Использование языка программирования Python для описания ограничений на архитектурные модели[1]

[2]Е.В. Корныхин <kornevgen@cs.msu.ru>
[1,2,3,4]А.В. Хорошилов <khoroshilov@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3]Московский физико-технический институт (государственный университет),
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9
[4] Национальный исследовательский университет «Высшая школа экономики»
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. В данной статье предлагается подход к описанию и верификации структурных ограничений на архитектурные модели, в основе которого лежит переиспользование возможностей языка программирования Python, инструментов, библиотек, документации и методических материалов для языка Python. Использование в качестве основы широко известного языка должно уменьшить порог вхождения в предлагаемый подход. Ограничения становятся частью архитектурной модели, в идеале они разрабатываются вместе с моделью. Ограничения записываются на языке программирования Python в виде функций с одним аргументом (он обозначает проверяемый компонент модели) и возвращаемым значением логического типа, снабженных специальным декоратором (исполнимой аннотацией). Чтобы проверить выполнение ограничений для модели, генерируется и выполняется программа на языке Python. В этой программе создается архитектурная модель и затем выполняются нужные функции-ограничения.

Подход был реализован в среде MASIW Framework – это среда моделирования программно-аппаратных систем на языке AADL, выполненная на базе широко известной среды разработки Eclipse. В среду MASIW Framework были интегрированы инструменты разработки на языке программирования Python – это инструмент PyDev, хорошо известный разработчикам на Python в среде Eclipse. Этот инструмент упрощает выполнение программ на Python, содержит в себе мощный редактор программ на Python с раскраской кода и автодополнением. Такие возможности удалось

[1] Исследование проводилось при финансовой поддержке РФФИ в рамках проекта №14-07-00443

задействовать из-за особенностей генерируемых исходных кодов на Python: классы строятся из компонентов модели, поля классов – из подкомпонентов, соединений и т.п., методы – из ограничений, иерархия классов и пакетов – из иерархии компонентов и пакетов исходной архитектурной модели.

Ключевые слова: архитектурное моделирование; верификация моделей; язык программирования Python, язык моделирования AADL.

DOI: 10.15514/ISPRAS-2015-27(5)-8

Для цитирования: Корныхин Е.В., Хорошилов А.В. Использование языка программирования Python для описания ограничений на архитектурные модели. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 143-156. DOI: 10.15514/ISPRAS-2015-27(5)-8.

1. Введение

Архитектурное моделирование становится важным этапом разработки сложных систем ответственного применения. Оно позволяет на ранних этапах разработки выявлять ошибки и, тем самым, уменьшать затраты на разработку. Обычно архитектурное моделирование выполняется с использованием специализированных языков и сред моделирования, важной составляющей которых является функциональность по описанию ограничений на модели и верификации моделей. Поскольку основной фокус архитектурных моделей заключается в описании структуры целевой системы и связей между ее компонентами, то и ограничения на модели в первую очередь рассматриваются структурные в отличие от ограничений на поведение целевой системы в динамике.

В разных средах моделирования проверка структурных ограничений строится на основе различных подходов. Например, среда может проверить модель по тем требованиям, которые включены в язык моделирования. Такие требования обычно описываются в руководствах по языку моделирования, и их поддержка реализована непосредственно в среде моделирования. Любая модель должна выполнять все требования языка моделирования. Например, если в модели описана некоторая иерархическая структура из компонентов, обозначающая, что компонент одного уровня иерархии является частью компонента более высокого уровня, и для каждого компонента определен размер, то размер объемлющих компонентов не должен превышать сумму размеров вложенных компонентов.

Однако таких встроенных требований недостаточно для тщательной проверки: необходимо учитывать функциональные требования, требования внутренней согласованности модели, требования из предметной области, которые могут быть различными для разных моделей. При архитектурном моделировании проверка таких требований играет ключевую роль. Поскольку они могут быть различными для разных моделей, они не могут быть предопределенными, поэтому необходим способ записи этих требований пользователем. Способ

записи должен быть однозначен, понятен среде моделирования и достаточен для автоматической проверки модели.

Например, для языка архитектурного моделирования AADL (Architecture Analysis and Design Language) для этой цели был предложен специализированный язык REAL (Requirements and Enforcements Analysis Language) [1]. Спецификация на языке REAL состоит из набора «теорем», которые итеративно обходят модель, делают промежуточные вычисления и проверку различных условий. REAL проектировался как простой язык для описания ограничений. В частности, мы в своей практике проверки моделей столкнулись с недостатками этого языка: в нем нет ряда структур данных и операций, удобных, а иногда и необходимых для записи сложных ограничений. Некоторые из недостатков были решены в других языках, развивающих идеи языка REAL, но далеко не все [2]. Например, сложные проверки имеет смысл разбивать на более мелкие, особенно если более мелкие проверки повторяются в разных больших проверках. Причем необходима возможность отдельного описания часто используемых проверок или промежуточных вычислений и их использования для более крупных проверок и вычислений (то, что языках программирования достигается за счет механизмов подпрограмм и рекурсии). Таких возможностей нет ни в REAL, ни в его последователях, что ограничивает их практическое применение.

Гибкость – не единственное важное требование к языку описания ограничений. Приходится учитывать, насколько сложно изучить его конечным пользователям.

В данной статье мы предлагаем подход к построению языка описания ограничений, ключевая идея которого состоит в переиспользовании мощных возможностей традиционных языков программирования в удобном для пользователя виде. Также мы рассмотрим вопросы реализации подхода на примере языка архитектурного моделирования AADL.

2. Требования к языку описания ограничений

Язык описания ограничений нуждается в развитых возможностях, привычных для многих языков программирования, в частности, в богатом наборе выражений, операторов, переиспользуемых компонентов кода (функций и библиотек функций). Эти возможности можно разработать специально для того или иного языка описания ограничений, а можно воспользоваться возможностями существующего языка программирования, если разработать на его основе язык описания ограничений. Последний подход позволяет не только воспользоваться продуманными мощными возможностями языка, но и существующими инструментами разработки, библиотеками компонентов кода и опытом разработчиков. Пользователи могут быть уже знакомыми с данным языком программирования или они смогут воспользоваться имеющейся литературой, обучающими курсами или знаниями коллег по данному языку программирования.

С другой стороны, язык описания ограничений должен быть интегрирован с языком архитектурного моделирования. Он должен позволять описывать ограничения в терминах языка архитектурного моделирования, должен обеспечивать удобный синтаксис для доступа к компонентам модели и указания, для каких компонентов нужно проверять выполнение тех или иных ограничений. В языке AADL последнее решается за счет встроенного механизма расширений (annex). Решение остальных упомянутых проблем требует дополнительного исследования, но в качестве основы можно взять саму модель на языке AADL.

Язык описания ограничений может по-разному использовать язык программирования. Например, язык описания ограничений может переиспользовать только некоторые синтаксические конструкции (выражения, операторы и т.п.), а остальное в этом языке (в частности, семантика конструкций) будет сделано по-своему. Но возможно и более глубокое переиспользование, в котором правильные тексты на языке программирования будут правильными текстами и на языке описания ограничений. Последний подход позволит переиспользовать среды разработки, компиляторы, интерпретаторы, средства документирования, средства отладки и т.п. Однако при этом возможно чрезмерное усложнение синтаксиса и ухудшение читабельности кода, особенно если в языке программирования нет достаточных средств расширения.

Кроме вышеперечисленного при выборе языка программирования следует учитывать его распространенность и сложность изучения. Пользователям будет проще изучить и начать правильное использование расширения (известного им) языка программирования, чем полноценного нового языка спецификации.

Поскольку многие из перечисленных выше характеристик выполнены для языка программирования Python, мы предлагаем его в качестве основы для языка описания ограничений для языка архитектурного моделирования AADL. Мы реализовали поддержку этого языка в рамках среды архитектурного моделирования MASIW Framework [3, 4].

Мы считаем возможности языка программирования Python достаточно развитыми, существует большое число библиотек компонентов, реализованных на Python, существует большое число обучающих курсов и методической литературы. Инструменты разработки Python имеются практически для всех широко используемых сред разработки (в частности, и для среды Eclipse, на основе которой реализована среда MASIW Framework). В нашем подходе язык описания ограничений является полным расширением языка Python, т.е. любую программу на Python можно считать программой на нашем языке описания ограничений. Более того, это расширение реализовано непосредственно на самом языке Python и для пользователя выглядит как библиотека классов и функций.

Ограничения на компоненты могут быть размещены непосредственно в определениях компонентов. В языке AADL для этого есть такой механизм расширений, как annex. Ограничение представляется функцией на языке Python, помеченной декоратором. Проверяемая архитектурная модель доступна функции как объект, структура которого повторяет структуру модели. При программировании функций можно использовать известные навыки императивного, функционального и объектно-ориентированного программирования для языка Python. Проверка модели, снабженной ограничениями, выполняется достаточно просто – это выполнение всех функций, помеченных нужным декоратором.

При разработке языка описания ограничений и инструментов проверки модели возникают следующие задачи: какими конструкциями языка программирования следует пользоваться для описания ограничений, как интегрировать инструменты разработки на языке программирования в среду архитектурного моделирования. Далее рассмотрим эти задачи подробнее.

3. Интеграция языка Python в среду моделирования

От среды моделирования, в которой имеется поддержка языка описания ограничений, нужны такие возможности как удобное написание ограничений (с раскраской кода ограничений, с контекстными подсказками и автодополнением и т.п.), удобная проверка модели (скрывающая в себе технические особенности запуска компиляторов или интерпретаторов и показывающая результат запуска в виде, привычном при архитектурном моделировании), удобная отладка и т.п.

Среда MASIW Framework реализована на базе широко известной среды разработки Eclipse. Для получения удобного инструмента написания ограничений достаточно воспользоваться одним из плагинов для Eclipse для разработки на языке Python (например, плагином PyDev). Для проверки моделей готового плагина нет. От него требуется, чтобы он мог выполнить код ограничений над моделью, которой обладает среда моделирования, при помощи интерпретатора языка программирования. Поскольку этот интерпретатор не является частью среды моделирования, необходимо их специальным образом интегрировать. Эта интеграция может быть выполнена одним из трех следующих способов:

- вся проверка модели проводится средой моделирования, т.е. без использования отдельных компиляторов или интерпретаторов языка программирования; в этом подходе проверка модели встроена в среду моделирования, в частности, сама среда моделирования проводит синтаксический анализ и выполнение текстов ограничений, используя внутренние протоколы доступа к проверяемой модели; от языка программирования используется только синтаксис и семантика;
- вся проверка модели проводится вне среды моделирования; в этом подходе проверяемая модель отчуждается от среды моделирования в виде кода на языке программирования, на котором написаны ограничения; ограничения без изменений встраиваются в этот код; полученный код выполняется также вне среды моделирования существующими компиляторами и интерпретаторами, а результат передается в среду моделирования;
- среда моделирования работает параллельно с интерпретатором языка программирования и предоставляет части модели по запросу интерпретатора; предполагается наличие в среде моделирования модуля, выполняющего преобразование запросов интерпретатора в запросы к модели в среде моделирования по ее внутренним протоколам.

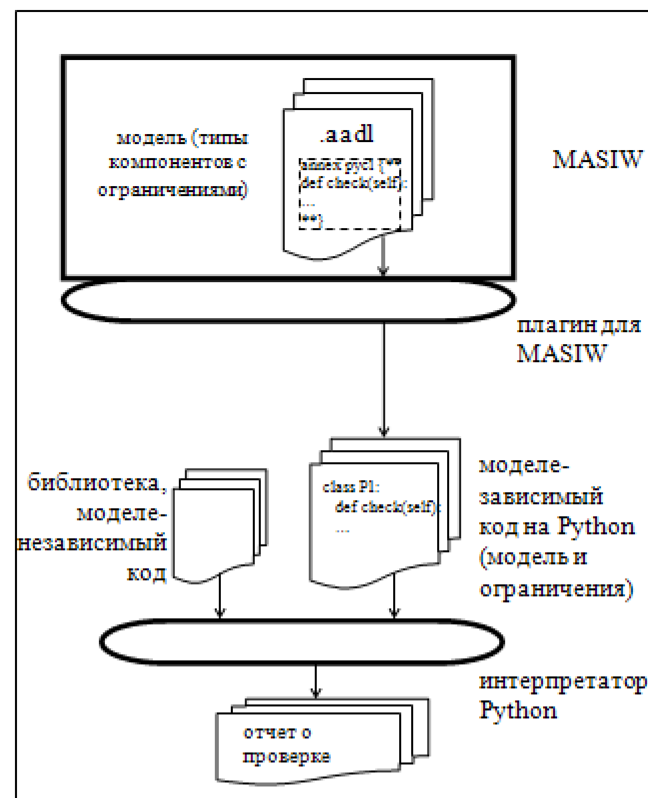

Рис. 1. Проверка модели в MASIW Framework.

Процесс проверки модели изображен на рис. 1. Ограничения пишутся внутри определений типов компонентов. Затем на основе этих определений специальный плагин для среды MASIW Framework генерирует набор моделе-

зависимых исходных текстов на языке Python. Эти тексты включают в себя код, повторяющий типы компонентов (сюда включены и ограничения, т.к. они находились в определениях типов компонентов), и генератор целевого проверяемого объекта (это объект одного из типов компонентов). Моделе-зависимый код соединяется с заранее подготовленным моделе-независимым кодом и библиотеками на Python, которые использованы в ограничениях. Полученный код выполняется обычным интерпретатором Python: этот код строит объект с проверяемой моделью и, итерируясь по его подкомпонентам, вызывает все функции-ограничения. В итоге получается отчет о сделанной проверке модели.

4. Представление проверяемой модели на языке программирования

Архитектурная модель на языке AADL состоит из определений типов компонентов. Этот тип описывает, что каждый компонент этого типа будет содержать определенные подкомпоненты и различные атрибуты. Ограничения зачастую формулируются для каждого компонента того или иного типа. Поэтому удобно включить ограничения в определения типа. Более того, один тип может наследовать другой тип (что означает включение всего, что описано в типе-предке в тип-потомок).

Все это приводит к выводу о родстве типов компонентов в AADL с классами в объектно-ориентированном программировании. Мы решили следовать этому и генерировать класс для каждого типа компонента. Атрибуты типа компонента становятся полями класса, функции-ограничения – методами класса.

Рассмотрим пример типа-процесса, в котором есть входной порт данных, и возможный код на языке Python для него:

```
process P1
features
    input: in data port FastPort;
…
end P1;
data FastPort
…
end FastPort;

class FastPort:
    …
class P1:
    def __init__(self):
        self.features = {}
        self.features["input"] = self.input =
    InDataPort(FastPort)
        ...
```

Класс P1 соответствует типу компонента-процесса P1. Метод __init__ класса P1 – это конструктор объектов класса P1. Он создает все поля объекта (в данном случае, инициализирует таблицу для feature, создает объект класса FastPort и помещает его в эту таблицу).

У разных классов будут повторяющиеся части. Поэтому их можно вынести в отдельные классы и сократить код генерируемых классов:

```
    class P1(ComponentTypeInstance, Process):
        def __init__(self):
            super(P1, self).__init__()
        def init_features(self):
            self.features["input"] = self.input =
    InDataPort(FastPort)
 …
```

В последнем случае метод init_features вызывается конструктором класса ComponentTypeInstance, который в свою очередь вызывается конструктором класса P1. Метод init_features переопределен в классе P1. В нем записана инициализация полей для features именно для типа P1.

Использования словарей для подкомпонентов, feature и т.п. позволяет использовать существующий синтаксис языка Python для итерирования по содержимому словаря (т.е. для итерирования по всем подкомпонентам, feature и т.п.). Для обращения к отдельным подкомпонентам, feature и т.п. заводятся дополнительные поля, имя которых совпадает с именем подкомпонента, feature и т.п. (в примере выше это input). Согласно стандарту языка AADL имена подкомпонентов, feature и т.п. должны различаться внутри одного и того же типа компонента и в них могут входить только такие символы, которые встречаются в идентификаторах в языках программирования.

Кроме полей из определений типов компонентов, для повышения удобства генерируются дополнительные поля и методы классов. Например, для соединений генерируются дополнительные поля, означающие концы соединения.

После того, как в классе сгенерированы конструкторы и поля из определения типа и удобные служебные поля и методы, в определение класса вставляется содержимое annex с ограничениями. Поскольку это содержимое уже написано на языке Python, то не требуется какая-либо сложная обработка и трансляция этого содержимого, чтобы им можно было пользоваться. Возможна лишь необходимость подстроить отступы в сгенерированном коде, чтобы он соответствовал отступам в содержимом из annex.

Чтобы приблизить язык Python и язык AADL, нужно решить еще одну задачу: Python – регистро-зависимый язык, а AADL – регистро-независимый, поэтому

необходимо сделать так, чтобы в коде ограничений можно было использовать произвольный регистр идентификаторов. Эта задача была решена, благодаря тому, что в Python можно управлять тем, как по имени поля получить объект-поле.

Наконец, поскольку цель ограничения – проверить модель, а не изменить ее, был реализован запрет изменения объектов, представляющих модель, из кода ограничений.

5. Описание ограничений на языке Python

Практика показывает, что среди различных требований очень востребованы требования внутренней согласованности компонентов того или иного типа. Компоненты могут включать подкомпоненты, сами компоненты и их подкомпоненты могут обладать свойствами и интерфейсными сущностями – они должны быть согласованы. Такое требование логично сопоставить с типом компонента и описать в виде функции («функции-ограничения»), которая получает ссылку на проверяемый объект соответствующего типа и возвращает булевское значение («истина», если требование выполнено для данного ей компонента, «ложь», иначе). Одним из способов сопоставления является включение функции в конструкцию определения типа компонента. Важно, что функция-ограничение может делать все то, что может делать программа на языке Python.

Для промежуточных вычислений тоже могут потребоваться функции, но эти функции не должны автоматически вызываться для проверки и не обязаны иметь указанную выше сигнатуру: нужен способ разграничения функций внутри определения одного и того же типа компонента, какие функции будут обозначать ограничения, а какие функции будут вспомогательными. В языке Python для этого есть такое средство, как декоратор. В частности, функция-ограничение – это будет функция с декоратором @constraint. Вспомогательная функция – это функция без этого декоратора.

Рассмотрим пример функции-ограничения check из типа P1, проверяющей следующее требование: «каждый компонент типа P1 должен иметь свойство Size и это свойство должно быть положительным числом». Посмотрите код этой функции (обратите внимание, что он записан на чистом Python):

```
annex pycl {**
…
  @constraint
  def check(self):
    return “Size” in self.properties and
self.properties[“Size”] > 0
…
**};
```


Обратите внимание на то, что функция-ограничение check имеет аргумент для ссылки на объект, представляющий проверяемый компонент. Альтернативный подход – это безаргументные функции-ограничения (как сделано в языке REAL). Такие функции самостоятельно выполняют итерирование по всей модели и выбор подходящих объектов для проверки. Наличие явного аргумента дает возможность получить более детальную статистику проверки модели, а именно иметь для каждого проверяемого компонента множество функций-ограничений, равных истине («выполненные требования») и множество функций-ограничений, равных лжи («невыполненные требования»). Такая информация позволяет упростить локализацию ошибки и ее исправление.

Декораторы могут сами иметь аргументы. Декоратор @constraint может иметь декоратор с текстовым описанием проверяемого ограничения, удобное для чтения человеком. Такое неформальное описание может быть включено в отчет о проверке или использовано для организации трассировки требований:

```
  @constraint(“Each P1 component must have a
Size property which must be positive”)
  def check(self):
    return “Size” in self.properties and
self.properties[“Size”] > 0
```

В следующем примере используется вспомогательная рекурсивная функция. Функция-ограничение может вызывать другие вспомогательные функции и функции-ограничения, что можно использовать для организации цепочек проверок:

```
  @constraint(“Each P1 component must have a
Size property which must be a power of 2”)
  def check(self):
       def ispower2(n):
            if n < 2 or n % 2 != 0:
                 return False
            else:
                 return ispower2(n/2)
    return “Size” in self.properties and
ispower2(self.properties[“Size”])
```

Функция на языке Python может иметь несколько декораторов одновременно. Для примера мы определили декоратор @precondition. Он позволяет определить условие, при невыполнении которого проверка ограничения не имеет смысла. Например, требование «свойство Size должно быть степенью двойки» не имеет смысла, если такого свойства нет у компонента:

```
  @precondition(lambda self: “Size” in
self.properties)
```

```
  @constraint("Each P1 component must have a
Size property which must be a power of 2")
  def check(self):
      def ispower2(n):
          if n < 2 or n % 2 != 0:
              return False
          else:
              return ispower2(n/2)
    return ispower2(self.properties["Size"])
```

Для отладки программ часто пользуются т.н. «отладочной печатью». Это дополнительные операторы печати некоторых фраз, по наличию которых при выполнении программы судят о выполнении тех или иных участков программы и о значениях переменных в момент их выполнения. Во время выполнения проверки модели отладочная печать возможна, но для ее использования при отладке необходимо ее структурировать с привязкой к контексту проверки, т.к. отладочная печать разных вызываемых функций-ограничений сливается в единый текст. Чтобы избежать этой необходимости, мы добавили метод info(). Он имеет 1 аргумент для некоторой строки. Она будет вставлена в отчет, если выполнение дойдет до вызова этого метода. Этот метод автоматически запоминает контекст, в котором он вызван:

```
  @precondition(lambda self: "Size" in
self.properties)
  @constraint("Each P1 component must have a
Size property which must be a power of 2")
  def check(self):
      def ispower2(n):
          if n < 2 or n % 2 != 0:
              return False
          else:
              return ispower2(n/2)
     if self.properties["Size"] <= 0:
          self.info("Value of a Size property
is not positive")
          return False
     if not ispower2(self.properties["Size"]):
          self.info("Value of a Size property
is not a power of 2")
          return False
     return True
```


6. Заключение

В данной статье обсуждается построение языка описания ограничений на базе языка программирования. Предложенный подход позволяет переиспользовать богатые возможности языка программирования и существующую инфраструктуру (среды разработки, компиляторы, интерпретаторы, отладчики, средства документирования). Кроме того, данный подход должен позволить понизить порог вхождения в язык описания ограничений, если этот язык построен на базе хорошо известных концепций и синтаксиса. Существующих возможностей расширения языков программирования достаточно, чтобы описывать ограничения в терминах модели без изменения среды моделирования. Наконец, этот подход позволяет минимизировать затраты на разработку инструментов выполнения и разработки ограничений.

В качестве примера в статье использовалась реализация этого подхода в среде моделирования MASIW Framework. Для нее был разработан плагин, позволяющий проверять, выполнены ли в модели ограничения, описанные на расширении языка программирования Python. В качестве основы для реализации плагина использовался широко известный плагин PyDev к среде Eclipse, позволяющий разрабатывать программы на языке Python в этой среде.

Список литературы

[1]. O. Gilles, J. Hugues. Expressing and enforcing user-defined constraints of AADL models. Proceedings of the 5th UML&AADL Workshop. 2010. PP. 337-342.

[2]. D. Cofer, A.Gacek, S.Miller, M.W. Whalen, B. LaValley, L. Sha. Compositional Verification of Architectural Models. NASA Formal Methods, Proceedings of the 4th International Symposium. 2012. PP. 126-140.

[3]. D. Albitskiy, A. Khoroshilov, I. Koverninskiy, M. Olshanskiy, A. Petrenko, A. Ugnenko. AADL-Based Toolset for IMA System Design and Integration. SAE International Journal of Aerospace. 2012, 5. PP. 294-299. doi:10.4271/2012-01-2146.

[4]. Д.В. Буздалов, С.В. Зеленов, Е.В. Корныхин, А.К. Петренко, А.В. Страх, А.А. Угненко, А.В. Хорошилов. Инструментальные средства проектирования систем интегрированной модульной авионики. Труды Института системного программирования РАН. 2014, т.26, вып. 1. с.201-230. doi:10.15514/ISPRAS-2014-26(1)-6.

Python-Based Constraint Language for Architecture Models

[2]E. Kornykhin <kornevgen@cs.msu.ru>

[1,2,3,4]A. Khoroshilov <khoroshilov@ispras.ru>

[1]Institute for System Programming of the RAS,

25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.

[2] Lomonosov Moscow State University,

GSP-1, Leninskie Gory, Moscow, 119991, Russia.

[3] *Moscow Institute of Physics and Technology (State University)*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] *National Research University Higher School of Economics (HSE)*
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. The paper presents an approach to specify constraints on AADL models in Python-based language and a toolset allowing to verify that constraints. The goal of the approach is to enable reusing of existing rich facilities of Python language, tools, and libraries as well as to reduce learning curve of engineers. Constraints must be placed into component annexes. These constraints must be written in Python programming language as functions with one argument (an object to be checked), Boolean result, and special decorator. A plugin for a modeling environment generates a program in Python from the model components declarations. While it is executing this program creates an object with the model instance and checks the object by functions from annexes. This approach is implemented in MASIW Framework that allows checking constraints on model instance. The implementation is made upon PyDev, a well-known Eclipse-plugin for Python developing in Eclipse and reuses integration of Eclipse with Python from PyDev. The Python sources generated are enough to involve automatically such PyDev tools as code coloring, auto-completion for classes (components in AADL), fields of classes (subcomponents, connections, features, etc. in AADL), methods of classes (constraints from annexes), hierarchy of packages and classes (according to components hierarchy in AADL).

Keywords: architecture modeling; constraints; Python; AADL; model verification.

DOI: 10.15514/ISPRAS-2015-27(5)-8

For citation: E. Kornykhin, A. Khoroshilov. Python-Based Constraint Language for Architecture Models. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 143-156 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-8.

References

[1]. O. Gilles, J. Hugues. Expressing and enforcing user-defined constraints of AADL models. Proceedings of the 5th UML&AADL Workshop. 2010. PP. 337-342.
[2]. D. Cofer, A.Gacek, S.Miller, M.W. Whalen, B. LaValley, L. Sha. Compositional Verification of Architectural Models. NASA Formal Methods, Proceedings of the 4th International Symposium. 2012. PP. 126-140.
[3]. D. Albitskiy, A. Khoroshilov, I. Koverninskiy, M. Olshanskiy, A. Petrenko, A. Ugnenko. AADL-Based Toolset for IMA System Design and Integration. SAE International Journal of Aerospace. 2012, 5. PP. 294-299. doi:10.4271/2012-01-2146.
[4]. D.V. Buzdalov, S.V. Zelenov, E.V. Kornykhin, A.K. Petrenko, A.V. Strakh,A.A. Ugnenko, A.V. Khoroshilov. Instrumental'nye sredstva proektirovanija sistem integrirovannoj modul'noj avioniki [Tools for System Design of Integrated ModularAvionics]. Trudy ISP RAN [The Proceedings of ISP RAS], 2014, vol. 26, n.1, pp. 201-230 (in Russian). doi: 10.15514/ISPRAS-2014-26(1)-6.