. Для оператора было выбрано название “разбиение”, которое следует отличать от “расщепления” — одного из двух вариантов переинтерпретации, аналогично “слияние” следует отличать от “склейки”. Смысл оператора разбиения состоит в отделении временно избыточной части выделенного блока памяти в скрытую область, доступную через некоторую теневую переменную, с целью последующего присоединения отделенной части назад к видимой из исходной программы части блока, и, как следствие, восстановления исходного размера этого блока.

Расширение абстрактного синтаксиса УПЯ приведено на рис. 5. В расширенном синтаксисе введена новая операция — приведение типа указателя (**cast**), использование которой, однако, не обязательно для чисто синтаксического размещения указателя на один композитный

$$
\begin{array}{rll}
t_p ::= & \ldots & \\
 & | \ \mathbf{cast}(t_p, t, t) & \textit{приведение типа} \\
 & {}^{*}| \ \mathbf{rip}(t_p, t_p) & \textit{разбиение} \\
 & & \\
o ::= & \ldots & \\
 & {}^{*}| \ \mathbf{rmem}(t_p, t, t) & \textit{переинтерпретация} \\
 & {}^{*}| \ \mathbf{mend}(t_p, t_p) & \textit{склейка}
\end{array}
$$

Рис. 5. Расширение абстрактного синтаксиса УПЯ.

тип в выражении, семантически требующем указатель на какой-либо другой. Соответствующее ограничение не может быть наложено в РУПЯ в силу его нетипизированности, но это ограничение накладывается как в настоящем, типизированном языке Jessie, так и в языке Си. Использование операции приведения типа указателя обязательно для того, чтобы оказались выполненными предусловия последующих операций **acc**, **psub** или **shift**, либо операторов **upd** или **free**. При гипотетическом (в силу ограничений, накладываемых на уровне языка Си) отсутствии операции приведения типа, а также при (действительно возможном) отсутствии оператора переинтерпретации **rmem**, программа на промежуточном языке завершается аварийно при выполнении любой из этих последующих операций (или операторов), а для соответствующего УВ (предусловия операции) не проходит проверка решателем.

Операция **rip** и операторы **mend** и **rmem** соответствуют описанным операторам разбиения/склейки и расщепления/слияния. Операция **rip** принимает два параметра — указатель во временно уменьшаемый блок памяти, который затем будет использован в исходной программе, и строго превосходящий его указатель в тот же самый блок памяти, имеющий смещение, с которого будет начинаться скрытая (временно избыточная и мешающая выполнению переинтерпретации) часть разбиваемого блока. Результатом выполнения операции является (теневой) указатель на начало нового (отсоединенного) блока памяти, представляющего временно избыточную часть исходного блока. Предполагается, что этот указатель будет сохранен в теневой переменной для последующего использования в операторе склейки (**mend**). Промежуточный язык, однако, позволяет использовать этот указатель и как-либо иначе, однако такие варианты использования не позволяют выразить ни язык спецификаций ACSL, ни язык Си, на которых записывается исходная специфицированная программа. Таким образом, семантика промежуточного языка оказывается для некоторых случаев шире семантики исходного. Первый аргумент операции **rip** — указатель в исходный блок памяти остается в результате выполнения операции неизменным, меняется лишь длина соответствующего блока памяти. Этот указатель может быть использован в последующем операторе переинтерпретации (**rmem**), который аналогичен операции **cast**, за исключением того, что он изменяет атрибуты соответствующего блока памяти и указателя (отображения A и T), а также память (отображение **M**) в качестве побочного эффекта, оставляя сам указатель (значение абстрактного типа) без изменений. После выполнения переинтерпретации указатель, полученный в результате выполнения операции приведения типа **cast**, становится действительным указателем внутрь свежего блока памяти соответствующего типа (с соответствующими метками в отображении T) и может быть использован как обычный указатель. Исходный блок памяти (с исходными метками в отображении T) при этом становится временно или окончательно недействительным. Последующая обратная операция **rmem** может быть использована для восстановления действительности исходного блока памяти с одновременным освобождением переинтерпретированного блока, и соответствующим обновлением памяти (**M**) и отображений A и T. Наконец, оператор склейки **mend** может быть использован для восстановления исходного размера изначального блока в результате соединения его видимой (и, возможно, измененной) и скрытой (теневой) частей.

Таким образом, РУПЯ по сравнению с УПЯ (рис. 3) расширен двумя дополнительными операциями — приведения типа указателя (**cast**) и разбиения блока памяти (**rip**), и двумя операторами, — переинтерпретации блока памяти

(**rmem**) и склейки прежде разбитого блока памяти (**mend**). Операция **cast** выражает некоторые не префиксные (переинтерпретирующие) приведения типов указателей, а оператор **rmem** служит для выражения обоих случаев переинтерпретации блока памяти — слияния и расщепления (переинтерпретации между типами одинакового размера можно относить к обоим случаям).

3.2. Расширение модели памяти

Для уточнения и формализации семантики операций **cast** и **rip** и операторов **rmem** и **mend** рассмотрим соответствующим образом расширенную модель памяти. Расширим модель памяти УПЯ, представленную ранее, модельными функциями и предикатами, позволяющими выразить введенные ограничения на работу с новыми конструкциями из расширенного языка.

Для поддержки переинтерпретации памяти введем новую функцию $\varphi : \mathbb{T} \times \mathbb{T} \to \mathbb{Z}$. Эта функция определена на множестве всевозможных упорядоченных пар меток композитных типов и полей объединений следующим образом:

- если метки t_1 и t_2 таковы, что: (1) размер s_{t_1} первого композитного типа или поля объединения с меткой t_1 кратен размеру второго (s_{t_2} с меткой t_2); (2) каждая из меток t_1 и t_2 соответствует либо полю объединения, либо композитному типу без полей-указателей (этому условию не удовлетворяет, в частности, метка $\top$); и (3) определен некоторый логический предикат $rmem_{t_1,t_2}(\mathbf{M}, \mathrm{T}, l, l', m)$, выражающий высокоуровневую семантику побайтового равенства между значениями полей структуры или одного поля объединения с меткой t_1 и полями структуры или полем объединения с меткой t_2 (если t_1 или t_2 являются меткой объединения, можно считать предикат $rmem_{t_1,t_2}$ тождественно истинным, так как соответствующая переинтерпретация не позволит в дальнейшем считать переинтерпретированное значение ни из какого поля соответствующего объединения), то $\varphi(t_1, t_2) = s_{t_1} \div s_{t_2}$;
- если вместо выполнения первого условия размер второго композитного типа или поля объединения (с меткой t_2) кратен размеру первого (с меткой t_1), а два остальных условия остаются выполненными, то $\varphi(t_1, t_2) = -(s_{t_2} \div s_{t_1})$;
- $\varphi(t_1, t_2) = 0$, иначе.

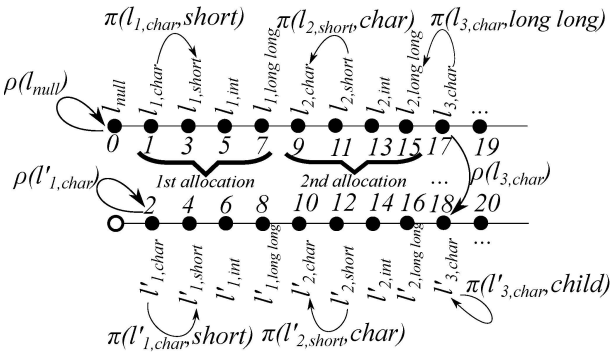

Рис. 6. Уникальные метки блоков памяти в расширенной блочно-байтовой модели памяти.

Предикат $rmem_{t_1,t_2}$ можно определять лишь по необходимости, например, для переинтерпретации в/из фиктивных структур с меткой `char`, и оставлять неопределенным (тем самым запрещая соответствующие переинтерпретации) для более сложных случаев, таких как переинтерпретации памяти между структурами со сложным выравниванием полей.

Следующую функцию $\pi : \mathbb{L} \times \mathbb{T} \to \mathbb{L}$ определим на множестве всевозможных пар уникальных меток блоков памяти и меток композитных типов или полей объединений, как показано на рисунке 6.

Здесь используется взаимно однозначное соответствие между множествами $\mathbb{L}$ и $\mathbb{N} \cup \{0\}$. Уникальные метки блоков памяти теперь индексированы метками из множества $\mathbb{T}$. При выполнении операции выделения памяти (**alloc**) метка-индекс из множества $\mathbb{T}$ у метки нового блока памяти из множества $\mathbb{L}$ должна совпадать с явно указанной в параметре операции **alloc** меткой выделяемой структуры или объединения. Таким образом при выделении памяти уникальная метка блока l_{it} (или $l_{i,t}$), индексированная меткой типа t, может быть поставлена в соответствие только блоку памяти, содержащему массив структур или объединений с этой же самой меткой типа t (единственную структуру или объединение можно рассматривать как частный случай массива). Метки блоков памяти также сгруппированы таким образом, что каждой метке $l_{i,t}$ (и

только ей) соответствует некоторая единственная прилежащая справа к ней теневая метка $l'_{i,t}$ и набор меток $\{l_{i,t} \mid t \in \mathbb{T}\}$ для переинтерпретации, где для каждой пары меток выполнено неравенство $l_{i,t} \neq l_{j,t'}, i \neq j \wedge t \neq t'$. Функция π ставит в соответствие метке $l_{i,t}$ соответствующую ей в ее группе метку для переинтерпретации $l_{i,t'}$, так что $\pi(l_{i,t}, t') = l_{i,t'}$. Среди меток для переинтерпретации исходная метка выделяется лишь в предусловии операции **alloc**, во всех остальных случаях блоки памяти, изначально выделяемые с помощью операции **alloc** и блоки, полученные с помощью оператора переинтерпретации **rmem** из других блоков, не различимы.

Последняя новая функция $\rho : \mathbb{L} \to \mathbb{L}$ ставит в соответствие метке $l_{i,t}$ соответствующую ей прилежащую справа метку $l'_{i,t}$ и используется для определения операции **rip** и оператора **mend**.

В целом с помощью трех введенных функций φ, π и ρ можно формализовать семантику операций **cast** и **rip** и операторов **rmem** и **mend**.

Эта семантика должна, в частности, поддерживать инвариант, состоящий в том, что в точности та память, которая доступна через действительные указатели в исходной программе доступна и в соответствующей ей программе на промежуточном языке. Операция **rmem** помимо вычислительной семантики дополнительно имеет также и неявную аналитическую семантику. Она выражается постусловием $rmem_{t_1,t_2}(\mathbf{M}, \mathrm{T}, l, l', A[l])$, которое трактуется как оператор предположения (**assume**), выполняемый непосредственно после оператора **rmem**.

Таким образом, расширение УПЯ операторами переинтерпретации и разбиения позволяет расширить область его применимости для представления исходных программ на языке Си, использующих некоторые низкоуровневые приведения типов указателей. Хотя семантика РУПЯ с исходным представлением отображения A с помощью массива позволяет выполнять разбиение блока памяти только с отделением его старшей части (с большими значениями адресов), в настоящем языке Jessie, реализующем локальное кодирование блоков памяти, возможно отделение как старшей, так и младшей части выделенного блока памяти. При использовании локального кодирования блоков памяти семантика промежуточного языка становится достаточно выразительной для представления с помощью этого языка большинства практически используемых случаев низкоуровневого приведения типов указателей, таких как операции кодирования/декодирования, буферизации и переупорядочения байт, часто используемые в драйверах файловых систем и сетевых устройств.

4. ВЫВОД И НАПРАВЛЕНИЯ ДАЛЬНЕЙШЕЙ РАБОТЫ

В статье представлен упрощенный промежуточный язык — аналог языка Jessie, одновременно поддерживающий различаемые объединения и префиксные приведения типов указателей на структуры и совместимый по своей семантике с семантикой соответствующего подмножества языка Си. Представленный язык был использован одновременно в целях сжатого изложения концепций, лежащих в основе инструмента дедуктивной верификации Jessie (и соответствующего промежуточного языка [8, 24]), и в качестве отправной точки для дальнейшего расширения. В статье было представлено расширение представленного языка, позволяющее выражать семантику некоторых низкоуровневых приведений типа указателей на структуры или объединения, не содержащие полей-указателей.

С практической точки зрения представленные расширения позволяют применять расширенный язык для анализа более широкого класса программ на языке Си, в частности, модулей ядра ОС Linux. В этой связи необходимо еще выполнить значительную работу по доведению текущей (измененной) реализации инструмента Jessie до приемлемого уровня качества (в реализации, в частности, есть известные ошибки и ограничения, связанные с анализом регионов и вычислением эффектов [19, 11, 9]), а также по дедуктивной верификации некоторого значительного объема кода модулей ядра ОС Linux, прежде чем можно будет представить какие-либо значимые практические результаты. На момент написания статьи в реализации уже присутствует поддержка оператора переинтерпретации, но не оператора разбиения, так что верификации уже поддаются, например, участки кода, осуществляющие переупорядочение байт, но все

еще не поддаются некоторые случаи реализации упаковки/распаковки и буферизации.

С теоретической же точки зрения остается много важных направлений дальнейшей работы, в частности, строгая формализация семантики промежуточного языка, проверка соответствия его семантики низкоуровневой семантике языка Си, а также проверка корректности генерируемого инструментом верификации аксиоматического представления программы.

СПИСОК ЛИТЕРАТУРЫ

1. *Gomes C.P., Kautz H., Sabharwal A., Selman B.* Satisfability solvers. 2008.
2. *Kroening D., Strihman O.* Deision Proedures: An Algorithmic Point of View. 1 edition. Springer Publishing Company, Incorporated, 2008.
3. *Reynolds J.C.* Separation logic: A logic for shared mutable data structures // Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. LICS'02. Washington, DC, USA: IEEE Computer Society, 2002. P. 55–74. http://dl.acm.org/citation.cfm?id=645683.664578.
4. Extended static checking for java / Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge et al. // Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. PLDI'02. New York, NY, USA: ACM, 2002. P. 234–245. http://doi.acm.org/10.1145/512529.512558.
5. *Barnett M., Leino K. Rustan M., Schulte W.* The spec# programming system: An overview // Proceedings of the 2004 International Conference on Concstruction and Analysis of Safe, Secure, and Interoperable Smart Devices. CASSIS'04. Berlin, Heidelberg: Springer-Verlag, 2005. P. 49–69. http://dx.doi.org/10.1007/978-3-540-30569-9_3.
6. Vcc: A practical system for verifying concurrent c / Ernie Cohen, Markus Dahlweid, Mark Hillebrand et al. // Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics. TPHOLs'09. Berlin, Heidelberg: Springer-Verlag, 2009. P. 23–42. http://dx.doi.org/10.1007/978-3-642-03359-9_2.
7. Frama-c: A software analysis perspective / Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov et al. // Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12. Berlin, Heidelberg: Springer-Verlag, 2012. P. 233–247. http://dx.doi.org/10.1007/978-3-642-33826-7_16.
8. *Moy Y.* Automatic Modular Staticx Safety Checking for C Programs: Ph.D. thesis / Université Paris-Sud. 2009. January. http://www.lri.fr/ marche/moy09phd.pdf.
9. *Talpin J.-P., Jouvelot P.* Polymorphic type, region and effect inference // Journal of Functional Programming. 1992. V. 2. P. 245–271.
10. *Tofte M., Talpin J.-P.* Region-based memory management // Information and Computation. 1997. V. 132. № 2. P. 109–176. http://www.sciencediret.com/sciene/article/pii/ /S0890540196926139.
11. *Hubert T., Marché C.* Separation analysis for deductive verification // Heap Analysis and Verification (HAV'07). Braga, Portugal: 2007. March. P. 81–93. http://www.lri.fr/ marche/hubert07hav.pdf.
12. *Moy Y., Marché C.* Modular inference of subprogram contracts for safety checking // Journal of Symbolic Computation. 2010. V. 45. P. 1184–1211.
13. *Burstall R.M.* Some techniques for proving correctness of programs which alter data structures // Machine intelligence. 1972. V. 7. № 23–50. P. 3.
14. *Bornat R.* Proving pointer programs in hoare logic // Proceedings of the 5th International Conference on Mathematics of Program Construction. MPC'00. London, UK, UK: Springer-Verlag, 2000. P. 102–126. http://dl.acm.org/citation.cfm?id=648085.747307.
15. *Moy Y.* Union and cast in deductive verification // Proceedings of the C/C++ Verification Workshop. V. Technical Report ICIS-R07015.Radboud University Nijmegen, 2007. July. P. 1–16. http://www.lri.fr/∼moy/Publis/moy07ccpp.pdf.
16. *Marché C.* Jessie: An intermediate language for java and verification // Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification. PLPV'07.New York, NY, USA: ACM, 2007. P. 1–2. http://doi.acm.org/10.1145/1292597.1292598.
17. Ccured in the real world / Jeremy Condit, Matthew Harren, Scott McPeak et al. // Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. PLDI'03. New York, NY, USA: ACM, 2003. P. 232–244. http://doi.acm.org/10.1145/781131.781157.
18. A precise yet efficient memory model for c / Ernie Cohen, Michal Moskal, Stephan Tobies, Wolfram Schulte // Electron. Notes Theor. Comput. Sci.

2009. October. V. 254. P. 85–103. http://dx.doi.org/10.1016/j.entcs.2009.09.061.

19. *Steensgaard B.* Points-to analysis in almost linear time // Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'96. New York, NY, USA: ACM, 1996. P. 32–41. http://doi.acm.org/10.1145/237721.237727.

20. *Filliâtre J.-C., Paskevich A.* Why3: Where programs meet provers // Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13. Berlin, Heidelberg: Springer-Verlag, 2013. P. 125–128. http://dx.doi.org/10.1007/978-3-642-37036-6_8.

21. *Dijkstra E.W.* Guarded commands, nondeterminacy, and formal derivation of programs // Communications of the ACM. 1975. August. V. 18. № 8. P. 453–457.

22. *Filliâtre J.-C., Marché C.* The why/krakatoa/ /caduceus platform for deductive program verification // In CAV'07. 2007. P. 173–177.

23. *Filliâtre J.-C., Marhé C.* Multi-prover verification of C programs // 6th International Conference on Formal Engineering Methods / Ed. by Jim Davies, Wolfram Schulte, Mike Barnett. V. 3308 of Lecture Notes in Computer Science. Seattle, WA, USA: Springer, 2004. November. P. 15–29. http://www.lri.fr/ filliatr/ftp/publis/caduceus.ps.gz.

24. Архитектура Linux driver verification / В.С. Мутилин, Е.М. Новиков, А.В. Страх и др. // Труды Института системного программирования АН. 2011. Т. 20. С. 163–187.