Теорема 2. Пусть для якобиана коммутативной системы (2) выполнено равенство

$$J(z, x) \equiv 0,$$

тогда некоммутативная система уравнений (1) либо не имеет решений (в виде ФСР $z = z(x)$), либо имеет любое конечное число решений, либо бесконечно много решений.

Суть теоремы 2 состоит в том, что равенство нулю якобиана не ограничивает свойств некоммутативной системы уравнений.

Учитывая, что система

$$f = 0, \ \ldots, \ f = 0,$$

из n одинаковых уравнений с n некоммутативными неизвестными $z_1, \ldots, z_n$, имеющая тождественно равный нулю якобиан, эквивалентна одному уравнению $f = 0$, сформулируем следствие: одно уравнение с некоммутативными неизвестными $P_1(z, x) = 0$ может не иметь решений, а также иметь конечное и бесконечное число решений.

В этом состоит фундаментальное отличие от одного уравнения над полем комплексных чисел, которое всегда имеет решения в виде аналитических функций.

ЛИТЕРАТУРА

1. *Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л.* Алгебра. Языки. Программирование. Киев: Наукова думка, 1973.
2. *Salomaa A. and Soitolla M.* Automata-Theoretic Aspects of Formal Power Series. N.Y.: Springer Verlag, 1978.
3. *Егорушкин О. И., Колбасина И. В., Сафонов К. В.* О совместности систем символьных полиномиальных уравнений и их приложении // Прикладная дискретная математика. Приложение. 2016. № 9. С. 119–121.
4. *Egorushkin O. I., Kolbasina I. V., and Safonov K. V.* On solvability of systems of symbolic polynomial equations // Журн. СФУ. Сер. Матем. и физ. 2016. Т. 9. Вып. 2. С. 166–172.
5. *Семёнов А. Л.* Алгоритмические проблемы для степенных рядов и контекстно-свободных грамматик // Доклады АН СССР. 1973. № 212. С. 50–52.

УДК 004.056.5, 004.94 DOI 10.17223/2226308X/15/21

ПРИЕМЫ ДЕДУКТИВНОЙ ВЕРИФИКАЦИИ ПРОГРАММНОГО КОДА С ИСПОЛЬЗОВАНИЕМ AstraVer Toolset

А. О. Кокорин, С. Д. Тиевский, П. Н. Девянин

Описывается ряд практических приёмов дедуктивной верификации программного кода на языке Си на соответствие спецификациям его функций, заданных на языке ACSL. Для такой верификации используется основанный на платформе Frama-C набор инструментов AstraVer Toolset. Апробация этих приёмов осуществлена при верификации программного кода модуля управления доступом, реализованного в подсистеме безопасности PARSEC отечественной защищённой операционной системы специального назначения Astra Linux Special Edition. Благодаря использованию этих приёмов удалось упростить спецификации функций PARSEC, уменьшить трудоёмкость и ускорить процесс их дедуктивной верификации.

Ключевые слова: *дедуктивная верификация программного кода, ACSL, Frama-C, AstraVer Toolset, Astra Linux.*

Введение

При разработке отечественной защищённой сертифицированной по высшим классам защиты (уровням доверия) операционной системы специального назначения (ОССН) Astra Linux Special Edition [1, 2] применяется широкий спектр технологий верификации, статического и динамического анализа её программного кода. При этом особое внимание уделяется коду подсистемы безопасности PARSEC, которая в ОССН реализует механизм управления доступом и строится на основе мандатной сущностно-ролевой ДП-модели управления доступом и информационными потоками в ОС семейства Linux (МРОСЛ ДП-модели) [3]. В этой связи, во-первых, верифицируется сама модель, для чего её описание переводится с математического языка на машиночитаемый язык формального метода Event-B [4, 5]. После этого осуществляется дедуктивная верификация модели с применением инструмента Rodin и её верификация по методу проверки моделей инструментом ProB [6]. Во-вторых, наличие развитой модели, достаточно детально описывающей на математическом и формализованном языках механизм управления доступом ОССН, создаёт условия для верификации реализации модели непосредственно в программном коде подсистемы безопасности PARSEC. В результате для ОССН обеспечивается выполнение требования «Методики выявления уязвимостей и недекларированных возможностей в программном обеспечении» [7] в части разработки формальных (математических) описаний модулей, реализующих функции безопасности, и верификации их согласованности с моделью.

В настоящей работе рассмотрена только дедуктивная верификация, которая позволяет получить гарантию корректности исходного кода относительно его спецификаций, т. е. дедуктивная верификация заключается в разработке спецификаций функций (в первую очередь их контрактов, включающих, как минимум, предусловия и постусловия функций) программного кода PARSEC на языке ANSI/ISO C Specification Language (ACSL) [8] и доказательстве с применением набора инструментов AstraVer Toolset [5, 9] того, что в предположении о получении на вход каждой функции допустимых данных (предусловие выполнено) она возвращает соответствующий результат (постусловие выполнено). При этом также доказывается, что функции завершают работу корректно, т. е. не «зацикливаются» и не содержат ошибок вида неопределённого поведения.

Результатом является описание нескольких приёмов дедуктивной верификации программного кода, разработанных (либо развитых и адаптированных на основе существующих) и апробированных авторами.

1. Порядок верификации функций

Для дедуктивной верификации программного кода подсистемы безопасности PARSEC используется набор инструментов AstraVer Toolset. Он основан на открытой платформе верификации Frama-C [5, 10] и инструменте дедуктивной верификации Why3 [11]. В его состав входит доработанный плагин Jessie [12], задачей которого является преобразование кода на языке Си и спецификаций на языке ACSL в множество утверждений для доказательства (англ. *Proof Obligation — PO*). Эти утверждения получает инструмент Why3, который далее передаёт их на проверку выполнимости с помощью различных инструментов — автоматических, таких, как SMT-решатели CVC4, Alt-Ergo, Z3 и др., или интерактивных, таких, как инструмент Coq.

Спецификации на языке ACSL разрабатываются вручную экспертом и представляют собой набор формальных утверждений относительно поведения функций программного кода. Эти спецификации состоят из предусловий (условий нормального вы-

полнения функции) и постусловий, истинность которых гарантируется после выполнения функции. Они записываются перед каждой специфицируемой функцией в комментарии, начинающемся с символов /*@. Предусловия задаются с помощью ключевого слова requires, постусловия — с помощью ключевого слова ensures.

Пример 1. Пусть необходимо задать спецификации функции pdpl_get подсистемы безопасности PARSEC, предназначенной для увеличения счётчика использований метки безопасности в поле ucnt структуры типа PDPL_T, на которую указывает входной параметр l.

```
/*@ requires \valid(l);
requires l->ucnt.counter < INT_MAX;
assigns l->ucnt.counter;
ensures \result == l;
*/

const PDPL_T* pdpl_get(const PDPL_T *l) {
PDPL_T *ncl = (PDPL_T*)l;
atomic_inc(&(ncl->ucnt));
return l;
}
```

Для этого в предусловия функции, во-первых, включено требование корректности указателя l, которое определяется с помощью ключевого слова \valid. Во-вторых, в него добавлено ограничение на значение счётчика, которое должно быть меньше INT_MAX, что гарантирует отсутствие целочисленного переполнения (константы INT_MAX и INT_MIN, определённые в стандартной библиотеке языка Си в заголовочном файле limits.h, задают максимальное и минимальное значения переменных целого типа int соответственно). В-третьих, в предусловии указан разрешённый для изменения в функции участок памяти. Это осуществляется с помощью рамочного условия assigns, которое разрешает изменение только переменной l–>ucnt.counter (при этом в условии assigns не требуется указывать изменяющиеся локальные переменные). В завершении спецификаций в постусловие с помощью ключевого слова \result задано возвращаемое значение функции.

2. Использование глобальных переменных для отслеживания вызовов функций захвата/освобождения разделяемой памяти

В программах на языке Си необходимо обеспечить корректное управление разделяемой памятью [13] на всех путях исполнения. При верификации кода необходимо задавать спецификации функций, захватывающих и освобождающих такую память. Однако некоторые функции захвата/освобождения разделяемой памяти в ядре ОС семейства Linux определены с использованием сложных конструкций из макросов, что затрудняет написание спецификаций в полном соответствии с реализацией функции. Это также усложняется тем, что в программном коде применяются операторы условного перехода «if then else», а иногда безусловного перехода «goto», которые при верификации затрудняют отслеживание операций захвата/освобождения разделяемой памяти на всех путях выполнения кода. Кроме того, модель памяти, используемая набором инструментов AstraVer Toolset [9], не поддерживает верификацию параллельно исполняемого программного кода.

Для верификации функций захвата/освобождения разделяемой памяти предлагается следующий приём: формировать спецификации, только моделирующие поведение этих функций, для чего использовать «теневой» код на языке ACSL [8]. Переменные «теневого» кода задаются с помощью ключевого слова ghost. Такой код не обрабатывается компилятором Си, а предназначен для использования только инструментами верификации, которые работают с ним как с программным кодом. Это удобно во многих случаях для описания в спецификациях сложных свойств поведения функций, в том числе работающих с разделяемой памятью.

Пример 2. Пусть необходимо отслеживание доступов к критической секции для обращения на чтение к памяти в ядре ОС семейства Linux, осуществляемого с помощью функций rcu_read_lock (вход в критическую секцию для чтения памяти) и rcu_read_unlock (выход из критической секции). В этом случае необходимо обеспечить равенство числа вызовов rcu_read_lock и rcu_read_unlock.

```
// «Теневая» глобальная переменная - счетчик обращений к памяти
//@ ghost int ghost_rcu_lock;

// Предусловие для предотвращения целочисленного переполнения счетчика
/*@ requires ghost_rcu_lock < INT_MAX;
assigns ghost_rcu_lock;
// Увеличение счетчика ghost_rcu_lock на единицу
ensures ghost_rcu_lock == \old(ghost_rcu_lock) + 1;
*/
static inline void rcu_read_lock(void);

// Предусловие для предотвращения целочисленного переполнения счетчика
/*@ requires ghost_rcu_lock > INT_MIN;
assigns ghost_rcu_lock;
// Уменьшение счетчика ghost_rcu_lock на единицу
ensures ghost_rcu_lock == \old(ghost_rcu_lock) - 1;
*/
static inline void rcu_read_unlock(void);
```

Предположим, что в некоторой функции f вызываются функции захвата/освобождения критической секции и в спецификациях этой функции указано assigns ghost_rcu_lock, т. е. внешние по отношению к функции области памяти не должны изменяться. При этом в коде функции f при использовании критической секции допущена ошибка.

```
/*@ requires ghost_rcu_lock < INT_MAX;
assigns ghost_rcu_lock;
ensures ghost_rcu_lock == \old(ghost_rcu_lock);
*/
void f(int a) {
rcu_read_lock();
if (a == 0) {
goto exit;
}
```

```
rcu_read_unlock();
exit:
}
```

В функции f обеспечен вход в критическую секцию с помощью функции rcu_read_lock, но если выполняется условие «a == 0», то выход из критической секции не осуществляется. Для этой функции утверждение «ensures ghost_rcu_lock == \old(ghost_rcu_lock)» не пройдёт верификацию, так как оно требует неизменность счётчика ghost_rcu_lock, что позволяет контролировать равенство числа входов и выходов из критической секции. Недостаток предложенного приёма в примере 2 заключается в том, что в каждой функции, вызывающей rcu_read_lock, необходимо указывать условие ghost_rcu_lock $<$ INT_MAX. Без него верификация не будет завершаться, так как функция rcu_read_lock временно увеличивает значение ghost_rcu_lock, что без проверки указанного условия может привести к целочисленному переполнению.

Таким образом, приём, основанный на применении «теневого» кода на языке ACSL, позволяет моделировать поведение функций захвата/освобождения разделяемой памяти, что снижает риск появления ошибок в использующем их программном коде. При этом он не предполагает задания спецификаций таких функций, в точности им соответствующих. В итоге достигается компромисс между сложностью верификации и её результативностью.

3. Проверка корректности спецификаций вызываемых функций

При дедуктивной верификации как программного кода с применением спецификаций функций на языке ACSL, так и моделей на других формализованных языках, например метода Event-B [5, 6], существует опасность доказательства произвольных утверждений из некоторого ошибочного ложного условия или совокупности ложных условий. Такие условия могут не иметь явных признаков ошибки и при верификации могут рассматриваться экспертом как истинные.

В примере 2 счётчик числа входов и выходов из критической секции ghost_rcu_lock имеет целочисленный тип int (целое число в диапазоне между INT_MIN и INT_MAX), а не integer (произвольное целое число), так как «теневые» переменные (ghost) на языке ACSL могут иметь только типы, используемые в языке Си. Поэтому в спецификациях необходимо указывать в качестве предусловий функций rcu_read_lock и rcu_read_unlock ограничения для предотвращения переполнения этого счётчика, иначе в вызывающих функциях появится возможность доказывать произвольные утверждения. Поэтому необходимо контролировать корректность спецификаций тех функций, которые полностью не верифицируются, а только моделируются.

Для этого предлагается использовать следующий приём: добавлять заведомо ложное постусловие в спецификациях вызывающих функций, например «ensures 1 == 0». Если при этом верификация будет успешно завершаться, то в спецификациях допущена ошибка. Рассмотрим пример.

Пример 3. Пусть, аналогично примеру 2, необходимо отслеживание доступов к критической секции для обращения на чтение к памяти в ядре ОС семейства Linux. При этом в спецификациях функции входа в критическую секцию rcu_read_lock пропущено условие assigns ghost_rcu_lock.

```
// «Теневая» глобальная переменная - счетчик обращений к памяти
//@ ghost int ghost_rcu_lock;
```

```
/*@
// Увеличение счетчика ghost_rcu_lock на единицу
ensures ghost_rcu_lock == \old(ghost_rcu_lock) + 1;
*/
static inline void rcu_read_lock(void);

// Заведомо ложное условие
/*@ ensures 1 == 0; */
void f() {
rcu_read_lock();
}
```

Условие «ensures $1 == 0$» в спецификациях функции f проходит доказательство, хотя очевидно, что оно ложно. Это стало возможным из-за некорректной спецификации функции rcu_read_lock. На самом деле, постусловие функции rcu_read_lock гарантирует увеличение счётчика ghost_rcu_lock, в то время как отсутствует условие assigns. По умолчанию инструменты верификации считают, что если контракт без assigns написан для функции без тела (объявления функции), функция может изменять «всё». На практике функции, которые вызывают данную функцию, невозможно верифицировать верно, что делает написание контрактов для подобных функций обязательным. Если у функции есть тело, но нет assigns, то инструменты попытаются аппроксимировать множества изменяемых мест в памяти [8]. Инструменты верификации принимают это ложное условие и, исходя из него, доказывают произвольные утверждения. После проведения верификации при заведомо ложном постусловии «ensures $1 == 0$» его необходимо удалить, так как в противном случае уже в функциях, вызывающих f, будут верифицироваться произвольные утверждения.

4. Обход ограничений инструментов верификации при сравнении значений целочисленных типов и указателей

В языке программирования Си указатели представляют собой целые числа заданного для конкретной аппаратной платформы размера. Поэтому перетипизация целого числа в указатель (и наоборот) не требует дополнительных машинных инструкций, что позволяет сравнивать переменные таких типов друг с другом. Так, часто функции ядра ОС семейства Linux в качестве возвращаемого значения могут выдавать либо указатель на переменную, либо приведённый к типу указателя целочисленный код ошибки.

Однако модель памяти, используемая в наборе инструментов AstraVer Toolset, не допускает сравнений переменных целочисленного типа и произвольных указателей. Это затрудняет верификацию функций, в которых такие сравнения производятся.

Чтобы обойти это ограничение, предлагается следующий приём: использовать сравнение указателя с \null вместо сравнения с приведённой к типу указателя целочисленной переменной. Рассмотрим пример.

Пример 4. Проанализируем функции ядра ОС семейства Linux: ERR_PTR (преобразует целочисленный код ошибки в тип указателя) и IS_ERR (проверяет, возвращается ли в указателе код ошибки). Первая функция имеет следующий программный код:

```
static inline void * __must_check ERR_PTR(long error)
{
return (void *) error;
}
```

Пусть вызывающая функцию ERR_PTR функция use_err_ptr в случае успешного выделения памяти функцией get_int в переменной *pointer* возвращает указатель на неё, в противном случае — приведённый к типу указателя код ошибки -EFAULT:

```
#define EFAULT 13

/*@ assigns \nothing;
ensures \result == \null || \valid(\result);
*/
int* get_int();

int* use_err_ptr()
{
int* pointer = get_int();
if (!pointer) {
return ERR_PTR(-EFAULT);
} else {
return pointer;
}
}
```

В наборе инструментов AstraVer Toolset целочисленный тип и тип указателя несовместимы, что не позволяет записать в спецификациях постусловие функции use_err_ptr как «ensures \valid(\result) || (\result == -EFAULT). Чтобы решить эту задачу, предлагается моделировать ERR_PTR как функцию, возвращающую нулевой указатель, задав её следующие спецификации:

```
/*@ assigns \nothing;
ensures \result == \null;
*/
static inline void *ERR_PTR(long error);
```

Сравнение \result == \null поддерживается AstraVer Toolset. При этом теряется информация о коде ошибки, однако становится возможным описание в спецификациях поведения функций, использующих ERR_PTR, в том числе функции use_err_ptr:

```
/*@ assigns \nothing;
ensures \result == \null || \valid(\result);
*/
int* use_err_ptr();
```

Тогда спецификации функции IS_ERR, проверяющей, содержит ли возвращаемый указатель код ошибки, можно описать так:

```
/*@ assigns \nothing;
ensures \result == (ptr == \null);
*/
static inline bool IS_ERR(const void *ptr);
```

Таким образом, использование в спецификациях функций сравнения указателя с \null вместо сравнения с приведённой к типу указателя целочисленной переменной не полностью соответствует поведению функций, однако этот приём позволяет верифицировать большую часть программного кода таких функций.

5. Леммы, показывающие корректность предикатов

С помощью языка ACSL [8] для описания сложных свойств поведения функций возможно определение лемм, аксиом и предикатов. Леммы задаются ключевым словом lemma. Они определяются экспертом и могут помочь средствам верификации при доказательстве утверждений. Например, лемма mean_property (о том, что среднее арифметическое двух целых чисел находится между их значениями) может являться «подсказкой» при верификации программы бинарного поиска:

```
/*@ lemma mean_property:
\forall integer x,y; x <= y ==> x <= (x+y)/2 <= y;
*/
```

Аксиомы аналогичны леммам, но принимаются средствами верификации без попытки доказательства. Они задаются с помощью ключевого слова axiom.

Предикаты являются выражениями с результатом логического типа boolean. Они задаются с помощью ключевого слова predicate. Например, предикат is_positive определяет, является ли параметр x положительным целым числом:

```
//@ predicate is_positive(integer x) = x > 0;
```

При верификации нетривиальных программ часто необходимо описывать сложные предикаты. Одним из возможных подходов здесь является задание предиката через аксиомы. Рассмотрим пример.

Пример 5. Зададим предикат чётности целого положительного числа через аксиомы Even (для чётных чисел) и NotEven (для нечётных чисел):

```
/*@ axiomatic Even {
predicate is_even(integer x);
axiom Even: is_even(0) &&
(\forall integer x; is_even(x) ==> is_even(x + 2));
axiom NotEven: !is_even(1) &&
(\forall integer x; !is_even(x) ==> !is_even(x + 2));
}*/
```

Предикат is_even определяется в блоке axiomatic Even. Ключевое слово axiomatic не означает, что всё, что находится внутри блока, является утверждениями, принимаемыми без доказательства, а лишь группирует находящиеся внутри леммы, предикаты или логические функции.

Однако при определении предиката через аксиомы, корректность которых не доказывается, возрастает риск ошибки в спецификациях. В некоторых случаях определение предиката как функции на языке Си проще, чем через аксиомы (например, для предиката is_even можно использовать выражение на языке Си «x % 2 == 0»).

Предлагается следующий приём: задавать предикат так же, как реализована функция на языке Си, а затем с помощью введения дополнительных лемм доказывать, что этот предикат обладает требуемыми свойствами. Рассмотрим пример.

Пример 6. Для задания множеств небольшой мощности (до 32 элементов) может быть использована битовая маска, которая представлена переменной типа unsigned int языка Си. Тогда функция «множество a является подмножеством множества b» на языке Си может быть определена следующим образом:

```
int is_subset(unsigned int a, unsigned int b) {
return (a & b) == a;
}
```

Функция is_subset принимает две задающие множества маски бит в виде параметров типа unsigned int. В соответствии с предложенным приёмом, для функции is_subset спецификации могут быть заданы следующим образом:

```
/*@ // Предикат "является подмножеством"
predicate isSubset(unsigned int a, unsigned int b) =
(a & b) == a; // соответствует реализации функции
// на языке Си is_subset
*/
/*@ axiomatic isSubsetProperties {
// Проверяемые свойства предиката isSubset
// Проверка для конкретных значений множеств
lemma testIsSubset_1:
isSubset(5, 13); // 5 == 0b0101; 13 == 0b1101
// Проверка для конкретных значений множеств
lemma testIsSubset_2:
!isSubset(13, 1); // 13 == 0b1101; 1 == 0b0001
// Проверка: пустое множество подмножество любого множества
lemma emptyIsAlwaysSubset: // (1)
\forall unsigned int x;
isSubset(0, x);
// Проверка: подмножество пустого множества - пустое множество
lemma onlyEmptyIsSubsetOfEmpty: // (2)
\forall unsigned int x;
isSubset(x, 0) ==> (x == 0);
// Проверка: отношение быть подмножеством рефлексивно
lemma isSubsetReflexive: // (3)
\forall unsigned int x;
isSubset(x, x);
// Проверка: отношение быть подмножеством транзитивно
lemma isSubsetTransitive: // (4)
\forall unsigned int x, y, z;
```

```
isSubset(x, y) && isSubset(y, z) ==> isSubset(x, z);
}*/
/*@ // Постусловие:  задание предиката isSubset на языке ACSL
ensures \result == (isSubset(a, b) ? 1 : 0);
*/
int is_subset(unsigned int a, unsigned int b);
```

В блоке axiomatic заданы леммы. Для верификации этих лемм достаточно использовать автоматические средства доказательства. После того, как доказано, что предикат обладает требуемыми свойствами, несущественные леммы целесообразно удалить (в примере это леммы testIsSubset_1 и testIsSubset_2), так как большое число лемм затрудняет функционирование средств автоматического доказательства.

Таким образом, предложенный приём позволяет избежать излишнего использования аксиом для определения предикатов, однако при этом необходимо вводить леммы, с помощью которых проверяется корректность задания предикатов.

Заключение

Изложены некоторые практические приёмы дедуктивной верификации программного кода на языке Си на соответствие спецификациям его функций, заданных на языке ACSL. Эти приёмы успешно апробированы при верификации программного кода подсистемы безопасности PARSEC ОССН Astra Linux Special Edition. Они во многих случаях позволяют упростить и ускорить верификацию программного кода на языке Си, увеличить число функций, которые могут быть верифицированы, и повысить уверенность в корректности написанных спецификаций. В дальнейшем предполагается расширять состав таких приёмов, адаптированных для применения при верификации программного кода системного программного обеспечения, в особенности ОССН. Всё перечисленное должно быть полезно при дедуктивной верификации программного кода других сертифицированных средств защиты информации.

ЛИТЕРАТУРА

1. `https://astralinux.ru/products/astra-linux-special-edition/` — Операционная система специального назначения Astra Linux Special Edition. 2022.
2. *Буренин П. В., Девянин П. Н., Лебеденко Е. В. и др.* Безопасность операционной системы специального назначения Astra Linux Special Edition: учеб. пособие для вузов. 3-е изд., перераб. и доп. М.: Горячая линия — Телеком, 2019. 404 с.
3. *Девянин П. Н.* Модели безопасности компьютерных систем. Управление доступом и информационными потоками: учеб. пособие для вузов. 3-е изд., перераб. и доп. М.: Горячая линия — Телеком, 2020. 352 с.
4. *Abrial J.-R., Butler M., Hallerstede S., et al.* Rodin: An open toolset for modelling and reasoning in Event-B // Intern. J. Software Tools for Technology Transfer. 2010. V. 12. No. 6. P. 447–466.
5. *Девянин П. Н., Ефремов Д. В., Кулямин В. В. и др.* Моделирование и верификация политик безопасности управления доступом в операционных системах. М.: Горячая линия — Телеком, 2019. 214 с.
6. *Девянин П. Н., Леонова М. А.* Приёмы по доработке описания модели управления доступом ОССН Astra Linux Special Edition на формализованном языке метода Event-B для обеспечения её автоматизированной верификации с применением инструментов Rodin и ProB // Прикладная дискретная математика. 2021. № 52. С. 83–96.

7. https://fstec.ru/normotvorcheskaya/informatsionnye-i-analiticheskie-materialy/2171-informatsionnoe-soobshchenie-fstek-rossii-ot-10-fevralya-2021-g-n-240-24-647. Информационное сообщение ФСТЭК России от 10.02.2021 № 240/24/647.
8. https://frama-c.com/html/acsl.html — ANSI/ISO C Specification Language. 2022.
9. https://www.ispras.ru/technologies/astraver_toolset/ — Система верификации AstraVer Toolset. 2022.
10. *Kirchner F., Kosmatov N., Prevosto V., et al.* Frama-C: a software analysis perspective // Formal Aspects of Computing. 2015. No. 27(3). P. 573–609.
11. *Filliatre J.-C. and Paskevich A.* Why3 — where programs meet provers // LNCS. 2013. V. 7792. P. 125–128.
12. *Marche C. and Moy Y.* The Jessie Plugin for Deductive Verification in Frama-C. INRIA Saclay Ile-de-France and LRI, CNRS UMR, 2012.
13. *Колисниченко Д. Н.* Разработка Linux-приложений. СПб.: БХВ-Петербург, 2012. 432 с.

УДК 004.056.5, 004.94 DOI 10.17223/2226308X/15/22

СРАВНЕНИЕ СПОСОБОВ МОДЕЛИРОВАНИЯ МЕХАНИЗМОВ УПРАВЛЕНИЯ ДОСТУПОМ ОС И СУБД НА ФОРМАЛИЗОВАННОМ ЯЗЫКЕ МЕТОДА Event-B С ЦЕЛЬЮ ИХ ВЕРИФИКАЦИИ ИНСТРУМЕНТАМИ Rodin И ProB

М. А. Леонова, П. Н. Девянин

В результате перевода описания формальной модели управления доступом промышленной ОССН Astra Linux Special Edition (МРОСЛ ДП-модели) из математической в формализованную нотацию на языке метода Event-B, её автоматизированной верификации инструментами Rodin и ProB и проведённых в ГК Astra Linux научных исследований разработаны два способа моделирования взаимодействующих между собой систем, самостоятельно реализующих развитые механизмы управления доступом, такие, как ОС и СУБД. Эти способы основаны на использовании различных вариантов построения иерархии спецификаций МРОСЛ ДП-модели в формализованной нотации с применением техники пошагового уточнения. Сравнение способов показало как их достоинства, так и недостатки в части сложности написания спецификаций, необходимости повторения доказательств при верификации инструментом Rodin, возможности устранения эффекта «комбинаторного взрыва» при верификации инструментом ProB. По результатам сравнения сделан вывод, что рассмотренные способы могут быть полезны при разработке других формальных моделей управления доступом и их верификации с применением соответствующих средств.

Ключевые слова: *формальная модель управления доступом, МРОСЛ ДП-модель, Event-B, верификация, дедуктивная верификация, Rodin, метод проверки модели, ProB.*

Введение

В средствах защиты информации (СЗИ), таких, как ОС и СУБД, механизм управления доступом выполняет одну из основных функций по обеспечению их безопасности. Для достижения доверия к корректности этого механизма, создания условий для научного обоснования выполнения им заданных для СЗИ требований безопасности уже многие десятилетия разрабатываются формальные модели управления доступом [1, 2].