Письма в ЭЧАЯ. 2023. Т. 20, № 4(249). С. 676–685

СОВЕРШЕНСТВОВАНИЕ УПРАВЛЕНИЯ ЭКСПЕРИМЕНТОМ В ПРОГРАММНОМ КОМПЛЕКСЕ SONIX+

А. С. Кирилов, Л. А. Трунтова [1]

Объединенный институт ядерных исследований, Дубна

Работа посвящена совершенствованию программного комплекса Sonix+ на измерительных установках реактора ИБР-2 для минимизации потерь измерительного времени при включении математических действий (суммирования матриц, построения спектральных распределений интенсивности на основе данных из списка событий и проч.) непосредственно в процедуру эксперимента на языке Python (скрипт). Для этого выполнение всех подобных операций было выделено из основного скрипта в один или несколько фоновых. Для выполнения фоновых скриптов в структуру комплекса добавлен специальный класс устройств. Помимо этого, применявшийся ранее для связи пользовательского интерфейса со скриптом механизм контрольных точек был заменен новым — на основе декораторов и контекстных менеджеров языка Python. Это позволило существенно упростить скрипт, повысить его надежность, а также сократить время его разработки или модификации.

The paper is devoted to enhancing the Sonix+ software package at the IBR-2 instruments to minimize the loss of measurement time when including mathematical operations (summing matrices, building spectral distributions of intensity based on data from the list of events, etc.) directly in the experimental procedure in the Python language (script). For this purpose, the execution of all such operations was separated from the main script into one or more background scripts. A special device class has been added to the complex structure to execute background scripts. Besides this, the checkpoint mechanism previously used to connect the user interface with the script has been replaced by a new one — based on Python language decorators and context managers. This significantly simplified the script, improved its reliability, and reduced the time of its development or modification.

PACS: 07.05.–t; 07.05.Bx

Программный комплекс Sonix+ разработан в ЛНФ ОИЯИ для управления экспериментальными установками на реакторе ИБР-2. На сайте [1] изложена подробная информация об истории создания, основных принципах организации и структуре комплекса.

Хотя в целом структура Sonix+ и основные принципы его организации сохраняются неизменными, сам комплекс постоянно совершенствуется вместе с развитием аппаратуры установок, методики проведения измерений, обновлением используемого

[1] E-mail: ltruntov@jinr.ru

программного обеспечения. Длительное эволюционное развитие любого программного проекта время от времени приводит к необходимости выполнять так называемый рефакторинг — переработку кода с возможным изменением внутренней структуры программы, не затрагивающую ее функциональности. В последнее время это стало актуально и для нашего проекта.

Если в начале 2000-х гг. пользовательский скрипт, т. е. запись последовательности действий эксперимента на языка Python, представлял собой преимущественно линейно выполняемую последовательность операций, в которой отсутствовали длительные по времени вычисления, то впоследствии ситуация существенно изменилась. По мере внедрения на установках реактора ИБР-2 позиционно-чувствительных детекторов (ПЧД) возникла необходимость выполнения значительных по объему вычислений непосредственно в ходе эксперимента. Например, измерения на рефлектометрах организованы как последовательность коротких экспозиций, результаты которых необходимо суммировать on-line. Данные с ПЧД могут быть представлены как в виде спектральных распределений интенсивностей в формате трехмерных массивов (гистограмм), так и в виде списка отдельных событий (так называемых list-mode файлов). Последние также необходимо перевести в формат спектральных распределений, т. е. выполнить их гистограммирование. Реализация этих действий в основном потоке непродуктивно увеличивала общее время выполнения эксперимента.

Начальное решение с выделением вычислений в отдельные подпроцессы (subprocess) Python оказалось неудачным. Оно существенно усложнило библиотеку операций, затруднило ее отладку и, в целом, оказалось не очень надежным в рамках существующей на тот момент версии управления измерением. Последнее обстоятельство привело к переосмыслению этого внутреннего механизма Sonix+.

ОРГАНИЗАЦИЯ УПРАВЛЕНИЕМ ИЗМЕРЕНИЯ ПЕРЕД РЕФАКТОРИНГОМ

В комплексе Sonix+ пользователь может управлять устройствами как в ручном режиме, так и в автоматическом через скрипт. На практике автоматический режим управления является основным. На рис. 1 приведена схема управления измерением в этом режиме. В Sonix+ существенную часть содержания скрипта представляют вызовы так называемых *операций библиотеки*. Эта библиотека уникальна для каждой установки, так как составляется согласно принятой в ней методике измерений. Каждая *операция* оформлена по определенным правилам Sonix+ в виде процедуры на языке Python.

Для выполнения скрипта в комплексе предусмотрен специальный модуль — *интерпретатор*, отмеченный на схеме прямоугольником *Is*. Связь виджета управления измерением графического пользовательского интерфейса (*GUI*) и интерпретатора происходит по протоколу, реализованному на языке Python в *системных скриптах*. Взаимодействие модулей в рамках Sonix+ организовано через хранилище данных *Varman* [6] (в принятых терминах «база данных» — далее *БД*). С полным составом модулей Sonix+ можно также ознакомиться на сайте [1].

До последних изменений текущей версией интерпретатора была третья версия [5]. В ней выполнение скрипта может быть временно приостановлено (*Stop*), возобновлено

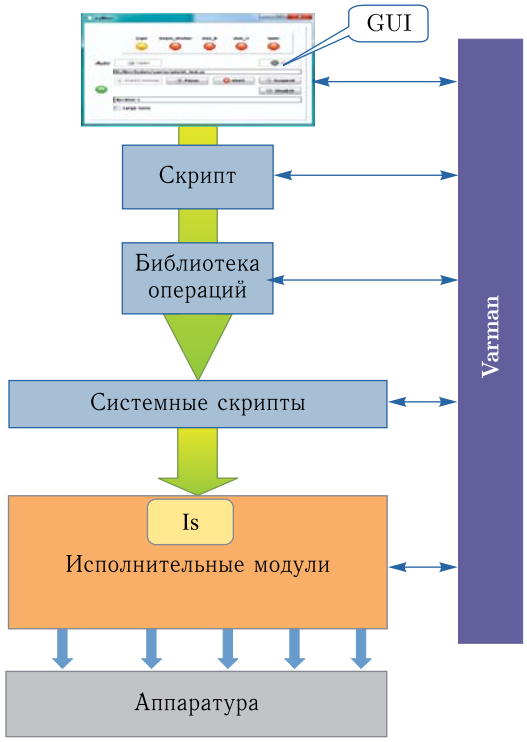

Рис. 1. Схема управления измерением до рефакторинга

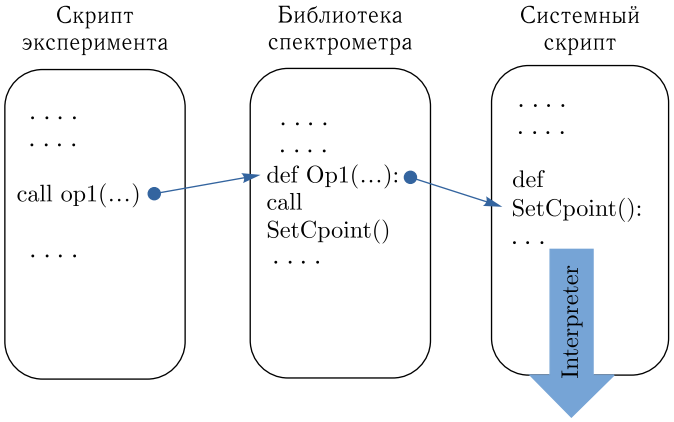

Рис. 2. Схема управления измерением по методу контрольных точек

(*Continue*), экстренно прекращено (*Abort*) или закончено с предварительным выполнением отложенных действий (*Stop&Exit*). Управление процессом измерения было реализовано через механизм контрольных точек (системный скрипт *cont_points*), который подробно рассмотрен в работе [6]. По этой методике (рис. 2) одна или, по

необходимости, несколько контрольных точек помещаются непосредственно в код операции, и именно в них осуществляется взаимодействие с интерпретатором. Там же должны быть отработаны действия по команде *Abort*. С другой стороны, выполнение команды *Stop&Exit* зависит от контекста измерения, поэтому ее реализация не может быть запрограммирована в системном скрипте и выносится в код операции.

ПРИЧИНЫ ОТКАЗА ОТ ИСПОЛЬЗОВАНИЯ SUBPROCESS

Начальным решением поставленной задачи было вынесение всех вычислительных процессов в независимый фоновый подпроцесс (subprocess), который синхронизировался с основным через отдельные переменные *БД*. Этот вариант усложнял реализацию команды *Abort*. Недостаточно было просто оборвать экспозицию в основной ветви скрипта, необходимо также было дождаться завершения действий и в фоновой. Наиболее ярко эффект проявляется там, где присутствует вложенность операций. Например, на фурье-инструментах процедура набора данных при заданном графике вращений фурье-прерывателя происходит в контексте запуска и остановки экспозиции. Она встроена в операцию с протоколом прерывателя (открытие и закрытие), а все измерение начинается и заканчивается открытием и закрытием температурного протокола. Для того чтобы прервать подобное измерение корректно, необходимо пройти через все эти уровни вложенности и закрыть их.

Ситуация еще более усложнилась при подключении установок к центральному репозиторию [7]. Дело в том, что окончательная запись данных в репозиторий должна происходить во всех случаях завершения измерения, в том числе и при фатальных ошибках в процессе выполнения скрипта. При этом необходимо максимально полно выполнить фоновую обработку данных. Все это требует особо тщательного разбора возможных вариантов, что еще больше усложняет программирование операций библиотеки, особенно с учетом возможных вложенностей.

Для того чтобы исключить зависимость кодов отработки *Abort* и *Stop&Exit* в новых условиях, требовалось заменить механизм контрольных точек другим, обеспечивающим разделение кода собственно операции и кода отработки команд с учетом контекста возможных вложенностей.

ОСНОВНЫЕ ИДЕИ РЕШЕНИЯ

Основные идеи предложенного решения можно сформулировать в следующем:

• разделение скрипта на *основной* и *фоновые*. *Основной* скрипт ведет эксперимент, в *фоновые* выносятся все затратные по времени вычисления;

• для управления ходом интерпретации вместо контрольных точек — применение *декораторов* Python;

• для упрощения реализации команд *Abort* и *Stop&Exit* для вложенных операций — использование *менеджера контекста* Python.

Новая схема управления измерением проиллюстрирована на рис. 3. Для выполнения фоновых действий разработано специализированное устройство (на схеме — Bg), представляющее собой самостоятельный интерпретатор с FIFO-буфером команд на

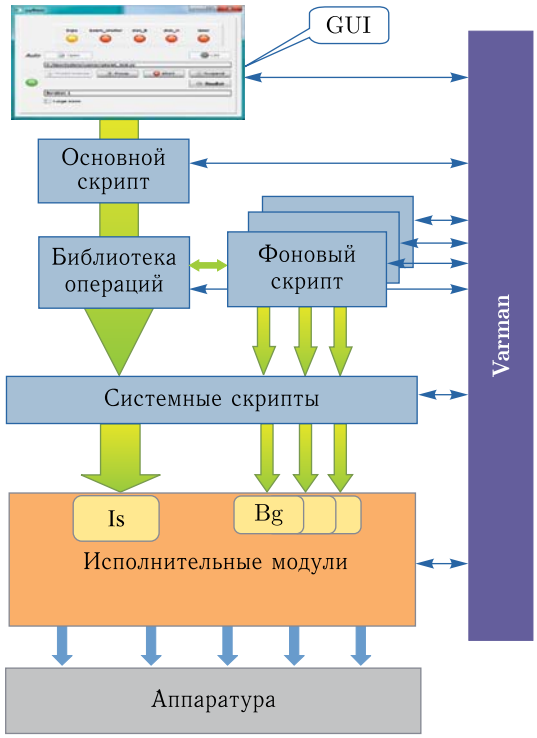

Рис. 3. Управление измерением после рефакторинга комплекса

входе. Более подробно этот класс устройств описан в прил. 1. Количество фоновых устройств определяется по потребности. Например, для дифрактометра ФДВР предусмотрено два устройства: одно для выполнения вычислений, другое — для передачи данных в хранилище.

Разделение скриптов повышает взаимную защищенность. Кроме того, появляется возможность оперативно отслеживать состояние фоновых процессов с помощью программы *Reflector* или соответствующего виджета.

ИЗМЕНЕНИЯ В ПРАВИЛАХ ОФОРМЛЕНИЯ ОПЕРАЦИЙ БИБЛИОТЕКИ

Как отмечалось выше, сложность программирования операций библиотек вызвана в первую очередь необходимостью выполнять адекватные в контексте измерения действия по реакции на команды *Abort* и *Stop&Exit*. С этой целью, а также для большей формализации оформления операций механизм контрольных точек был исключен. Вместо него связь пользовательского интерфейса со скриптом была организована с помощью так называемых *декораторов*. Декоратор в Python — это функция, которая позволяет «обернуть» другую функцию для расширения ее функциональности без непосредственного изменения ее кода (см. пример на рис. 4).

```
def sonix_command(cmd):
  def decorated(*args):

    res = cmd(*args)              # Выполнение операции
    if res[0] == 1:               # Ошибка?
      finalize_all_background_processes() # Завершение фоновых процессов
                                          # по ошибке
    elif res[0] == 0:
      check_after_command() # Проверка условий продолжения измерения и завершение
                            # его в случае указания пользователя
  return decorated

. . . . . .

@sonix_command        #                          декоратор
def Expo(File_name, Expo_time, Expo_mode): # операция библиотеки
. . .
return
```

Рис. 4. Иллюстрация объявления и пример применения декораторов в библиотеке

В новой версии комплекса в декораторы вынесена обработка сигналов управления и результатов выполнения операций, которые, особенно при возможных ошибках, будут гарантированно выполнены по единому алгоритму. На примере (рис. 4) показано, как происходит оформление операции с помощью декоратора. В нем декоратор *sonix_command* проверяет условие преждевременного завершения эксперимента и при необходимости корректно завершает его. При этом код самой операции *Expo* не изменяется.

Декораторы могут применяться последовательно. При необходимости можно составить набор декораторов для всех возможных ситуаций.

С использованием декораторов в системном скрипте *cont_points* оставлена только процедура запуска скрипта *RunScript* с обработкой исключений и буферизацией содержимого стандартных выводов (*output*, *error* и *final*), которые протоколируются обычным порядком.

ПРИМЕНЕНИЕ МЕНЕДЖЕРА КОНТЕКСТА ДЛЯ ВЛОЖЕННЫХ ОПЕРАЦИЙ

Контекстные менеджеры в Python позволяют задать поведение при работе с конструкцией *with* при входе и выходе из блока. Это упрощает работу с ресурсами в части их захвата и освобождения. Применение contextmanager из модуля contextlib для вложенных операций гарантирует отработку реакции на действия пользователя для каждой операции независимо от уровня вложенности этих операций.

Рис. 5 иллюстрирует действие этого метода на простейшем примере. Две операции *protocol*() и *chopper*() библиотеки, оформленные с помощью contextmanager, выполняются поверх цикла из трех команд print. Печать вывода показывает, что обе операции вызываются дважды: до цикла, выполняя заданное начальное действие, и после цикла — завершающее. Важно, что если бы на месте операторов *print* были реальные операции библиотеки, то независимо от наличия возможных ошибок завершение операций *protocol* и *chopper* было бы выполнено.

Задание операций

```
@contextmanager
def protocol():
    print('Open protocol')
    try:
        yield
    finally:
        print('Close protocol')

@contextmanager
def chopper():
    print('Start chopper')
    try:
        yield
    finally:
        print('Stop chopper')
```

Фрагмент кода

```
...
with protocol(),chopper():
    for i in range(0,3):
        print('iteration ', i)
...
```

Печать результата

```
Open protocol
Start chopper
iteration  0
iteration  1
iteration  2
Stop chopper
Close protocol
```

Рис. 5. Иллюстрация применения контекстного менеджера для вложенных операций. В левом столбце приведен код операций, в правом — пример кода вызова и печать результата

ДРУГИЕ ИЗМЕНЕНИЯ В БИБЛИОТЕКАХ

Помимо применения декораторов и менеджеров контекста для упрощения горизонтальных связей между операциями определен новый класс Experiment. В этом классе аккумулированы все параметры конкретного эксперимента, которые становятся доступны операциям через пространство глобальных переменных.

В заключение согласно правилам классического рефакторинга произведена чистка библиотек от неэффективных языковых конструкций, выделение общих фрагментов кода в отдельные модули, удаление неиспользуемых переменных и т. д.

НОВАЯ ВЕРСИЯ ИНТЕРПРЕТАТОРА

Радикальные изменения в правилах организации библиотек были дополнены сравнительно небольшой корректировкой модуля основного интерпретатора. Была удалена команда *Stop*, которая предназначалась для временной приостановки измерения с тем, чтобы за время паузы экспериментатор мог что-либо подправить в оборудовании. Практика показала невостребованность этой команды. Кроме того, в связи с изменением новой схемы отработки команд *Abort* и *Stop&Exit*, был упрощен дескриптор интерпретатора и состав его команд (см. прил. 2).

ЗАКЛЮЧЕНИЕ

Рассматриваемая версия комплекса с новым модулем интерпретатора и исправленным графическим интерфейсом успешно испытана в октябрьском цикле 2021 г. на спектрометре НЕРА. В настоящее время аналогичные версии подготовлены для дифрактометров ФДВР и ФСД. Эти установки, наряду с рефлектометрами, обладают

наиболее сложной методикой измерения и, соответственно, сложными библиотеками операций. До пуска реактора ИБР-2 предполагается провести испытания этих версий в максимально полном режиме, с тем чтобы к моменту возобновления измерений подготовить версии для остальных установок.

Авторы выражают глубокую признательность коллегам из отдела НЭОКС ЛНФ за ценные замечания и поддержку работы, а также ответственным за установки за помощь и терпение при работе с новым программным обеспечением.

Отдельная благодарность — В. И. Боднарчуку за замечания и существенную помощь в подготовке рукописи.

Приложение 1
УСТРОЙСТВО ДЛЯ ФОНОВЫХ ОПЕРАЦИЙ BACKGROUND_DEV

Устройство предназначено для запуска фонового скрипта и управления его выполнением. Передача команд и/или параметров производится через FIFO-буфер в виде строк. Максимальный размер строки 4000 байт. Для передачи переменных или структур Python можно воспользоваться доступным сериализатором, например, json [8].

Для конфигурации устройства, помимо стандартных параметров, необходимо задать полный путь к файлу фонового скрипта.

На рис. 6 приведен снимок дескриптора устройства, а в табл. 1 перечень его команд.

В настоящее время синхронизация фоновых скриптов с основным выполняется полностью через дескриптор соответствующего фонового устройства. При необходимости более широкого обмена данными можно воспользоваться дополнительным устройством типа fifo_dev. Это устройство реализует буфер обмена по технологии FIFO.

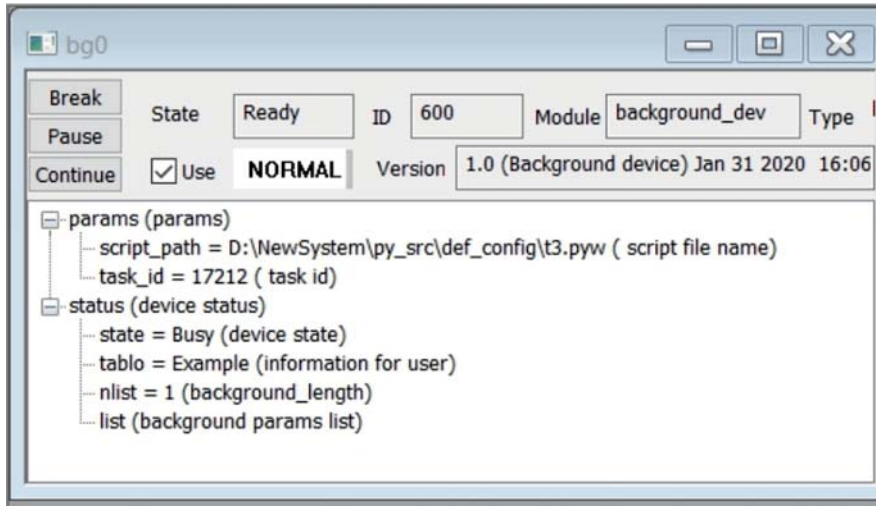

Рис. 6. Пример дескриптора фонового устройства

Таблица 1

Команда	Описание
clear ()	Очистить входной буфер
push (str)	Добавить строку (команду) во входной буфер
str = pop()	Считать очередную команду из буфера
start()	Запустить скрипт, заданный в конфигурации (внешний скрипт)
set_id (id)	Записать ID процесса скрипта (определяется в скрипте) в дескриптор устройства
set_state(state)	Установить статус устройства из скрипта
set_tablo(str)	Записать комментарий в статус
stop()	Завершить выполнение скрипта без выгрузки модуля
bg_exit()	Завершить выполнение скрипта и выгрузить модуль

Приложение 2
ИНТЕРПРЕТАТОР ВЕРСИИ 4

Изменения в модуле интерпретатора и связанных с ним системных скриптах преследовали две основные цели — изменение схемы отработки команд Abort и Stop&Exit и упрощение функционала модуля за счет удаления невостребованных по практике команд. В частности, из набора команд и дескриптора было исключено все, что относилось к реализации команд Stop и Continue пользовательского интерфейса. На рис. 7 приведен вариант нового дескриптора.

Также был упрощен набор состояний (табл. 2).

Из набора команд интерпретатора исключены команды для пошагового режима (табл. 3).

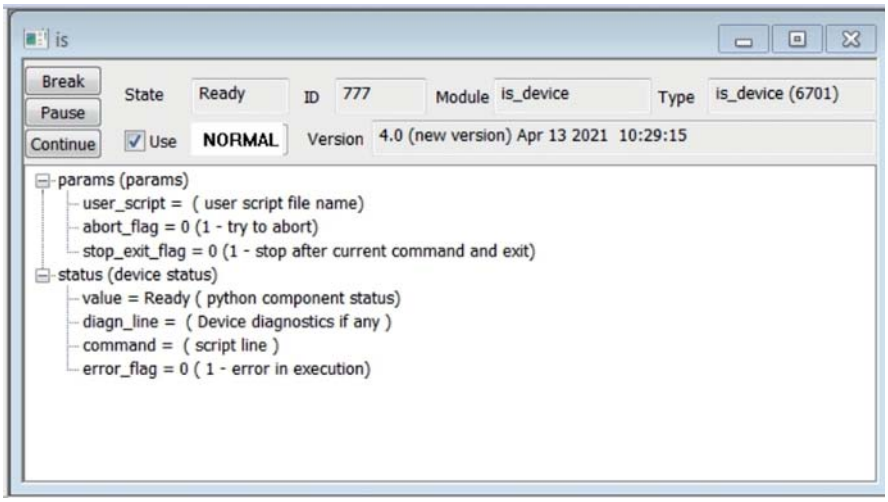

Рис. 7. Дескриптор модуля интерпретатора версии 4

Таблица 2

Состояние	Описание
None	Модуль выгружен
Ready	Готов к работе (сразу после перезагрузки и инициирования Python)
Busy	Процесс интерпретации выполняется
Susp	Процесс интерпретации приостановлен

Таблица 3

Команда	Описание
pystart (file_name)	Запуск эксперимента (переход в состояние Busy)
pystop_and_exit	После завершения текущей команды закончить эксперимент (функция Stop&Exit)
pyabort	Немедленно прервать эксперимент
pysuspend	Немедленно приостановить текущее выполнение (переход в состояние Susp)
pycontinue	Продолжить из состояния Susp
pyset_script (file_name)	Задать путь к файлу скрипта
pyset_script_line (line)	Записать в дескриптор текущую команду скрипта
pyset_error_flag	Установить флаг ошибки
pyset_diagn_line	Записать строку комментария в дескриптор
is_load	Загрузить/перегрузить модуль интерпретатора

СПИСОК ЛИТЕРАТУРЫ

1. https://sonix.jinr.ru/wiki/doku.php?id=ru:index
2. https://matplotlib.org/
3. https://numpy.org/
4. https://www.riverbankcomputing.com/software/pyqt/
5. *Кирилов А. С.* Эволюция модуля интерпретатора в инструментальном программном комплексе Sonix+. Сообщ. ОИЯИ Р10-2017-88. Дубна, 2017.
6. *Кирилов А. С., Юдин В. Е.* Реализация базы данных реального времени для управления экспериментом в среде MS Windows. Сообщ. ОИЯИ Р13-2003-11. Дубна, 2003.
7. *Кирилов А. С., Морковников И. А.* О концепции файлового хранилища для спектрометров ИЯУ ИБР-2. Сообщ. ОИЯИ Р10-2018-23. Дубна, 2018.
8. https://www.json.org/json-en.html

Получено 13 сентября 2022 г.