



Лекция 6. Методы проектирования и верификации: RAISE, VDM, MBT

А.К.Петренко, А.В.Хорошилов,
Е.В.Корныхин

МГУ ВМК, ИСП РАН

<http://sp.cmc.msu.ru/courses/fmsp>

Осень, 2012



План

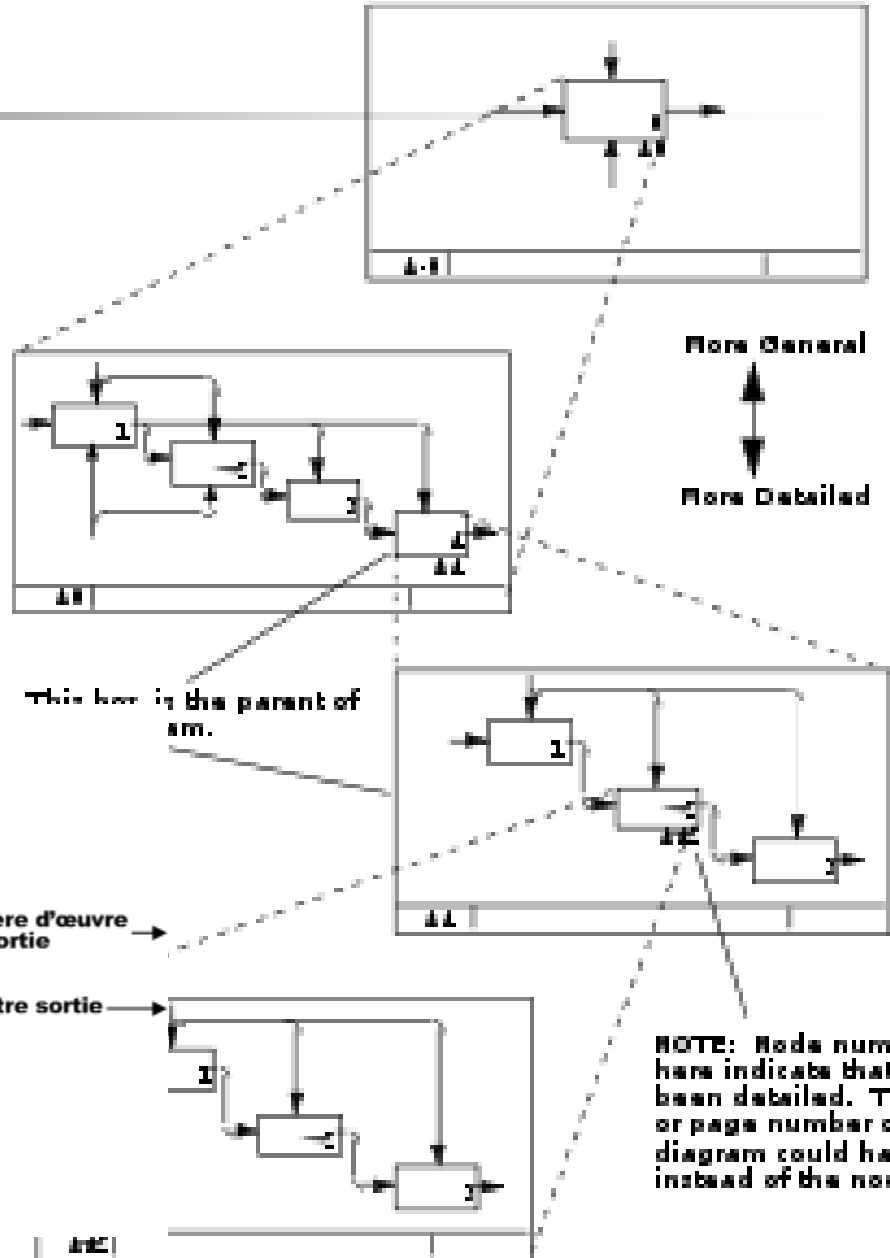
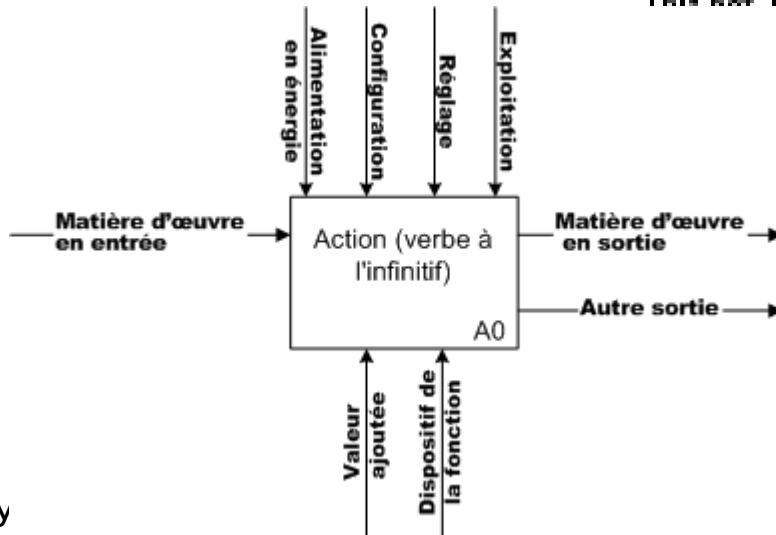
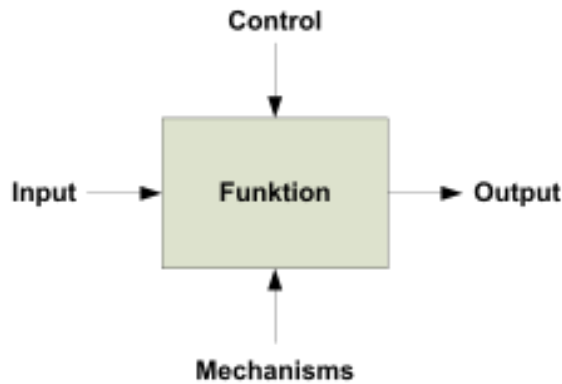
- Классические методы разработки: JSP, JDM, SADT
- Процессы проектирования в жизненном цикле разработки ПО
- Классические методы разработки ПО.
 - Трактовка RAISE.
 - Трактовка VDM.
- Постановка задачи верификации.



JSP, JSD, SADT

- Jackson Structured Programming (JSP)
- Jackson System Development (JSD)
- **Structured Analysis and Design Technique** (SADT)
 - SADT has been developed and field-tested during the period of 1969 to 1973 by Douglas T. Ross and SofTech, Inc..^{[1][4]} The methodology was used in the MIT Automatic Programming Tool (APT) project. It received extensive use starting in 1973 by the US Air Force Integrated Computer Aided Manufacturing program.

SADT



Процессы проектирования и верификации в жизненном цикле разработки ПО

Примеры видов деятельности, направленных на разработку ПО:

- Проектирование

- выделение отдельных модулей и определение связей между ними – определение интерфейсов взаимодействия

- Кодирование

- разработка кода отдельных модулей, разработка документации



Вопросы

- Все ли задачи проектирования перечислены здесь?
- Нужно ли включить задачи верификации в задачи проектирования и кодирования или лучше поставить задачу верификации отдельно?



Вопросы

Что такое верификация?

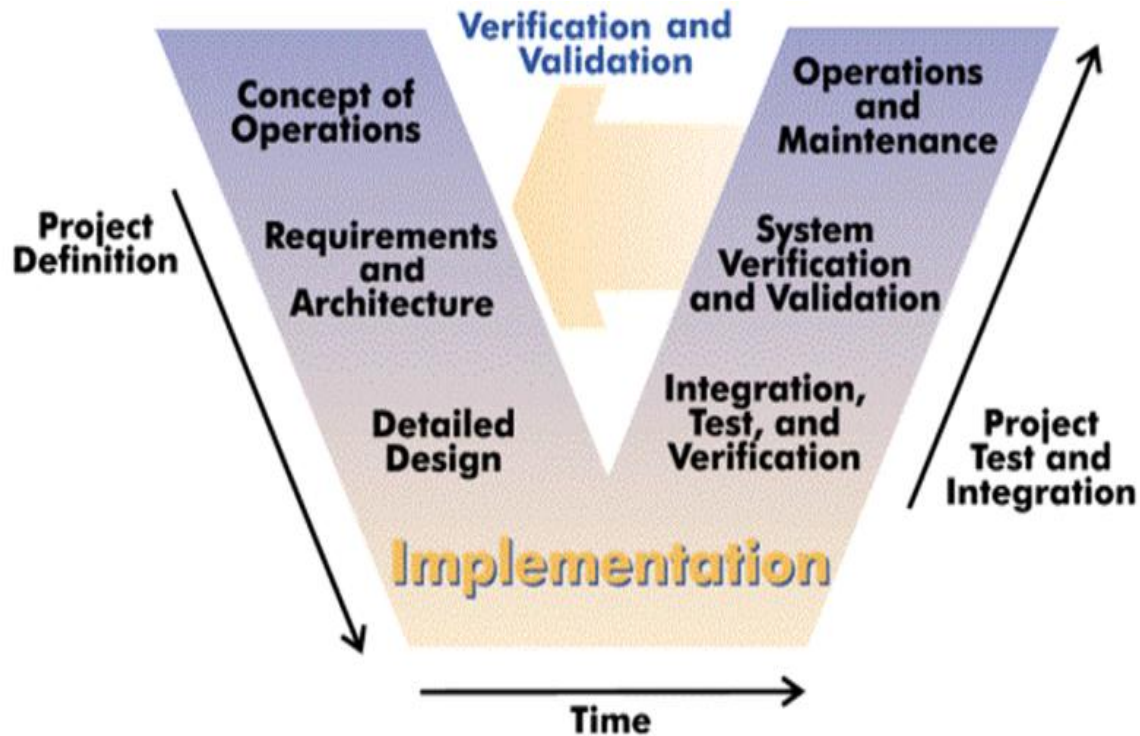
- Верификация обозначает проверку того, что ПО разработано в соответствии со всеми требованиями к нему, или что результаты очередного этапа разработки соответствуют ограничениям, сформулированным на предшествующих этапах.
- Валидация — это проверка того, что сам продукт правилен, т.е. подтверждение того, что он действительно удовлетворяет потребностям и ожиданиям пользователей, заказчиков и других заинтересованных сторон.

Процессы жизненного цикла ПО и систем (ISO 15504)

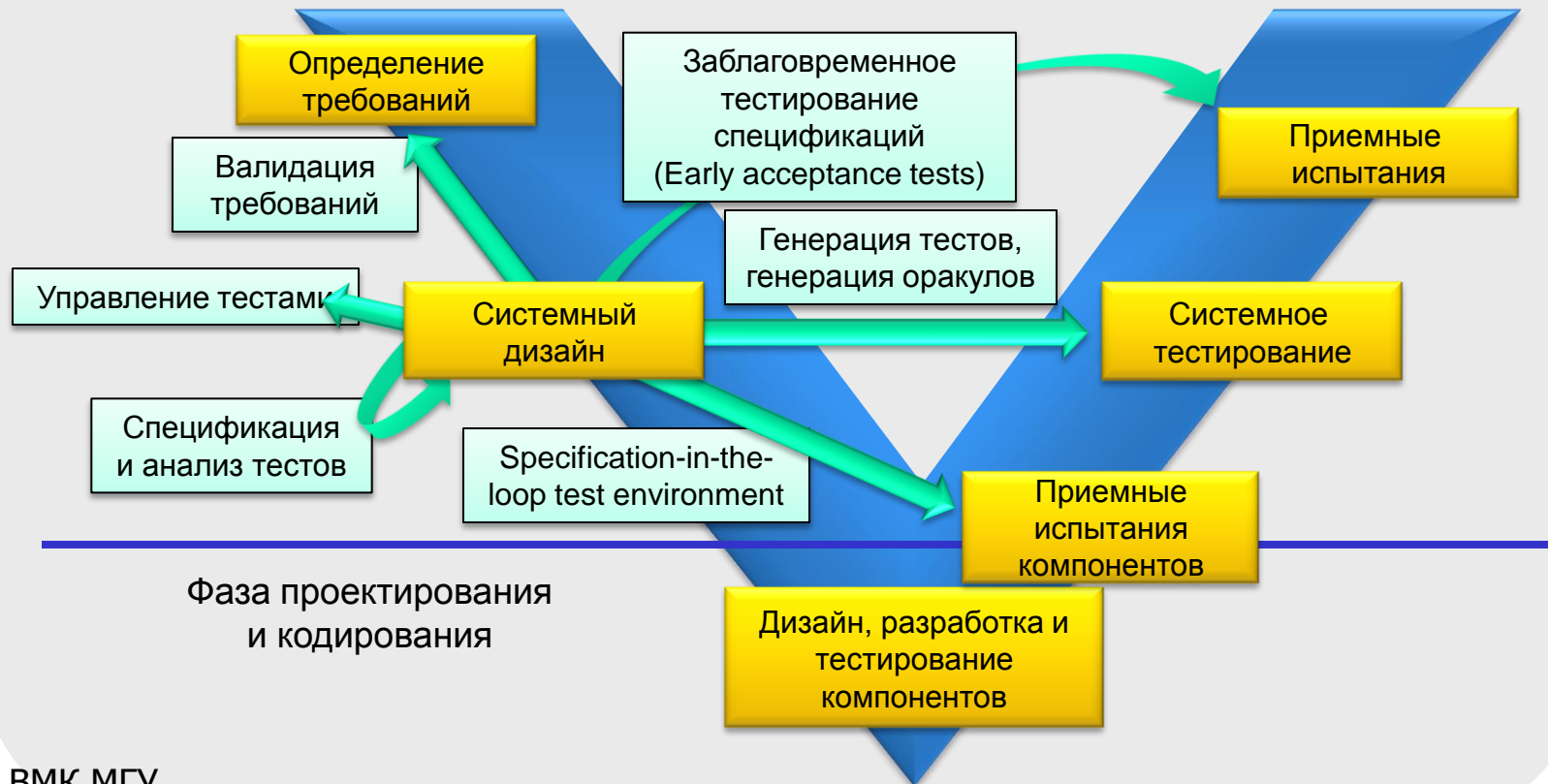
Отношения заказчик-поставщик	Процессы уровня организации	Процессы уровня проекта	Инженерные процессы	Процессы поддержки
Приобретение ПО; Составление контракта; Определение нужд заказчика; Проведение совместных экспертиз и аудитов; Подготовка к передаче; Поставка и развертывание; Поддержка эксплуатации; Предоставление услуг; Оценка удовлетворенности заказчиков	Развитие бизнеса; Определение процессов; Усовершенствование процессов; Обучение; Обеспечение переиспользования; Обеспечение инструментами; Обеспечение среды для работы	Планирование жизненного цикла; Планирование проекта; Построение команды; Управление требованиями; Управление качеством; Управление рисками; Управление ресурсами и графиком работ; Управление подрядчиками	Выделение системных требований и проектирование системы в целом; Выделение требований к ПО; Проектирование ПО; Реализация, интеграция и тестирование ПО; интеграция и тестирование системы; Сопровождение системы и ПО	Разработка документации; Управление конфигурацией; Обеспечение качества; Разрешение проблем; Проведение экспертиз

См. В. В. Кулямин. **Технологии программирования. Компонентный подход**

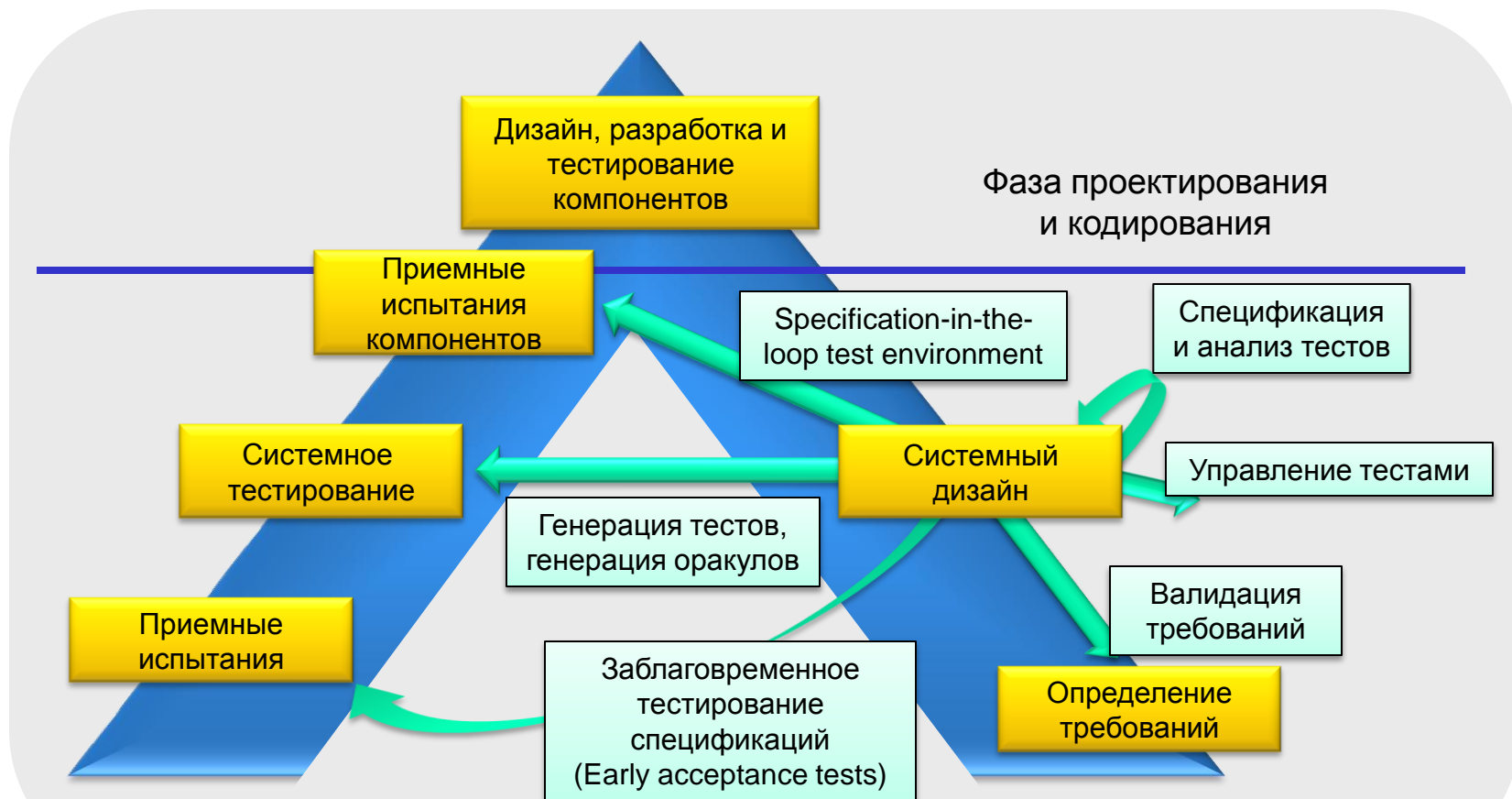
Верификация в жизненном цикле



Процессы анализа и верификации. Требования и системное проектирование



Процессы анализа и верификации. Требования и системное проектирование





Фокус на проектирование модулей

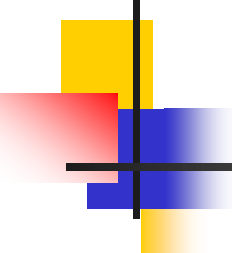
Подход: проектирование «сверху-вниз».

Уровни детализации. Зачем?

- Компактная и понятная модель/интерпретация
 - Простая, следовательно меньше ошибок (? эталон)
 - Простая и понятная заказчику
 - Другая (и более дешевая) – чтобы было с чем сравнивать

Недостатки:

- Сложность поддержки в актуальном состоянии
- Сложность поддержки согласованности уровней
- Позднее проявление проблем на уровне платформы реализации



RAISE Development Method – основные идеи

Разработка разбивается на шаги

- Сначала описывается максимально простая и абстрактная модель
- На каждом шаге строится более подробная и конкретная модель
- На каждом шаге проверяется согласованность моделей



RAISE Development Method – шаги процесса разработки

Шаг 0: объявление сорт-типов и определение сигнатур операций

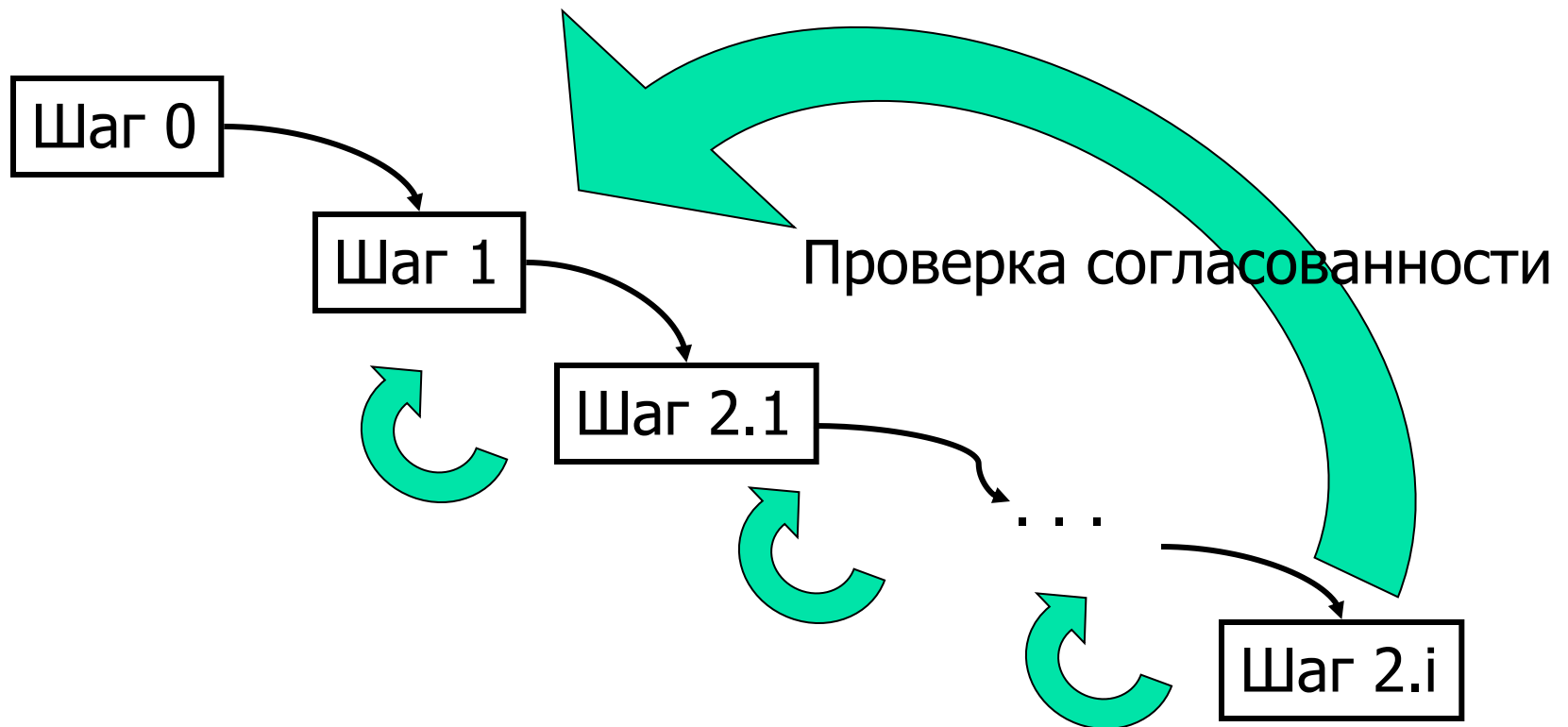
Шаг 1: аксиомы, алгебраическая спецификация

Шаг 2.1: структуры данных и отдельные спецификации для каждой операции (имплицитные и эксплицитные),
доказательство согласованности спецификаций шага 2.1 и шага 1

...

Шаг 2.i: детализация и расширение спецификаций предыдущего шага и доказательство согласованности с предыдущим шагом и с шагом 1. В конце возможна даже трансляция в код на языке программирования.

Шаги RAISE метода



Спецификация сигнатур. Пример

```
scheme DATABASE = class
  type Database, Key, Data
  value
    empty : Database,
    insert : Key >< Data >< Database ->
Database,
    remove : Key >< Database -> Database,
    defined : Key >< Database -> Bool,
    lookup : Key >< Database -~-> Data
end
```




Спецификация сигнатур

- Объявления абстрактных типов данных (так называемые сорт-типы)
- Сигнатуры операций (для некоторых операций сразу можно указать, что они не являются тотальными)
- Один из типов часто играет особую роль – целевой, то есть тот, который моделирует реализацию целевой системы (в данном примере – Database)



Виды операций

- Операции разделяются на два класса по их отношению к целевому типу:
 - генераторы – целевой тип встречается среди выходных параметров (`empty`, `insert`, `remove`)
 - обсерверы (наблюдатели, `observers`) – целевой тип только среди входных параметров (`defined`, `lookup`)



Виды операций (замечание)

Иногда из множества генераторов выделяют минимальное подмножество операций, при помощи которых можно получить любое значение целевого типа, и только эти операции называют генераторами, а остальные – трансформерами (transformers) или преобразователями.



Алгебраическая спецификация

```
scheme AlgS_DATABASE = class
  type Database, Key, Data
  value
    empty : Database,
    insert : Key><Data><Database-> Database,
    remove : Key >< Database -> Database,
    defined : Key >< Database -> Bool,
    lookup : Key >< Database -~-> Data
```

Алгебраическая спецификация (продолжение 1)

axiom

```
[ defined empty ]
```

```
all k:Key :-
```

```
    defined(k,empty) is false,
```

```
[ defined insert ]
```

```
all k,k1:Key, d:Data, db:Database :-
```

```
    defined(k,insert(k1,d,db)) is
```

```
    k = k1 /\ defined(k,db),
```

```
[ defined remove ]
```

```
all k,k1:Key, db:Database :-
```

```
    defined(k,remove(k1,db)) is
```

```
    k ~= k1 /\ defined(k,db),
```

Алгебраическая спецификация (продолжение 2)

```
[ lookup insert ]
all k,k1:Key, d:Data, db:Database :-
    lookup(k,insert(k1,d,db)) ==
    if k = k1 then d
    else lookup(k,db) end
pre k = k1 \ / defined(k,db),
[ lookup remove ]
all k,k1:Key, db:Database :-
    lookup(k,remove(k1,db)) ==
    lookup(k,db)
pre k ~= k1 /\ defined(k,db)
end
```



Алгебраическая спецификация

Общее со спецификацией сигнатур:

- Декларации типов данных как сорт-типов
- Сигнатуры операций

Отличие от спецификации сигнатур:

- Набор аксиом, связывающих последовательности операций



Полнота набора аксиом

На практике в большинстве случаев достаточно написать аксиомы для пар операций

- генератор / обсервер
- трансформер / обсервер

Моделе-ориентированная спецификация (1)

```
scheme ModOS_DATABASE = class
  type Key, Data, Record = Key << Data,
    Database = Record-set
  value
    empty : Database = {},
    insert : Key << Data << Database -> Database
    insert(k,d,db) is remove(k,db) union {(k,d)},
    remove : Key << Database -> Database
    remove(k,db) is db \ {(k,d) | d : Data :- true},
    defined : Key << Database -> Bool
    defined(k,db) is (exists d : Data :- (k,d) isin db),
    lookup : Key << Database -~-> Data
    lookup(k,db) as d post (k,d) isin db
  pre defined(k,db)
end
```

Моделе-ориентированная спецификация (2)

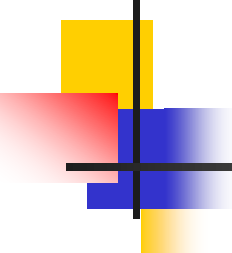
```
scheme ModOS_DATABASE = class
  type Key, Data, Record = Key >< Data,
    Database = { | s : Record-set :
                  is_wf_Database(rs) | }

  value
    is_wf_Database : Record-set -> Bool
    is_wf_Database(rs) is
      (all k : Key, d1, d2 : Data :-
        ( (k, d1) isin rs /\ (k, d2) isin rs)
          => d1 = d2 ),
    empty : Database = {},
```

Database = Record-set

Моделе-ориентированная спецификация (3)

```
insert : Key >< Data >< Database -> Database
insert(k,d,db) is remove(k,db) union {(k,d)},
remove : Key >< Database -> Database
remove(k,db) is db \ {(k,d) | d : Data :-
true},
defined : Key >< Database -> Bool
defined(k,db) is (exists d : Data :- (k,d)
isin db),
lookup : Key >< Database -~-> Data
lookup(k,db) as d post (k,d) isin db
pre defined(k,db)
end
```



Сравнение модели-ориентированной спецификации и алгебраической

- Определения типов данных (хотя, возможно, не всех),
 - часто - подтипов при помощи вспомогательных функций (is_wf_...)
- Имплицитные или эксплицитные спецификации операций



Отношение уточнения (refinement)

Две схемы находятся в отношении «одна уточняет или реализует другую» (refinement или implementation), если

- реализация сохраняет объявления всех сущностей, объявленных в абстрактной схеме, при этом
- сорт типы могут заменяться описанием типа
- подтипы могут заменяться своими максимальными типами
- могут появляться объявления и описания новых сущностей и свойств
- все свойства (аксиомы) абстрактной спецификации справедливы в реализации



Техника re-writing проверки согласованности моделей

Доказательство согласованности двух спецификаций (доказательство отношения «уточняет») состоит из цепочки вида

цель₁

аргументация₁

.....

цель_n

аргументация_n

Аргументация – это ссылка на правила re-writing, см. The RAISE Development Method.



Пример («реализация»)

```
scheme ModOS_DATABASE = class
type          Record = Key >< Data,
Database = {| rs : Record-set :- is_wf_Database(rs) |},          Key, Data
value        is_wf_Database : Record-set -> Bool,
empty : Database,
insert : Key >< Data >< Database -> Database,
remove : Key >< Database -> Database,
defined : Key >< Database -> Bool,
lookup : Key >< Database -~-> Data
axiom forall k : Key, d : data, rs : Record-set, db : Database :-
is_wf_Database(rs) is (all k : Key, d1, d2 : Data :- ((k,d1) isin rs /\
  (k,d2) isin rs) => d1 = d2),
empty is {},
insert(k,d,db) is remove(k,db) union {(k,d)},
remove(k,db) is db \ {(k,d) | d : Data :- db),
defined(k,db) is (exists d : Data :- (k,d) isin db),
lookup(k,db) as d post (k,d) isin db          pre defined(k,db)
end
```

Пример: доказательство выполнения аксиомы [defined_empty]

```
scheme AlgS_DATABASE =  
class  
type Record, Database, Key, Data  
value empty : Database,  
insert : Key >< Data >< Database -> Database,  
remove : Key >< Database -> Database,  
defined : Key >< Database -> Bool,  
lookup : Key >< Database ->> Data  
axiom  
[ defined empty ]  
all k:Key :-  
  defined(k, empty) is false,  
[ defined insert ]  
all k,k1:Key, d:Data, db:Database :-  
  defined(k,insert(k1,d,db)) is  
  k = k1  $\vee$  defined(k,db),
```

```
[ defined remove ]  
all k,k1:Key, db:Database :-  
  defined(k,remove(k1,db)) is  
  k  $\sim$  k1  $\wedge$  defined(k,db),  
[ lookup insert ]  
all k,k1:Key, d:Data, db:Database  
:-  
  lookup(k,insert(k1,d,db)) is  
  if k = k1 then d  
  else lookup(k,db) end  
pre k = k1  $\vee$  defined(k,db),  
[ lookup remove ]  
all k,k1:Key, db:Database :-  
  lookup(k,remove(k1,db)) is  
  lookup(k,db)  
pre k  $\sim$  k1  $\wedge$  defined(k,db)  
end
```


Пример (доказательство корректности уточнения)

ModOS_DATABASE реализует DATABASE поскольку:

- ModOS_DATABASE определяет все типы, определенные в DATABASE.
- ModOS_DATABASE определяет все константы и функции, определенные в DATABASE, и определяет их с такими же сигнатурами.
- Все аксиомы из DATABASE истинны в ModOS_DATABASE.

Пример доказательства для аксиомы [defined_empty]

```
[[defined(k,empty) is false]] unfold empty:  
[[defined(k,{}) is false]] unfold defined:  
[[exists d : Data :- (k,d) isin {}] is false]] isin_empty:  
[[exists d : Data :- false) is false]] exists_introduction:  
[[false is false]] is_annihilation:  
[[true]]  
qed
```



Ограниченность RAISE-метода

При уточнении нельзя:

- изменять сигнатуры
- изменять однажды определенные структуры данных

Венский метод

Vienna Development Method – VDM (1)

Шаг 0-ой: объявление сорт-типов и определение сигнатур операций

Шаг 1: аксиомы, алгебраическая спецификация

Шаг 1: структуры данных и спецификации для каждой операции (имплицитные и эксплицитные), доказательство корректности спецификаций

Шаг i: модификация сигнатур и структур данных и расширение спецификаций предыдущего шага, доказательство корректности и согласованности с предыдущим шагом.

В конце возможна даже трансляция в код на языке программирования.

Венский метод

Vienna Development Method – VDM (2)

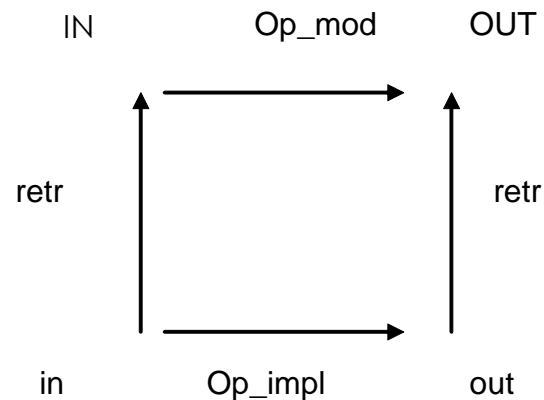
Имея имплицитную и эксплицитную спецификацию, нужно доказывать, что

$$\text{pre-Op}(\text{input}) \Rightarrow \text{post-Op}(\text{input}, \text{Op}(\text{input}))$$

Это надо доказывать на каждом уровне детализации.

Венский метод Vienna Development Method – VDM (3).

Как доказывать соответствие моделей разного уровня детализации?



Обозначения:

- Op_mod - операция в модельном (спецификационном) пространстве
- Op_impl - операция в реализационном пространстве
- in, out - входные и выходные данные в реализационном пространстве
- IN, OUT - входные и выходные данные в модельном пространстве
- retr - восстанавливающая функция (функция абстракции)

Постановка задачи верификации. Модель и реализация заданы явно

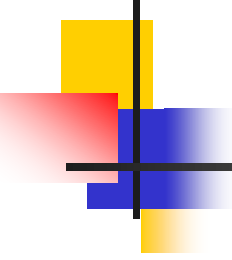
- Пред-условия абстрактного уровня (далее будем говорить *модели* и использовать суффикс *_mod*) не слабее пред-условий уточненного уровня (далее будем говорить *реализации* и использовать суффикс *_impl*):

$$\text{pre-Op_mod}(\text{retr}(\text{in})) \Rightarrow \text{pre-Op_impl}(\text{in})$$

- При условии, что предусловия не нарушены, результат выполнения функции *Op_mod* эквивалентен результату функции *Op_impl*, преобразованного функцией *retr*:

$$\begin{aligned} &\text{pre-Op_mod}(\text{retr}(\text{in})) \Rightarrow \\ &\text{eq}(\text{retr}(\text{Op_impl}(\text{in})), \text{Op_mod}(\text{retr}(\text{in}))), \end{aligned}$$

где *eq* – функция, задающая определение эквивалентности значений из области значений функции *Op_mod*.



Постановка задачи верификации. Модель неявная, реализация явная

В этом случае для доказательства соответствия нужно убедиться, что для каждой уточненной функции:

- пред-условия модели функции не слабее пред-условий реализации
 $\text{pre-Op_mod}(\text{retr}(\text{in})) \Rightarrow \text{pre-Op_impl}(\text{in})$
- пост-условие модели выполняется по отношению к входным данным, полученным трансформацией входных данных реализации и результатам, полученным трансформацией выходных данных реализации (при условии, что пред-условие для модели выполняется)

$$\text{pre-Op_mod}(\text{retr}(\text{in})) \Rightarrow \\ \text{post-Op_mod}(\text{retr}(\text{in}), \text{retr}(\text{Op_impl}(\text{in})))$$



Постановка задачи верификации. Модель и реализация неявные

- В этом случае для доказательства соответствия нужно убедиться, что для каждой уточненной функции
 - пред-условие модели не слабее пред-условия реализации:
$$\text{pre-Op_mod}(\text{retr}(\text{in})) \Rightarrow \text{pre-Op_impl}(\text{in})$$
 - пост-условия модели не сильнее пост-условий реализации:
$$\text{pre-Op_mod}(\text{retr}(\text{in})) \ \& \ \text{post-Op_impl}(\text{in}, \text{out}) \Rightarrow \text{post-Op_mod}(\text{retr}(\text{in}), \text{retr}(\text{out}))$$



Вопрос. Что делать, если строго доказать не удастся?

Ответ -

Разработать конечный набор тестов и при этом:

- Строго сформулировать предположения (гипотезы) – какие мы делаем допущения, полагаясь на протестированную программу.
- Строго сформулировать критерии адекватности/полноты набора тестов.
- Строго сформулировать критерии корректности
- Строго сформулировать критерии соответствия между моделями и реализацией