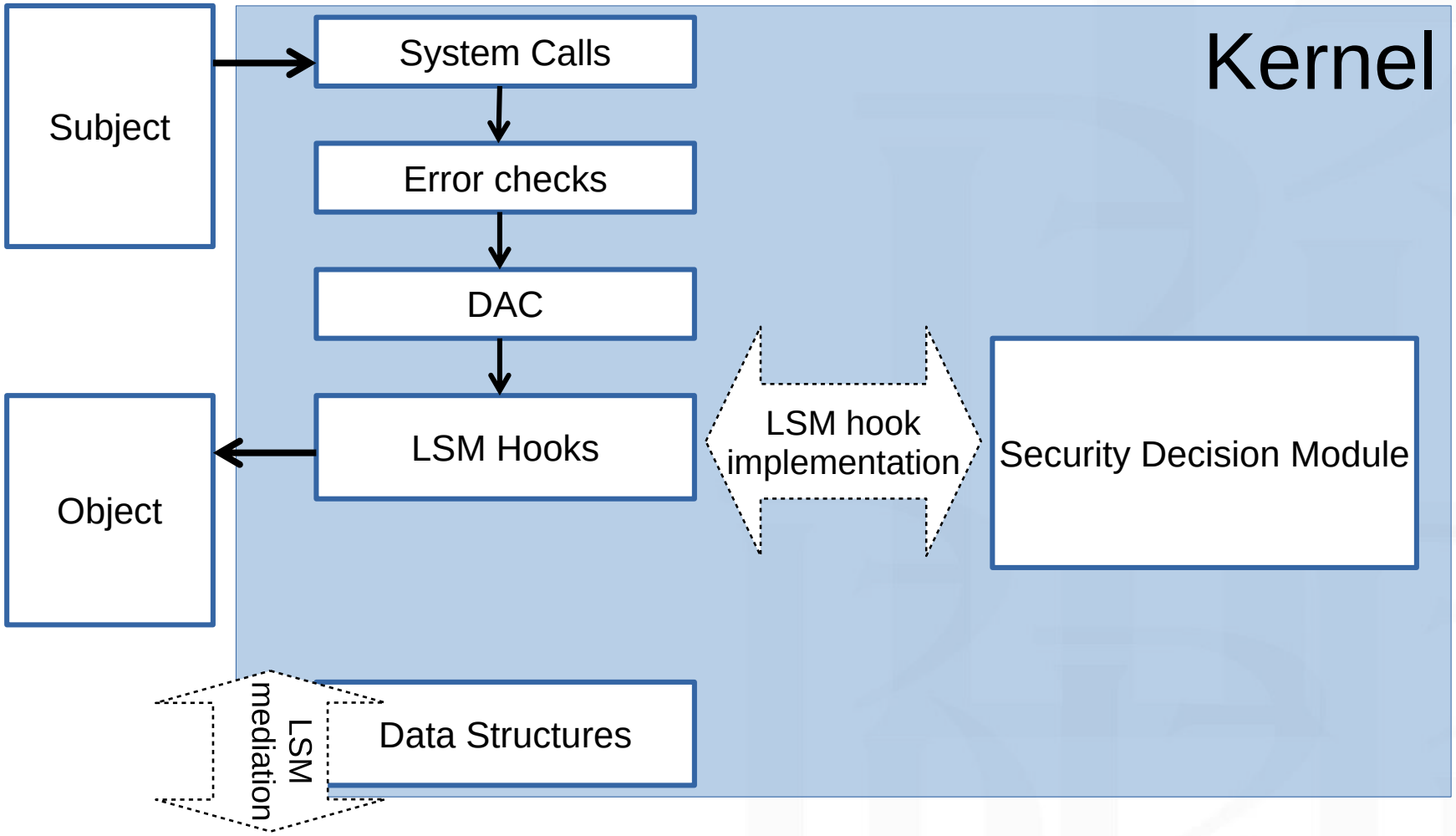
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

Формальная верификация модуля безопасности ядра Linux

Аспирант 1-го года обучения (напр. 05.13.11):
Ефремов Денис (defremov@hse.ru)
Научный руководитель:
проф., д-р физ.-мат. наук Иванников В.П.

Москва, ВШЭ, 16.06.2016

Модуль безопасности ядра Linux



Модуль безопасности Parsec ОС AstraLinux

- Имеет ~10 тысяч строк кода
- ~70 тысяч строк кода подключается из ядра
- Широко используются нестандартные расширения языка
- Ядро Linux навязывает определённый стиль кода
- Спецификации разрабатываются на основе модели в соответствии с реализацией
- Примерно раз в 3 месяца новый релиз

Модель требований к безопасности ядра

- Описывает то, как должно вести себя ядро ОС “в целом”
- Мандатная политика безопасности
- Политика ролевого разграничения доступа
- Пример глобального свойства: информация не может передаваться от объектов с высоким уровнем доступа к объектам с низким уровнем доступа
- Примерно 50 операций, несколько сотен страниц текстового документа
- Модель продолжает развиваться

Уровни применения формальных методов

Нулевой уровень:

Разрабатывается формальная спецификация, затем программный код пишется, глядя на неё.

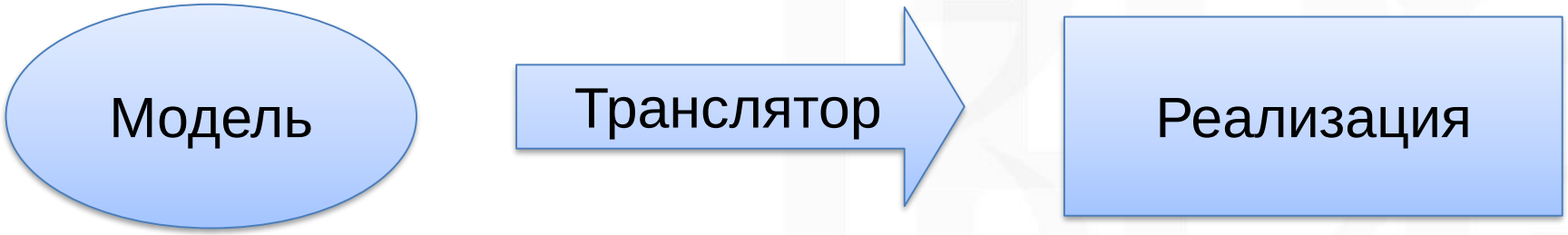
Первый уровень:

Программный код выводится из формальной спецификации автоматически.

Устоявшийся подход

Первый уровень предполагает, как минимум:

- Обоснование корректности модели
- Обоснование корректности трансляции
- Соответствие реализации её модели обосновывается через два предшествующих шага



Ключевые отличия

Проект по верификации модуля безопасности имеет следующие цели:

- Обоснование корректности модели
- Обосновании корректности реализации
- Обоснование корректности соответствия реализации модели

Постановка задачи

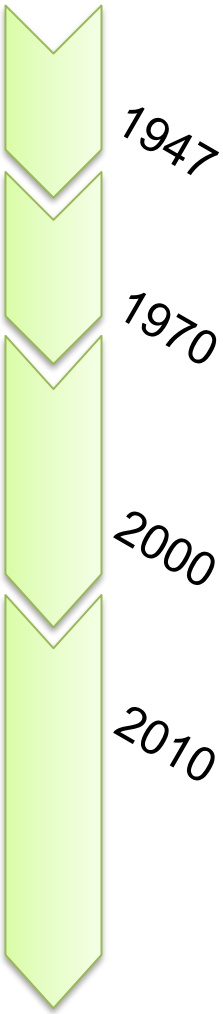
- Математическая модель д.т.н., доцента Девянина П.Н. (УМО ИБ)
- Код ядра ОС специального назначения AstraLinux
- Приблизить существующий нулевой уровень формализации к первому уровню
- Показать формальными методами соответствие или несоответствие реализации её модели

I Обоснование корректности реализации Дедуктивная верификация программ

Если выполнены предположения
- о входных данных программы
- о начальном состоянии
- о поведении окружения программы
- о работе компилятора

Позволяет доказать корректность программы
- для любых входных данных
- для любого начального состояния
- для всех возможных поведений окружения
- программа завершается и выходные данные соответствуют постусловию

I Обоснование корректности реализации Дедуктивная верификация программ



- Лекция Алана Тьюринга Лондонскому математическому обществу
- Методы Флойда/Хоара
- Инструменты дедуктивной верификации для Си, Java, C#
- SunRise, ESC/Java, Frama-C, LOOP, Boogie/VCC
- Применение к реальным проектам небольшого размера
- Атомная энергетика (Англия, Франция)
- Авионика (Airbus, NASA)
- Компоненты специализированных ОС (seL4, Hyper-V)

I Обоснование корректности реализации Ограничения дедуктивной верификации программ

- Трудоёмкость
- Инструменты поддерживают не все конструкции языков программирования
- Применяется для программ небольшого размера
  - обычно не более 10 тыс. строк

I Пример: что можно сказать о функции на языке Си по её коду? (1)

```
int avr(int a, int b)
{
    return (a + b) / 2;
}
```

- Она существует и написана на языке Си.
- Это чистая функция.
- Она вычисляет среднее между двумя целыми числами.
- При определённых условиях возможно целочисленное переполнение.

I Пример: что можно сказать о функции на языке Си по её коду? (2)

```
int avr(int a, int b)
{
    return (a + b) / 2;
}
```

- Возможно ли целочисленное переполнение в том контексте, где функция вызывается?
- Считать ли возможное целочисленное переполнение ошибкой?

I Пример: что можно сказать о функции на языке Си по её коду? (3)

```c
int *binsearch(int *base, int n, int key)
{
    int l = 0;
    int h = n - 1;

    while (l <= h) {
        int m = avr(l, h);
        int val = base[m];

        if (val < key) {
            l = m + 1;
        } else if (val > key) {
            h = m - 1;
        } else {
            return base + m;
        }
    }

    return NULL;
}
```

```c
int avr(int a, int b)
{
    return (a + b) / 2;
}
```

- Контекст: функция двоичного поиска
- Индексы l и h неотрицательны, l не превосходит h
- Возможна ошибка выхода за границу массива при целочисленном переполнении

I Пример: как доказать что код функции корректен? (1)

- Описать контекст вызова:

$$\phi: Z \times Z \to \{\top, \bot\}$$
$$\phi(a, b) \equiv a \geq 0 \wedge b \geq 0 \wedge a \leq b$$

- Описать условия, которым должны удовлетворять результаты:

$$\psi: Z \times Z \times Z \to \{\top, \bot\}$$
$$\psi(a, b, result) \equiv result = \frac{a + b}{2}$$

I Пример: как доказать что код функции корректен? (2)

- Формализовать понятие ошибки:

$$in\_bounds: Z \to \{\top, \bot\}$$
$$in\_bounds(n) \equiv MIN\_INT \leq n \leq MAX\_INT$$

- Формализовать код программы: функция $M[avr]$, которая возвращает результат $M[avr](a, b)$ в соответствии со своим программным кодом если завершается и завершается без ошибки, иначе возвращается специальное значение ω

- Доказать полную корректность:

$$\forall a, b\ \phi(a, b) \Rightarrow \left(M[avr](a, b) \neq \omega\ \&\&\ \psi\left(a, b, M[avr](a, b)\right)\right)$$

I Пример: как доказать что код функции корректен? (3)

- Доказать полную корректность:

$$\forall a, b\ \phi(a, b)$$
$$\Rightarrow \left(M[avr](a, b) \neq \omega\ \&\&\ \psi\left(a, b, M[avr](a, b)\right)\right)$$

- По отдельности safety и постусловия:

1. $\forall a, b\ \phi(a, b) \Rightarrow (M[avr](a, b) \neq \omega\ )$
2. $\forall a, b\ (\phi(a, b) \wedge M[avr](a, b) \neq \omega) \Rightarrow$
   $\psi\left(a, b, M[avr](a, b)\right)$

I Пример: как доказать что код функции корректен? (4)

```
/*@ requires 0 <= a && 0 <= b;
    requires a <= b;
    ensures \result == (a + b) / 2;
 */
int avr(int a, int b)
{
   return (a + b) / 2;
}
```

```
int avr(int a, int b)
{
   return a + (b - a) / 2;
}
```

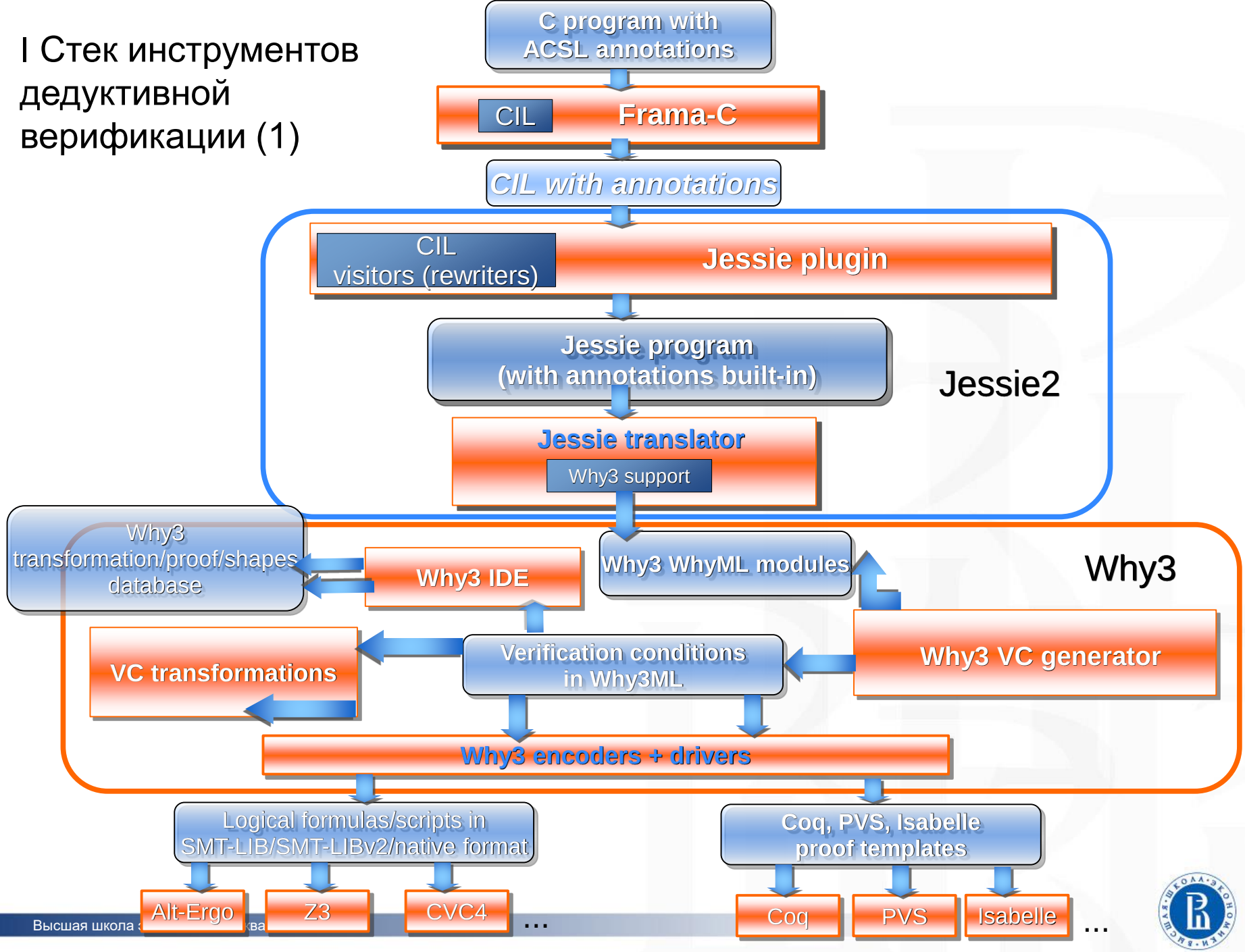
Условие верификации

Исправление кода или более строгие предусловия

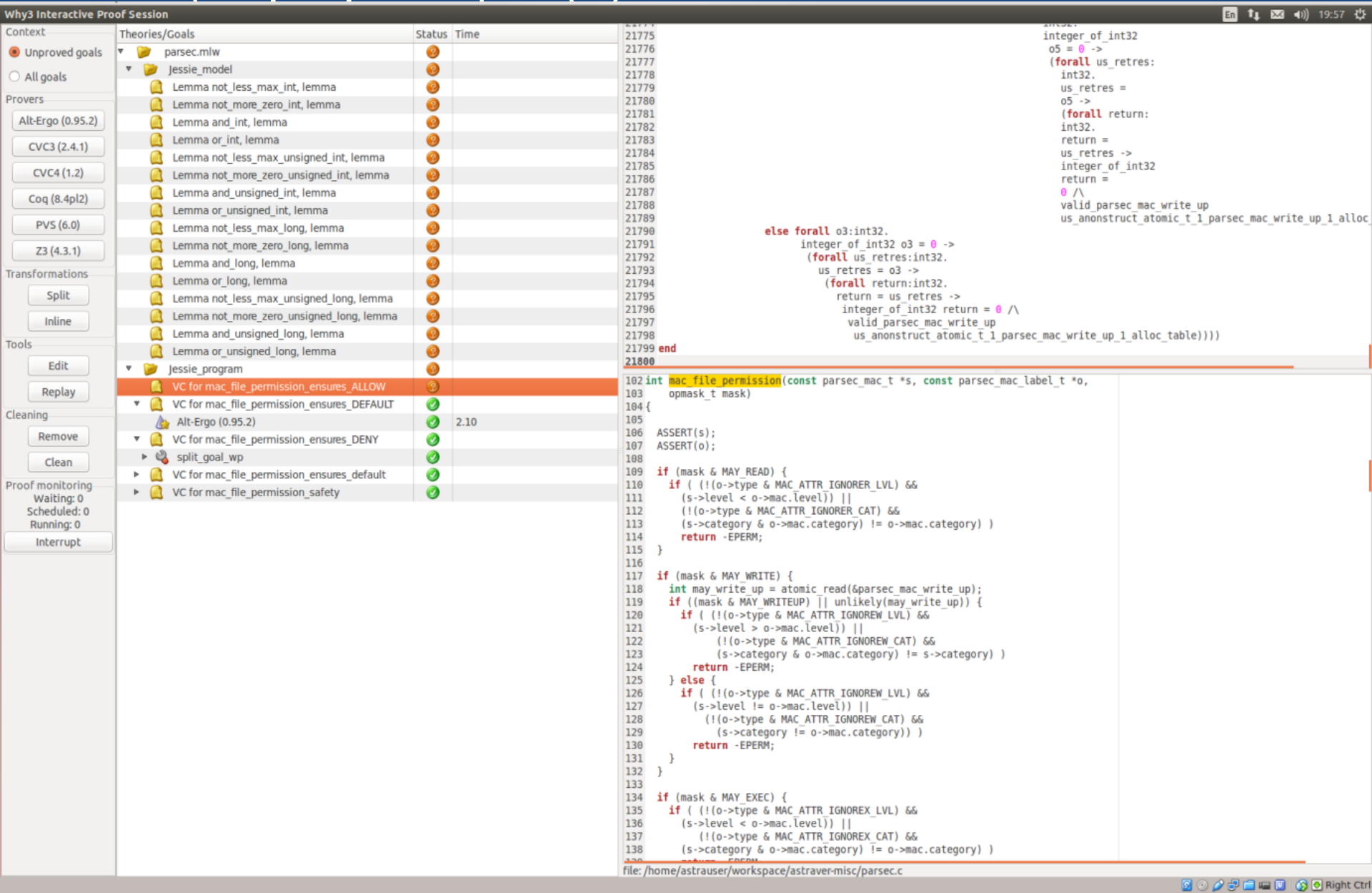
```
predicate in_bounds (n:int) = min <= n /\ n <= max

constant a : t17
constant b : t17

axiom H : of_int 0 <= a /\ of_int 0 <= b /\ a <= b
axiom H1 : in_bounds 2

constant o : t17
axiom H2 : to_int o = 2

goal WP_parameter_avr:
  in_bounds (to_int a + to_int b)
```

```
/*@ requires 0 <= a <= INT_MAX / 2;
    requires 0 <= b <= INT_MAX / 2;
    requires a <= b;
    ensures \result == (a + b) / 2;
 */
int avr(int a, int b)
```

I Стек инструментов дедуктивной верификации (1)



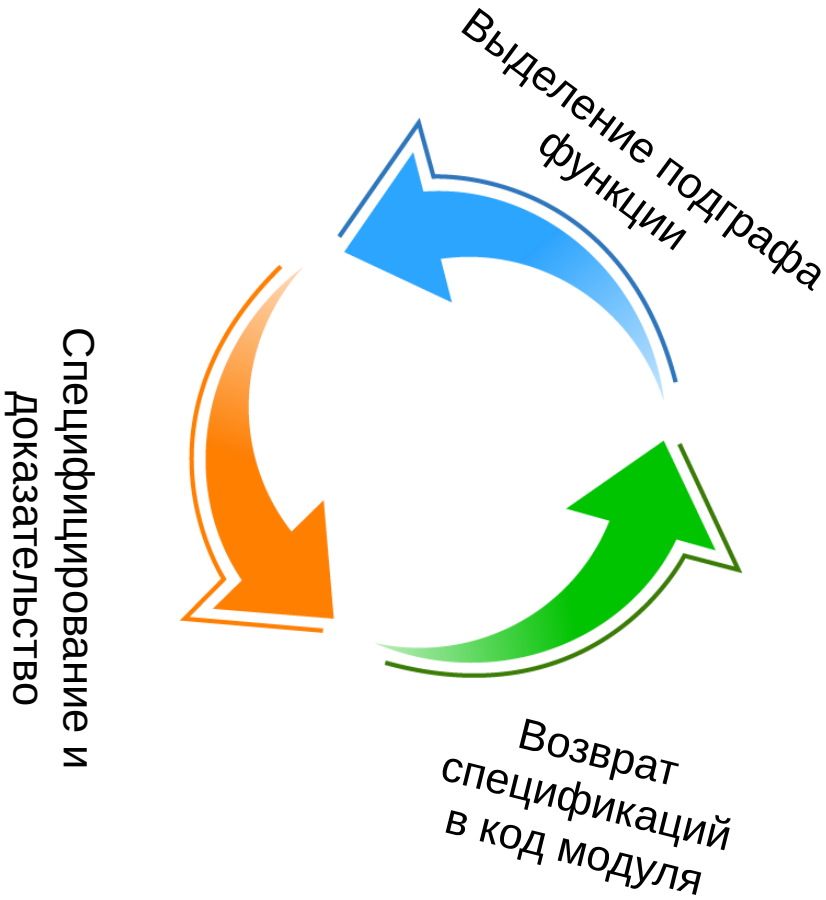
I Стек инструментов дедуктивной верификации (2)

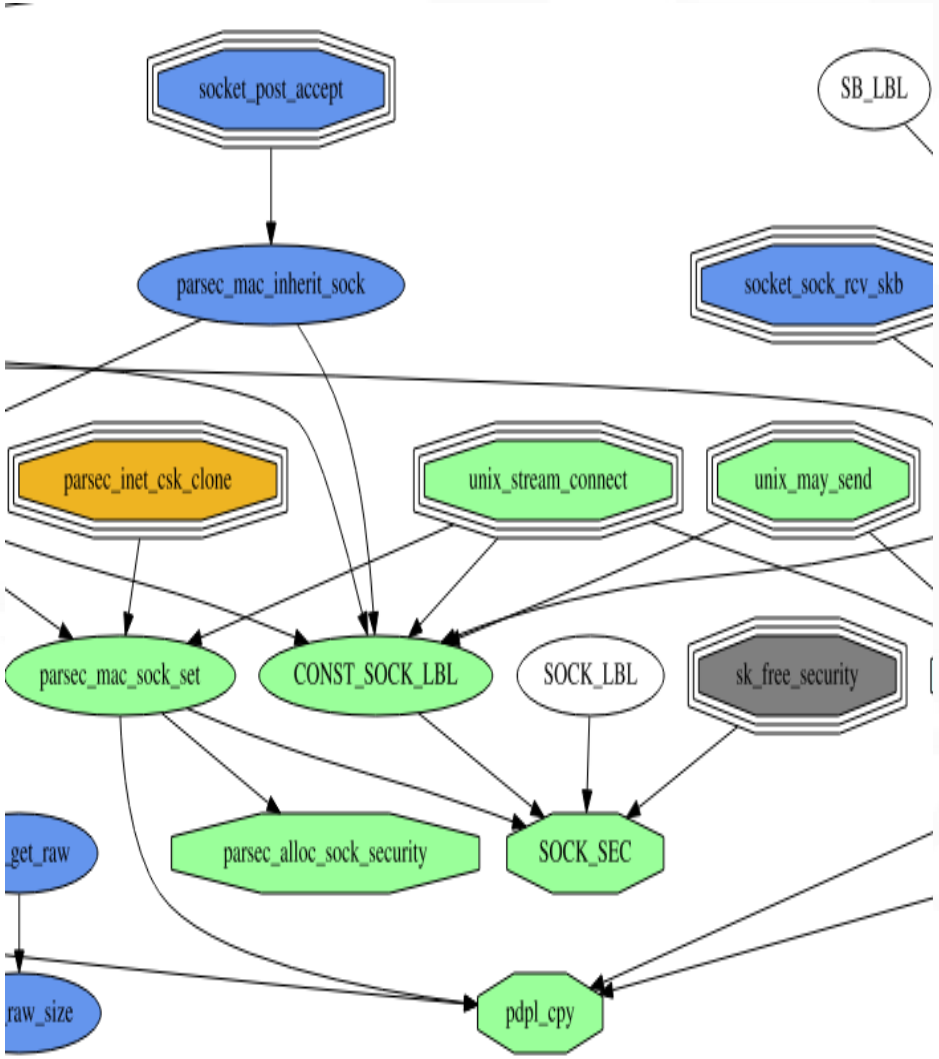


I Обоснование корректности реализации
Подход к разработке спецификаций

- Соглашение с разработчиками о стиле кода
- Приоритеты функций для верификации
- Учёт используемых функций ядра и интерфейсов LSM
- Дополнительные инструменты работы с кодом
- Процесс переноса спецификаций и передоказательства при смене релизов модуля Parsec

I Обоснование корректности реализации
Процесс разработки спецификаций



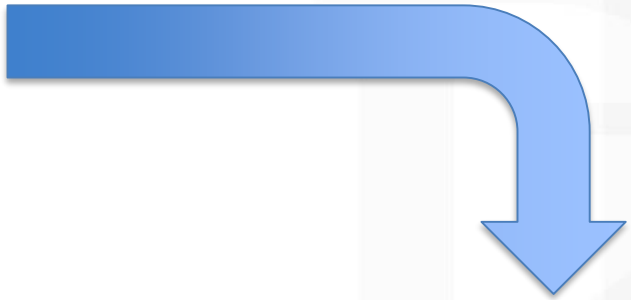


I Обоснование корректности реализации Ядро и модуль?

- Модуль ядра < 10 тыс. строк кода
  - Сформулированы предусловия для всех функций модуля, которые вызываются ядром: для функций LSM интерфейса
  - Они сформулированы исходя из понимания человеком того, в каком контексте должны вызываться
  - LSM интерфейс в документации описан очень кратко
- Ядро Linux с драйверами > 1000 тыс. строк кода
  - Новый релиз раз в несколько месяцев
  - LSM интерфейс расширяется, добавляются точки вызовов
- Как проверить что нет ошибок в спецификации, в описании контекстов вызова функций LSM интерфейса?
- Можно ли что-то проверить на стыке модуля и ядра?

I Пример: трансляция спецификации в исполняемую

```
/*@ requires 0 <= a && 0 <= b;
    requires a <= b;
    ensures \result == (a + b) / 2;
 */
int avr(int a, int b)
{
   return (a + b) / 2;
}
```

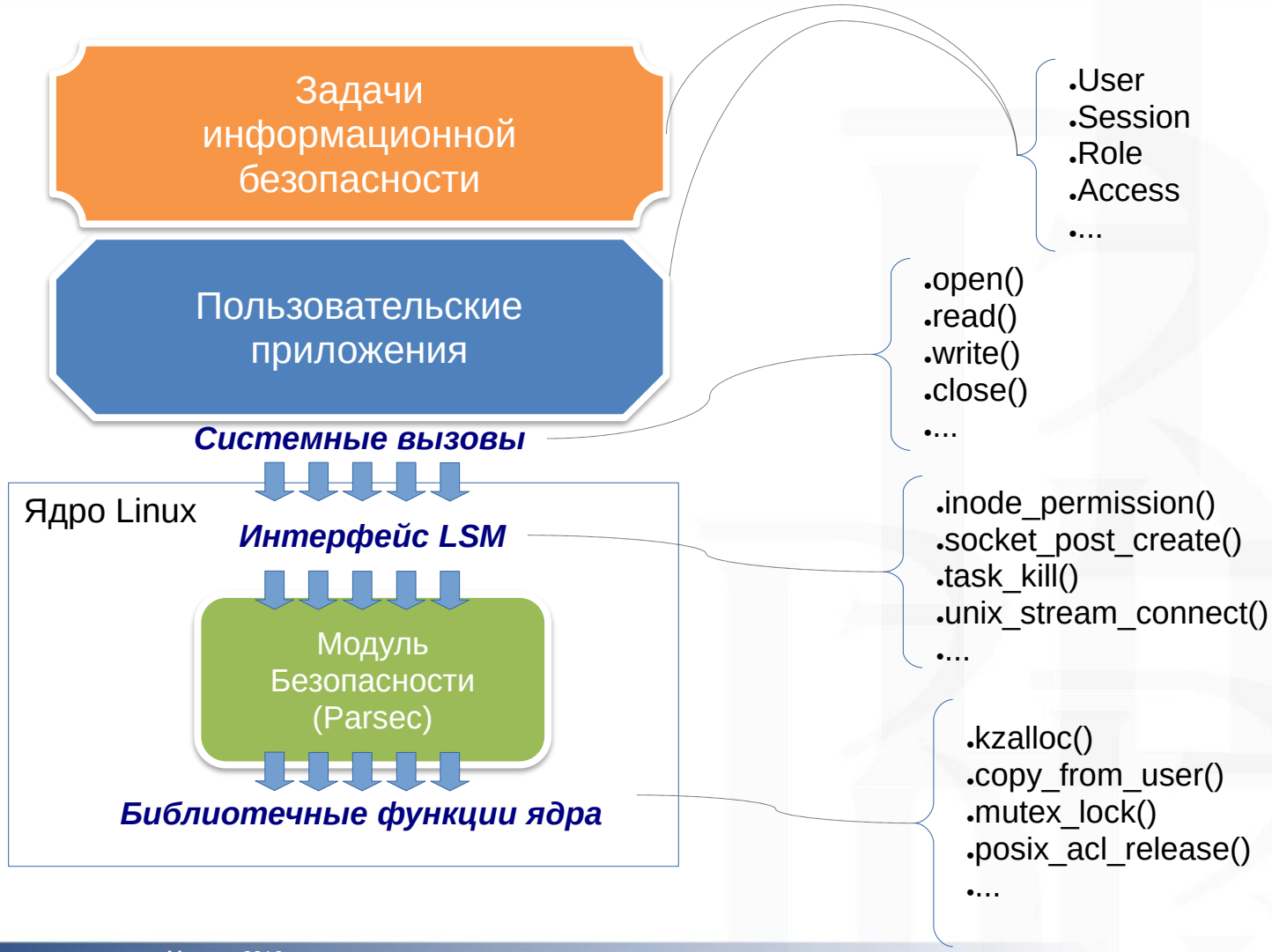


Частично возможно осуществить трансляцию спецификации в проверки на уровне кода.

E-ACSL:
- Длинная арифметика
- Отслеживание состояния памяти
- …

```
#include <assert.h>
int avr(int a, int b) {
    long long da = a, db = b;
    assert(a >= 0 && b >= 0);
    assert(a <= b);
    int r = a + (b - a) / 2;
    assert(r == ((da + db) / 2));
    return r;
}
```

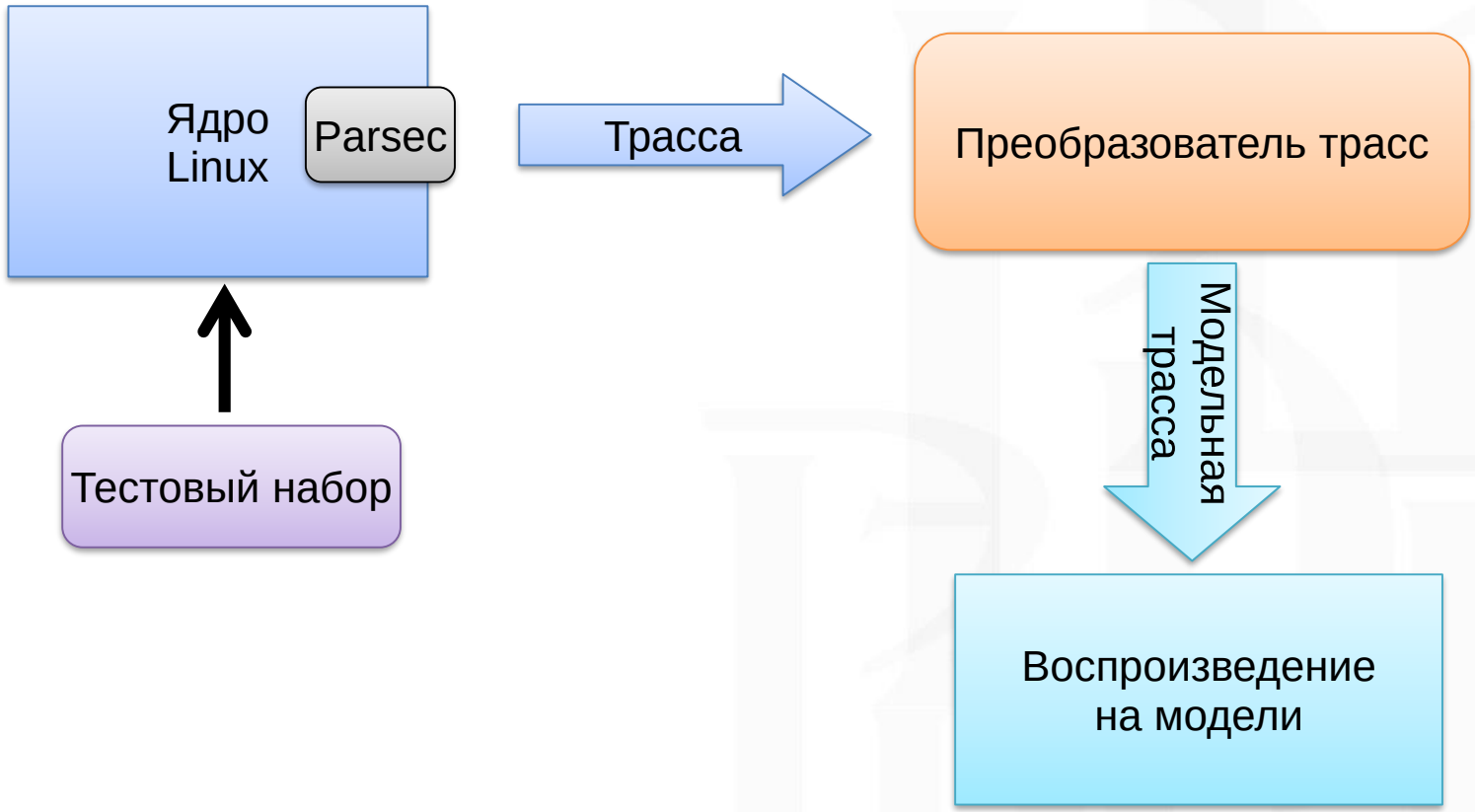
II Модель и модуль безопасности Parsec



II Соответствие модуля безопасности модели

- Модель описывает поведение всего ядра в целом
- Реализация модели заключается в модуле ядра LSM
- Модельная операция
  - open
  - create_user
- Операция в реальности на интерфейсе LSM
  - open -> частичный порядок вызовов {inode_permission*, follow_link*, cap_inode_alloc_security*, cap_d_instantiate*}
  - create_user -> нет прямого соответствия, так как задействованы несколько системных вызовов
  - Чтобы задать корректное соответствие между операциями модели и операциями LSM в модель требуется заложить то, как функционирует ядро
- Альтернативный подход — динамическая верификация

II Соответствие модуля безопасности модели Динамическая верификация (1)



II Соответствие модуля безопасности модели Динамическая верификация (2)

- Разность вердиктов в модели и ядре автоматически означает что трасса не воспроизводится на модели
- Выполнение AstraLinux осуществляется в виртуальной машине
- Трассы снимаются инструментом SystemTap
- В трассу попадают системные вызовы, вызовы интерфейса LSM с точками вызовов
- Ядро имеет встроенное средство для измерения тестового покрытия (GCOV)
- Покрытие в ядре измеряется по ключевым задействованным подсистемам (файловый драйвер, сетевой, ...)

II Соответствие модуля безопасности модели Динамическая верификация (3)

- Покрытие в ядре также измеряется по точкам вызова функций LSM интерфейса
- Трансляция только завершённых трасс
- По модели строится её исполняемая версия
- Покрытие по модели измеряется с учётом выделенных критериев

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

Спасибо за внимание!