Как писать bug database.

На данном этапе база данных известных ошибок представляет собой xml-файл, созданный в соответствии со схемой, расположенной по этому адресу: http://unitesk.com/bugdb/bugDB.xsd

База состоит из списка багов, каждый из которых имеет:

- Строковой идентификатор. Он не должен быть пустой строкой. Запрещается наличие двух багов с одним и тем же идентификатором.
- Один или более паттернов, по которым ищется соответствие с ошибками из трассы
- Тело описания бага, в котором может находиться произвольный xml-текст.

Каждый паттерн имеет:

- Обязательный атрибут `origin` (см. раздел ***Происхождение ошибки***)
- Опциональный идентификатор языка, на котором пишется текст паттерна.
- Текст паттерна.

Сейчас в качестве языка написания паттернов поддерживается только JavaScript.

JavaScript usage rules & hints.

Текст паттерна это:

- Выражение.
  В этом случае результатом вычисления паттерна будет результат вычисления этого выражения. Тип JavaScript выражения – аналог `boolean` или `java.lang.Boolean`.
- Набор statement-ов.
  В этом случае результатом выражения является значение специальной JS переменной `result`. Тип данных, присвоенных в эту переменную должен быть такой же (см. выше).

Результат вычисления паттерна трактуется в соответствии с семантикой JavaScript:

- Значения `0`, `NaN`, `null`, `""` (пустая строка),. `undefined` эквивалентны `false` – ошибка не соответствует данному паттерну.
- Остальные значения эквивалентны `true` – ошибка соответствует данному паттерну

Доступ к атрибутам модели ошибки трассы делается через идентификатор их имени, без какой-либо квалификации. Например: `info.endsWith( ….. )`.

Так же в текте паттерна определены стандартные JavaScript объекты и классы (`Number`, `RegExp` и т.д.).

С точки зрения правил разрешения имён текст паттерна можно рассматривать аналогично Java-тексту некоторой функции (или выражения внутри функции) класса модели ошибок трассы. То есть помимо атрибутов без квалификации объектом доступны методы и константы класса модели ошибки. Например, в выражении «`kind == POSTCONDITION_FAILED`»

- `kind` является атрибутом ошибки,
- а `POSTCONDITION_FAILED` – константой класса модели ошибки.

Общее правило обращения с атрибутами и вообще написания паттерна таково: ***писать почти как на Java 2.***

Сравнение с Java

Аналогично Java:

- Всё case sensitive!
- Если в некотором java объекте есть доступный метод, его можно вызвать. При этом можно передать в качестве параметра JS-объект подходящего типа.
- Доступ к элементам list-ов и map-ов через стандартные функции get из Java API.
- Доступ к элементам массива через `[ ]`.
- В JS есть константа `null`.

Отличия от Java:

- Доступ к атрибуту `attr` объекта `obj` пишется не через getter (`obj.getAttr()`), а как к полю – `obj.attr` .
- В тексте паттерна **не надо** делать различение между примитивными типами Java и их обёртками из `java.lang`(`java.lang.Integer` и `int`, например).
  Как следствие атрибуты типа `java.lang.Boolean` можно использовать в логических выражения JS напрямую.
- Исключение из вышесказанного: если в JavaScript вызывается Java-метод с декларированным -параметром типа `boolean`, а передаётся `java.lang.Boolean`, то будет ошибка. Пример:
  Java:
  ```
  Boolean getB(){ …. }
  boolean negate( boolean v ){ …. }
  ```
  JavaScript:
  ```
  negate( b )
  ```
  Это “странность” спецификации связки JavaScript и Java.
  Workaround стандартный: `negate( b.booleanValue() )`
- **Не надо** приводить результаты выражений. Например для списка строк сработает вот это: `stringList.get( 0 ).startsWith( "abc" );`
- Переменные объявляются без типа, но с ключевым словом `var`. Пример: `var parameter = ...` .
- В контексте паттерна **не доступны** имена из Java пакета `java.lang`(про пакеты см. ниже).

Особенности нашей трактовки JavaScript

В целях большего контроля за корректностью текстов паттернов были устранены некоторые вольности, допустимые JavaScript, а именно:

- Доступ к неизвестному атрибуту Java-объекта ссылочного типа приводит к ошибке вычисления паттерна, а не к `undefined` в качестве результата выражения
- Аналогичное поведение и в случае установки несуществующего атрибута.

Именоо поэтому, если хочется сохранить в скрипте какое-нибудь значение, следует использовать явную декларацию переменной через конструкцию `var`.

Для примитивных типов описанные выше ограничения наложить не удалось: так что kind.X даст в результате `undefined`, - будьте внимательны.

Декларация переменной, имя которой совпадает с именем атрибута ошибки, приведёт к установке его значения, а не к появлению новой переменной, перекрывающей этот атрибут.

Так как интерпретатор JavaScript написан на Java, то нельзя получать доступ к /устанавливать значения элементов массива по индексам большим стандарного Java-овского значения length.

Small hints

Как всё это работает: при обращении из скрипта к java-объекту он «конвертируется» в java script аналог. Соответственно, при передачи в java-метод скриптового объекта происходит обратное «преобразование». Правила преобразования в полноте своей неизвестны, но вполне можно руководствоваться common sense.

Если хочется использовать какие-то классы из Java патектов, то обращение к java-классу делается так через имя `Packages` и полное имя класса: `Packages.java.util.regex.Matcher`.

Пример использования доступа к пакетам и классам для трассировки вычисления несрабатывающего паттерна:

1. переписать паттерн из вида выражения в вид набора statement-ов (не забывая присваивать в конце в переменную `result`).
2. написать в начале паттерна: `var out = Packages.java.lang.System.out;`
3. пользоваться `out.println( <результаты промежуточных вычислений );`

В JavaScript объекты Java-классов пока предлагается не создавать, - могут быть тонкости.

JavaScript конструкци `for … in …` не работает для коллекций. Только для массивов.

Если что, можно смотреть на документацию и тексты о JavaScript-е вообще, а так же на документацию по движку: http://www.mozilla.org/rhino/doc.html . Там есть пара tutorial-ов, можно посмотреть бегло.

Пример

Трасса

```
<model_operation_start trace="1" kind="stimulus" package="util.search"
    signature="void insque_spec( struct ThreadId context, struct VoidTPtr element, struct
VoidTPtr pred )"
    refid="1" />
<model_value trace="1" kind="argument" type="struct ThreadId"
    name="context">
    <![CDATA[struct { 0, 1404, 3086919360 }]]>
</model_value>
<model_value trace="1" kind="argument" type="struct VoidTPtr"
    name="element">
    <![CDATA[struct { 0, 1404, 135634344 }]]>
</model_value>
<model_value trace="1" kind="argument" type="struct VoidTPtr"
    name="pred">
    <![CDATA[struct { 0, 0, 0 }]]>
</model_value>
<oracle_start trace="1" package="util.search"
    signature="void insque_spec( struct ThreadId context, struct VoidTPtr element, struct
VoidTPtr pred )" />
<precondition_end trace="1" />
<coverage_element trace="1" coverage="C" id="1" />
<exception trace="1" internal="false">
    <where></where>
    <info>
        <![CDATA[Mediator failed]]>
    </info>
</exception>
<info trace="1">
    <![CDATA[Segmentation fault: Invalid memory address '(nil)']]>
</info>
<oracle_end trace="1" />
```

```
<model_operation_end trace="1" />
```

База ошибок:

```
<?xml version="1.0" encoding="UTF-8"?>
<bugDB name="myBase"
       xmlns="http://unitesk.com/bugdb"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://unitesk.com/bugdb/bugDB.xsd"
>
  <bug id="BUG_1">
    <pattern origin="util.search.insque_spec">
      <![CDATA[
          kind == MEDIATOR_FAILED
       && info.endsWith( "Segmentation fault: Invalid memory address \'(nil)\'" )
       &&     modelOperation.signature
          == "util.search.search.void insque_spec( struct ThreadId context, struct VoidTPtr
element, struct VoidTPtr pred )"
       &&     modelOperation.getParameterValue( "pred" )
          == "struct { 0, 0, 0 }"
       && getCoveredElement( "C" ) == 1
      ]]>
    </pattern>
    <body> .... </body>
  </bug>
</bugDB>
```

Описание модели ошибки

Info

Сообщение, сопровождающее ошибку.

Трасса

```
<exception trace="1" internal="false">
  <where></where>
  <info><![CDATA[Postcondition failed]]></info>
</exception>
<info trace="1"><![CDATA[Requirement failed: {pthread_attr_setstacksize.04.01} error not
met because its predicate (pattr==((void *)0)) is false while error code EINVAL is set ;
]]></info>
```

Извлечение атрибута

```
info ==
"Postcondition failed
Requirement failed: {pthread_attr_setstacksize.04.01} error not met because its predicate
(pattr==((void *)0)) is false while error code EINVAL is set ; "
```

Kind

Kind – атрибут, вычисляемый на основе атрибута Info.

Для CTesK определяется следующее соответствие между значением атрибута Kind и шаблоном атрибута Info.

Info pattern	Kind	Comment
Postcondition failed	POSTCONDITION_FAILED	
Serialization failed	SERIALIZATION_FAILED	
Scenario function	SCENARIO_FUNCTION_FAILED	М.б. назвать как-нибудь по другому для

failed		унификации с JavaTesK
Mediator failed	MEDIATOR_FAILED	
Invariant failed	INVARIANT_FAILED	
Precondition failed	PRECONDITION_FAILED	
Incorrect set of interactions	INCORRECT_SET_OF_INTERACTIONS	
No back arc	UNCONNECTED_GRAPH	dfsm only
Deterministic path to the incompletely tested states not found	UNCONNECTED_GRAPH	ndfsm only
Nondeterministic behavior	NONDETERMINISTIC_GRAPH	dfsm only
Scenario initialization failed	SCENARIO_INITIALIZATION_FAILED	
is_stationary field should be initialized if deferred reactions are enabled	INVALID_SCENARIO	
save_model_state field should be initialized if deferred reactions are enabled	INVALID_SCENARIO	
restore_model_state field should be initialized if deferred reactions are enabled	INVALID_SCENARIO	
Non-deterministic serialization	NONDETERMINISTIC_MODEL	
Series has been finished in a nonstationary state	NONSTATIONARY_STATE	Interim only
Если атрибут internal="true" Structural failure Если kind неизвестен	INTERNAL_ERROR STRUCTURAL_FAILURE UNKNOWN_KIND	

Извлечение атрибута

`kind == POSTCONDITION_FAILED`

Для трассы из описания атрибута Info значением атрибута Kind будет `POSTCONDITION_FAILED`

Exception (Optional)

Stacktrace исключения, вызвавшего ошибку.

Атрибут определен, если ошибка порождена исключением. Для CTesK всегда не определен.

Трасса

```
<exception trace="1" internal="false">
  <where>example.ExampleSpecification.throwUnexpectedException( int )</where>
  <info>Unexpected exception in implementation</info>
  <backtrace><![CDATA[java.lang.NullPointerException
  at example.Example.throwException(Example.java:31)
```

```
  at
example.ObjectMediator_ExampleMediator._ts_implementation_throwUnexpectedException(Unknown
Source)
  at example.ObjectModel_ExampleSpecification.throwUnexpectedException(Unknown Source)
  at example.Oracle_ExampleSpecification.throwUnexpectedException(Unknown Source)
  at example.ObjectModel_ExampleSpecification.throwUnexpectedException(Unknown Source)
  at example.ScenarioDriver_ExampleScenario._ts_call_throwUnexpectedException(Unknown
Source)
  at example.ScenarioDriver_ExampleScenario.call(Unknown Source)
  at jatva.lang.TestEngine.perform_call(TestEngine.java:183)
  at
jatva.engines.DeterministicFSMWithReset_TestEngine.irun(DeterministicFSMWithReset_TestEngin
e.java:133)
  at jatva.lang.TestingModel.run(TestingModel.java:221)
]]></backtrace>
</exception>
```

Извлечение атрибута

```
exception ==
"java.lang.NullPointerException
  at example.Example.throwException(Example.java:31)
  at
example.ObjectMediator_ExampleMediator._ts_implementation_throwUnexpectedException(Unknown
Source)
  at example.ObjectModel_ExampleSpecification.throwUnexpectedException(Unknown Source)
  at example.Oracle_ExampleSpecification.throwUnexpectedException(Unknown Source)
  at example.ObjectModel_ExampleSpecification.throwUnexpectedException(Unknown Source)
  at example.ScenarioDriver_ExampleScenario._ts_call_throwUnexpectedException(Unknown Source)
  at example.ScenarioDriver_ExampleScenario.call(Unknown Source)
  at jatva.lang.TestEngine.perform_call(TestEngine.java:183)
  at
jatva.engines.DeterministicFSMWithReset_TestEngine.irun(DeterministicFSMWithReset_TestEngine.j
ava:133)
  at jatva.lang.TestingModel.run(TestingModel.java:221)"
```

в случае если атрибут не определен, `exception == null`

Scenario (optional)

Название сценария.
Атрибут определен, если ошибка произошла в сценарии. Сейчас и далее вложенность рассматривается рекурсивно, так если ошибка произошла внутри состояния, находящегося внутри сценария, то считается, что ошибка так же произошла и в сценарии.

Трасса

```
<scenario_start trace="1" name="attr_scenario" time="1159825239000" host="dolgopa"
os="Linux 2.6.13-15-default"/>
…
  <exception trace="1" internal="false">
```

Извлечение атрибута

```
scenarioName == "attr_scenario"

Если атрибут не определен, scenarioName == null.
```

State (optiona)

Имя состояния.
Атрибут определен, если ошибка произошла внутри состояния.

Трасса

```
<state trace="1" id="0"/>
<scenario_value trace="1" kind="state" type="" name=""><![CDATA[NULL]]></scenario_value>
…
  <exception trace="1" internal="false">
```

Извлечение атрибута

`stateName == “NULL”`

если атрибут не определен, stateName == null.

Transition; scenario method (optional)

Имя вызванного сценарного метода.
Атрибут определен, если ошибка произошла внутри перехода.

Трасса

```
<transition_start trace="1" id="0"/>
<scenario_value trace="1" kind="scenario method" type="" name=""><!
[CDATA[exec_scen]]></scenario_value>
...
  <exception trace="1" internal="false">
```

Извлечение атрибута

`transitionName == “exec_scen”`

если атрибут неопределен, transitionName == null

Transition; iteration variables (optional)

Итерационные переменные и их значения.

Трасса

```
<transition_start trace="1" id="0"/>
  <scenario_value trace="1" kind="scenario method" type="" name=""><!
[CDATA[exec_scen]]></scenario_value>
  <scenario_value trace="1" kind="iteration variable" type="int" name="i"><!
[CDATA[0]]></scenario_value>
  <scenario_value trace="1" kind="iteration variable" type="int" name="j"><!
[CDATA[1]]></scenario_value>
  <scenario_value trace="1" kind="iteration variable" type="int" name="k"><!
[CDATA[1]]></scenario_value>
  <scenario_value trace="1" kind="iteration variable" type="int" name="l"><!
[CDATA[1]]></scenario_value>
...
  <exception trace="1" internal="false">
```

Извлечение атрибута

`transitionIterationVariables ==`
Список итерационных переменных List { ‘i’, ‘j’, ‘k’, ‘l’ }

```
getTransitionIterationVariableValue(‘j’) == ‘1’
getTransitionIterationVariableValue(‘q’) == null
```

в случае если ошибка произошла вне перехода transitionIterationVariables == null

Invariant (Optional)

Сигнатура нарушенного инварианта.
Атрибут определен, если ошибка вызвана нарушением инварианта.

Трасса

```
<invariant_start trace="0" signature="Project4.Specification1.i"
id="Project4.Specification1@1"/>
  <exception trace="0" internal="false">
    <where><![CDATA[class=Project4.Specification1
invariant=i
]]></where>
    <info>Invariant violation</info>
  </exception>
<invariant_end trace="0"/>
```

Извлечение атрибута

`violatedInvariantSignature == "Project4.Specification1.i"`

если атрибут не определен, `violatedInvariantSignature`

Model operation; signature (optional)

Сигнатура модельного вызова.
Атрибут определен, если ошибка произошла внутри модельной операции.

Трасса

```
<model_operation_start trace="1" kind="stimulus" package="fs.meta" signature="int
__lxstat_spec( struct ThreadId context, CString *path, FileStatus *buff, ErrorCode
*errno )"/>
...
  <exception trace="1" internal="false">
```

Извлечение атрибута

```
modelOperation.signature ==
"int __lxstat_spec( struct ThreadId context, CString *path, FileStatus *buff, ErrorCode *errno
)"
```

если атрибут не определен, modelOperation == null

Model operation; name (optional)

Короткое имя модельной операции.
Атрибут определен, если ошибка произошла внутри модельной операции.

Трасса

```
<model_operation_start trace="1" kind="stimulus" package="fs.meta" signature="int
__lxstat_spec( struct ThreadId context, CString *path, FileStatus *buff, ErrorCode
*errno )"/>
...
  <exception trace="1" internal="false">
```

Извлечение атрибута

`modelOperation.name == "__lxstat_spec"`

Model operation; package (optional)

Пакет (подсистема) модельной операции.
Атрибут определен, если ошибка произошла внутри модельной операции.

Трасса

```
<model_operation_start trace="1" kind="stimulus" package=”fs.meta” signature="int
__lxstat_spec( struct ThreadId context, CString *path, FileStatus *buff, ErrorCode
*errno )"/>
...
  <exception trace="1" internal="false">
```

Извлечение атрибута

```
modelOperation["package"] == "fs.meta"
```

Model operation; model (optional)

Модельный объект, класс или функция.

Трасса

```
<model_object trace="0" id="1dff3a2">jatva.examples.pqueue.PQueueMediator@1dff3a2:{ items =
jatva.examples.pqueue.List@171bbc9:{ _size = 0, _head = null }, priorities =
jatva.examples.pqueue.List@1feca64:{ _size = 0, _head = null } }</model_object>
<model_operation_start trace="0" kind="stimulus" signature="public boolean
jatva.examples.pqueue.QueueSpecification.isEmpty()"/>
...
  <exception trace="1" internal="false">
```

Извлечение атрибута

```
modelOperation.targetKind ==
Значение выбирается из перечисления ( modelOperation.OBJECT | modelOperation.CLASS |
modelOperation.FUNCTION | modelOperation.UNKNOWN_TARGET )
modelOperation.OBJECT

modelOperation.target
=
“jatva.examples.pqueue.PQueueMediator@1dff3a2:{ items = jatva.examples.pqueue.List@171bbc9:
{ _size = 0, _head = null }, priorities = jatva.examples.pqueue.List@1feca64:{ _size = 0,
_head = null } }”

если ошибка произошла вне модельной операции, modelOperation == null
если в трассе нет информации о модели, modelOperation.target == null и
modelOperation.targetKind = modelOperation.UNKNOWN_TARGET
```

Model operation; parameters values (optional)

Параметры вызова модельной операции и их значения.

Трасса

```
<model_operation_start trace="1" kind="stimulus" package=”io.term” signature="int
getlogin_r_spec( struct ThreadId context, CString *name, unsigned long long namesize )"
refid="2"/>
  <model_value trace="1" kind="argument" type="struct ThreadId" name="context"><!
[CDATA[struct { 0, 13560, 3084523200 }]]></model_value>
  <model_value trace="1" kind="argument" type="CString *" name="@name"><!
[CDATA[]]></model_value>
```

```
  <model_value trace="1" kind="argument" type="unsigned long long" name="namesize"><!
[CDATA[1024]]></model_value>
...
     <exception trace="1" internal="false">
...
  <model_value trace="1" kind="argument" type="CString *" name="name"><!
[CDATA[]]></model_value>
  <model_value trace="1" kind="result" type="int"><![CDATA[25]]></model_value>
<model_operation_end trace="1"/>
```

Извлечение атрибута

```
modelOperation.parameters==
список имен параметров вызова модельной операции List {“context ”, “@name ”, “namesize”,
“name”}

modelOperation.getParameterValue(“unknown parameter”) == null

если ошибка произошла вне модельной операции, modelOperation == null
```

Model operation; result (Optional)

Значение, которое вернула модельная операция.

Атрибут определен, если ошибка произошла внутри модельной операции и модельная операция вернула значение. Не может быть определен одновременно с “model operation; exception”.

Трасса

```
<model_operation_start trace="1" kind="stimulus" package=”io.term” signature="int
getlogin_r_spec( struct ThreadId context, CString *name, unsigned long long namesize )"
refid="2"/>
  <model_value trace="1" kind="argument" type="struct ThreadId" name="context"><!
[CDATA[struct { 0, 13560, 3084523200 }]]></model_value>
  <model_value trace="1" kind="argument" type="CString *" name="@name"><!
[CDATA[]]></model_value>
  <model_value trace="1" kind="argument" type="unsigned long long" name="namesize"><!
[CDATA[1024]]></model_value>
...
     <exception trace="1" internal="false">
...
  <model_value trace="1" kind="argument" type="CString *" name="name"><!
[CDATA[]]></model_value>
  <model_value trace="1" kind="result" type="int"><![CDATA[25]]></model_value>
<model_operation_end trace="1"/>
```

Извлечение атрибута

```
modelOperation.returnValue == “25”

если ошибка произошла вне модельной операции, modelOperation == null
если атрибут не определен, modelOperation.returnValue == null
```

Model operation; exception (optional)

Исключение сгенерированное модельной операцией.

Атрибут определен, если ошибка произошла внутри модельной операции и модельная операция сгенерировала исключение. ?Неопределен для CTesK? Не может быть определен одновременно с “model operation; result”.

Trace

```
<model_operation_start trace="1"
signature="example.ExampleSpecification.throwExpectedException( int )"/>
...
     <exception trace="1" internal="false">
...
  <model_value trace="1" kind="exception" type="java.lang.NullPointerException"
value="java.lang.NullPointerException"/>
<oracle_end trace="1"/>
```

Извлечение атрибута

```
modelOperation.exception == "java.lang.NullPointerException"

если ошибка произошла вне модельной операции, modelOperation == null
если атрибут неопределен, modelOperation.exception == null
```

Covered elements (Optional)

Название определенных критериев покрытия и покрытых элементов.

Атрибут определен, если ошибка произошла внутри модельной операции и для модельной операции имеется информация о структуре покрытия.

Трасса

```
      <model_operation_start trace="1" kind="stimulus" package="io.term" signature="int
getlogin_r_spec( struct ThreadId context, CString *name, unsigned long long namesize )"
refid="2"/>
...
        <oracle_start trace="1" package="io.term" signature="int getlogin_r_spec( struct
ThreadId context, CString *name, unsigned long long namesize )"/>
          <coverage_structure trace="1">
            <coverage id="C">
             <element id="0" name="A"/>
             <element id="1" name="B"/>
             ...
            </coverage>
            <coverage id="C2">
             ...
            </coverage>
          </coverage_structure>
...
          <coverage_element trace="1" coverage="C" id="1"/>
...
          <exception trace="1" internal="false">
```

Извлечение атрибута

```
coverages == { "C", "C2" }
Список (List) названий элементов покрытия

getCoveredElementId("C") == 1
getCoveredElementId("C2") == -1
getCoveredElementId("absent coverage") == -1

getCoveredElementName("C") == "B"
getCoveredElementName("C2") == null
getCoveredElementName("absent coverage") == null
```

если атрибут не определен, `coverages == null`

Preformulas & postformulas (Optional)

Prime формулы и их пре (до вызова реализации) и пост (после вызова реализации) значения

Трасса

```
<model_operation_start trace="0" kind="stimulus" signature="public boolean
jatva.examples.pqueue.QueueSpecification.isEmpty()"/>
  <oracle_start trace="0" signature="public boolean
jatva.examples.pqueue.QueueSpecification.isEmpty()"/>
    <coverage_structure trace="0">
      <formulae>
         <formula id="0">items.isEmpty()</formula>
         <formula id="1">isEmpty</formula>
      </formulae>
...
    </coverage_structure>
...
     <prime_formula trace="0" id="1" value="true"/>
     <precondition_end trace="0"/>
     <branch trace="0" name="Single"/>
...
     <prime_formula trace="0" id="0" value="false"/>
...
  <oracle_end trace="0"/>
  <model_value trace="0" kind="result" type="boolean" value="true"/>
<model_operation_end trace="0"/>
```

Извлечение атрибута

`primeFormulae` ==
Список List имеющихся формул { “items.isEmpty()”, “isEmpty”}

```
getPrimeFormulaPreValue(“items.isEmpty()”) == true (Boolean.TRUE)

getPrimeFormulaPreValue(“isEmpty”) == null

getPrimeFormulaPostValue(“items.isEmpty()”) == null

getPrimeFormulaPostValue(“isEmpty”) == false (Boolean.FALSE)
```

если ошибка произошла вне модельной операции или отсутствует информация о prime формулах, primeFormulae == null

Marks (Optional)

Определение марок, покрытых до вызова реализации (сброса информации о покрытом элементе). В тестах CTESK не используется.

Атрибут определен, если ошибка произошла внутри модельной операции.

Трасса

```
<model_operation_start trace="1" kind="stimulus" package=”io.term” signature="CString
*getlogin_spec( struct ThreadId context, ErrorCode *errno )" refid="1"/>
...
 <mark trace="1" name="getlogin.01"/>
...
 <coverage_element trace="1" coverage="C" id="1"/>
...
 <exception trace="1" internal="false">
```

Извлечение атрибута

```
markCovered(“getlogin.01”) == true (Boolean.TRUE)
markCovered(“getlogin.5678”) == false (Boolean.FALSE)
```

Postmarks (Optional)

Список марок, покрытых после вызова реализации (сброса информации о покрытом элементе). Список не пуст, если в трассе присутствует информация о покрытых марках.

Атрибут определен, если ошибка произошла внутри модельной операции.

Трасса

```
<model_operation_start trace="1" kind="stimulus" package=”io.term” signature="CString
*getlogin_spec( struct ThreadId context, ErrorCode *errno )" refid="1"/>
...
 <coverage_element trace="1" coverage="C" id="1"/>
...
 <mark trace="1" name="getlogin.04"/>
...
 <exception trace="1" internal="false">
```

Извлечение атрибута

```
coveredPostmarks ==
Список List покрытых postmarks { “getlogin.04” }
Если атрибут неопределен, coveredPostmarks == null

postMarkCovered(“getlogin.04”) == true (Boolean.TRUE)
postMarkCovered(“getlogin.5678”) == false (Boolean.FALSE)
```

Violated requirements

Проверка, нарушено ли заданное требование.

Трасса

```
<exception trace="1" internal="false">
  <where></where>
  <property name="req_id.mkdir.12.01" value="mkdir.12.01"/>
  <property name="req_id.mkdir.12.02" value="mkdir.12.02"/>
  <info><![CDATA[Requirement failed: {mkdir.12.01;mkdir.12.02} Function returned
EOK while EACCES was expected by condition isEACCES_dir_mkrm(context, pre_fs,
abspath)!=False_Bool3]]></info>
</exception>
```

Извлечение атрибута

```
isRequirementFailed("mkdir.12.01") == true
isRequirementFailed("unknown_req") == false
```

Примечание: Генератор отчётов поддерживает и старый формат трассы (со списком нарушенных требований в тексте `<info>`), и новый (с соответствующими `<property>`), поэтому функция `isRequirementFailed()` возвращает `true`, если имя соответствующего требования встречается хотя бы в одном из этих мест.

mplementation signature, Implementation object, Implementation parameters, Implementation exception (Optional)

То же что и для модельной операции, но в контексте реализационной операции. Для CTesK не определены.

Извлечение атрибута пока не реализовано

Custom tags

Таблица дополнительных атрибутов ошибки

Трасса

```
<exception trace="0" internal="false">
  <info>Postcondition violation</info>
  <property name="object" value="jatva.examples.pqueue.PQueueMediator@1dff3a2"/>
  <property name="method" value="public void
jatva.examples.pqueue.PQueueSpecification.enq(java.lang.Object)"/>
</exception>
```

Извлечение атрибута

```
properties ==
список List имен custom свойств { “object”,  “method”}

getPropertyValue(“object”) == jatva.examples.pqueue.PQueueMediator@1dff3a2

getPropertyValue(“unknown property”) == null
```

Набор модельных вызовов (Optional, only for kind SERIALIZATION_FAILED)

Атрибут определен, если ошибка имеет SERIALIZATION_FAILED kind.

Набор модельных вызовов, для которых не удалось успешно построить сериализацию.

Трасса

```
<model_operation_start trace="1" kind="stimulus" package=”process.process”
signature="void
...
<model_operation_end trace="1"/>
<model_operation_start trace="1" kind="reaction" package=”process.process”
signature="ExecReturnType *execNewProcess_return( void )" refid="2"/>
...
<model_operation_end trace="1"/>
<serialization_start trace="1"/>
...
<serialization_end trace="1"/>
<exception trace="1" internal="false">
  <where></where>
  <info><![CDATA[Serialization failed]]></info>
</exception>
```

Извлечение атрибута

```
modelOperationSeries ==
```

список List вызванных модельных методов; доступ к модельным методам аналогичен доступу к модельному методу ошибки, например, если modelOperation – элемент списка, то получить его сигнатуру можно обратившись к атрибуту modelOperation.signature

Если атрибут не определен, modelOperationSeries == null

Набор промежуточных (interim) ошибок (Optional, only for kind SERIALIZATION_FAILED)

Атрибут определен, если ошибка имеет SERIALIZATION_FAILED kind.
Набор промежуточных ошибок.

Трасса

```
<model_operation_start trace="1" kind="stimulus" package=”process.process” signature="void
...
<model_operation_end trace="1"/>
<serialization_start trace="1"/>
...
<oracle_start trace="1" package=”process.process” signature="ExecReturnType
*execNewProcess_return( void )" ref="2"/>
...
     <exception trace="1" internal="false" interim="true">
       <where></where>
       <info><![CDATA[Postcondition failed]]></info>
      </exception>
      <info trace="1"> <![CDATA[Requirement failed: {execl.32} The environment shall be
taken from the environ in the calling process]]></info>
...
<serialization_end trace="1"/>
<exception trace="1" internal="false">
  <where></where>
  <info><![CDATA[Serialization failed]]></info>
</exception>
```

Извлечение атрибута

interimFailures ==
список List промежуточных ошибок, доступ к каждой из них аналогичен доступу к ошибке описанному выше, промежуточные ошибки не могут иметь kind == SERIALIZATION_FAILED

Если атрибут не определен, interimFailures == null

Происхождение ошибки

Специальный атрибут, позволяющий предварительно (до вычисления pattern’ов в Bug Knowledge Base) сопоставить ряд известных bug’ов из Knowledge Base текущей ошибке.

Данный атрибут представляет собой список строк и для CTesK вычисляется следующим образом.

1. Если modelOperation != null, то origins == список List из одной строки <подсистема_модельного_метода>.<имя_модельного_метода>, если подсистема отсутствует, то используется специальное обозначение “~default package~”. Например, если ошибка произошла в модельном методе main из подсистемы main, то атрибут будет равен {“main.main”}
2. Иначе, если interimFailures != null, то origins == список List из атрибутов interimFailure.origins.get(0) для каждой ошибки из списка interimFailures. Например если к ошибке относятся две промежуточные (interim) failure с origins { “main.main” } и { “main.f” }, то атрибут будет равен {“main.main”, “main.f”}
3. Иначе, если modelOperationSeries != null, то origins == список из строк, каждая из которых вычисляется на основании соответствующего элемента списка modelOperationSeries и имеет следующий вид <подсистема_модельного_метода>.<имя_модельного_метода>, если подсистема отсутствует, то используется специальное обозначение “~default package~” Например, если ошибка относится к нарушению серии, в которой были вызваны два модельных

метода: main без подсистемы и f без подсистемы, то значение атрибута будет равным {"~default package~.main", "~default package~.f"}

4. Иначе, origins == { UNKNOWN_ORIGIN }

Информация об окружении

В трассе помимо событий, произошедших во время выполнения теста, также присутствует информация о системном окружении, в котором запускался тест.

Эта информация выводится в трассу в самом её начале, сразу же после старта трассировки. По умолчанию CTESK сбрасывает следующие свойства окружения:

Трасса

```
<environment trace="1" name="Host" value="lfdev-power64.linux-foundation.org"/>
<environment trace="1" name="Operating System" value="Linux 2.6.18-5-powerpc64"/>
<environment trace="1" name="Product Build" value="20071002"/>
<environment trace="1" name="Product Name" value="CTesK"/>
<environment trace="1" name="Product Version" value="2.4.0.209"/>
```

Извлечение атрибута

Каждый failure имеет атрибут `env`. У всех failure из одного тестового потока значение этого атрибута и его содержимое совпадают.

```
env.getProperty( "Operating System" ).indexOf( "powerpc64" ) != -1
```

Сброс дополнительной информации об окружении

Сбросить дополнительную информацию об окружении можно при помощи функции `void setTraceUserEnv(const char *name, const char *value)`. Её нужно вызывать в самом начале функции `main`. Например:

```
setTraceUserEnv( "distribution", "debian" );
```