# PyIG: Automated Input Generation for Python

Gary Wilson Jr. and Nandita Raman
Electrical & Computer Engineering
The University of Texas
Austin, TX, USA
{gary.wilson,nandita.raman88}@gmail.com

## ABSTRACT

Automated input generation is important in the software field because it reduces the cost of testing. In this paper, we introduce *P*ython *I*nput *G*enerator (PyIG). PyIG is our implementation of an automated input generation tool for the Python language, and to the best of our knowledge is the first tool of its kind for Python. Our implementation is based on the workings of Korat and, similarly, uses instrumentation and recording of field accesses to guide its pruning of input space. We demonstrate a data structure example from the Korat source code, as well as two non-data-structure examples that showcase PyIG's general applicability. For automated input generation and test execution, we provide a custom extension to Python's standard unit testing library. We also present the technique our tool uses to increase performance through the use of multiple execution processes.

## 1. INTRODUCTION

Software testing is both costly, and costly to ignore [8]. Automated input generation reduces the cost of testing because it helps relieve developers from the burden of writing tests, a burden involving tedious, repetitious tasks that are all too easy for developers to ignore. Though it has been shown that the earlier bugs are found, the least costly they are to fix [8], in this paper we focus on reducing the cost of testing during the development and testing lifecycle stages.

Most modern programming languages today have libraries to facilitate unit testing[1], or automated testing of a software system's individual components. While test execution and reporting is well automated by these unit testing libraries, the actual construction of input data and writing of tests is typically still very much a manual process.

In this paper we introduce ***Py***thon ***I***nput ***G***enerator (PyIG), a tool that automates the creation of input data, including complex data structures, structured documents, and other scenarios that are able to map to a sequence or data structure (e.g. a series of function or method calls). This additional automation – hooked into the automated unit test library – removes the tedious input-parameter creation work from the developer, saving time while also providing greater test coverage of the system under test (SUT).

Our implementation language of choice for the tool presented in this paper was Python[2], an interpreted, dynamic-typed, high-level language that allows for rapid development and prototyping. While Python's dynamic type system makes for quicker development and concise, readable code, it also presents a challenge for input generation implementations. Since variables and function parameters can vary in type (and can even change during runtime), the possible input space becomes exponentially larger.

For example, a function that adds two parameters together (see Listing 1) can take several different types of inputs, including `strings`, `ints`, `longs`, `floats`, `lists`, and many more – including custom types. In these scenarios, where multiple input types are supported by a function, the search space is essentially increased by a whole new dimension, adding an exponentially increasing number of combinations to the input search space.

```python
def add(a, b):
    return a + b
```

**Listing 1: A simple example of an `add` function in Python that can take several different input types.**

## 2. RELATED WORK

Korat [1] is a tool that generates non-isomorphic test cases up to a finite bound. Type declaration of the data structure, finitization, and a predicate (e.g. a `RepOk()` method) are the required inputs to Korat. `RepOk` checks the consistency of the representation of the data structure. Finitization is a set of bounds that limits the input size, specifying the number of objects in each class. Two test cases are said to be isomorphic if parts of their object graphs reachable from the root object are isomorphic. With the finitization and predicate defined, Korat generates all non-isomorphic test cases that satisfy the input predicate.

[1]E.g. Java has JUnit (`http://www.junit.org/`), C# has NUnit (`http://www.nunit.org/`, Python has PyUnit (`http://docs.python.org/library/unittest.html`)
[2]Python website: `http://www.python.org/`

Backtracking is a key feature of Korat that expedites exploration of the bounded input space of finitized values. Korat works efficiently by monitoring the execution of the `RepOk` predicate and backtracking only on fields accessed by `RepOk`. Korat prunes its state space depending on the evaluation of the input predicate, i.e. if the method that checks the input predicate returns false, then test case generation is not necessary. Optionally, Korat uses a method pre-condition to check the constraints/assumptions that the method makes on the input, and a method post-condition to check the correctness of the method's output.

Model checking is a technique for verifying finite state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not.

Java Path Finder (JPF) [9] is an explicit state model checker for Java Byte Code developed by NASA. Even if the state space shrinks, JPF observes more about the program execution and finds more defects than normal tests. JPF uses partial order, symmetry reduction, slicing, abstraction, and runtime analysis techniques to reduce the state space. Symbolic execution is an explicit state model checking technique. Symbolic execution is useful for software testing because it can analyze if and when errors in the code may occur. It can be used to predict how code statements affect specified inputs and outputs, and is important for path traversal. As with automated input generation (e.g. Korat), symbolic execution cannot handle large data.

To the best of our knowledge, PyIG is the first automated input generation tool for the Python language. Hackner and Memon [3] made use of a Python script to analyze GUI test cases (written in jfcUnit) and output a set of paths for accessing a target GUI component. Swain and Scott [7] also made use of Python scripts for test case execution and post-processing. However, their Python scripts, which handled assignment of runtime values for use as parameters on the command line, were themselves generated from a saved file containing test cases generated by a Java-based tool. Ian, Manolis, and Thierry [2] made use of a few tools for case studies on model-based testing, including a tool named Rule-Based Python (RBP). While this tool was used for automated input generation, its similarity to our tool cannot be determined because the literature includes little detail about RBP and implies that it is a tool used internally within the IBM corporation.

## 3. IMPLEMENTATION

Here, we present the implementation details of PyIG, our automated input generation tool written in and for the Python language. The following sections describe the primary components of our tool, show these components in action, and describe a few of the tool's primary features.

### 3.1 Components

Below is an overview of the classes used in PyIG and their functions:

- `FieldDescriptor` - Performs the instrumentation for finitized fields, using Python's descriptor interface[3].
- `Field` - Represents a single finitized field, and maintains information about the instrumented object, the field's domain, and the field's current value.
- `FieldDomain` - Stores the possible values for each `Field` instance. The values are separated by type, which is used for generating non-isomorphic inputs (Section 3.7).
- `ClassDomain` - Stores possible instances of a specified class, to be used within a finitized field's `FieldDomain`. This class creates and stores `FieldDomain` and `Field` instances for the finitized fields of the specified class.
- `Factory` - Provides the primary machinery for automated input generation. Stores information about all created `ClassDomain` and `Field` instances, as well as all fields accessed during search of the input space (Section 3.7).
- `Warehouse` - Stores a reference from each `Field`'s instance object to the `Factory` instance that maintains its state information. Primarily used for parallel operation (Section 3.8), where multiple `Factory` instances are utilized.
- `TestCase` - A subclass of Python's `unittest.TestCase` class that executes user-defined hooks for finitization, invariant checking, and functional testing. This class also initializes the multiple execution processes for parallel input generation and test execution (Section 3.8).

We describe the use of these classes in more detail throughout the following sections, where we demonstrate the use of PyIG on the binary tree example – ported to Python – from the Korat source code [6].

### 3.2 Object Factory

To facilitate generation of objects and their attribute combinations, we have created an object factory class, named `Factory`, that generates new objects of the specified type (`ClassDomain` objects) and provides the functionality for instrumentation and finitization of fields (Python attributes). The `Factory` class keeps track of all constructed class domains and contains data structures for looking up all finitized fields by the instances those fields instrument.

In Listing 2, we show a binary tree example from the Korat source tree [6] that we have ported to Python. This example first defines a `BinaryTree` class[4]and a `Node` class. Each instance of the `BinaryTree` class is initialized with two attributes: the `root` attribute which can reference a `Node` object, and the `size` attribute which maintains the size of the tree (i.e. the number of nodes reachable from its root node). Each instance of the `Node` class is initialized with `left` and `right` attributes that each may refer to another `Node` instance or may be empty (i.e. assigned to Python's `None` object).

[3]Python Data Model Reference: `http://docs.python.org/reference/datamodel.html`

[4]Note that for brevity we have left out the `rekOK` class invariant method on the `BinaryTree` class. The full code for this example can be found in Appendix A.1.

```
 1 import inputgen

 3 class BinaryTree(object):

 5     def __init__(self):
 6         self.root = None
 7         self.size = 0

 9 class Node(object):

11     def __init__(self):
12         self.left = None
13         self.right = None

15 f = inputgen.Factory()
```

**Listing 2: `BinaryTree` and `Node` class definitions and instantiation of a `Factory` instance.**

With the necessary classes defined, next a `Factory` instance is instantiated (line 15). This `Factory` object is used to create class domains and to store references to the `Field` instances and their state information, as described above.

## 3.3 Class Domains

With the `Factory` instance instantiated from Listing 2, we now use that object to create class domains. Class domains represent a set of objects of a specified class, where each object in the set is a possible concrete instance used during exhaustive exploration of the input space. Continuing with the previous example, Listing 3 shows the creation of a `BinaryTree` class domain and a `Node` class domain. The `create` method of the `Factory` instance is used to create these class domains.

```
16 tree = f.create(BinaryTree)
17 nodes = f.create(Node, 3, none=True)
```

**Listing 3: Creation of `BinaryTree` and `Node` class domains.**

The `create` method takes one required argument and two optional arguments. The first (required) argument specifies the class to represent, the second (optional) argument specifies the number of objects to create, and the third (optional) argument specifies whether or not Python's `None` object is an allowed value. The `create` method returns a `ClassDomain` instance, which stores information about any finitized fields on the class it represents.

Listing 3 creates two `ClassDomain` instances: one with a single `BinaryTree` object assigned to the variable `tree`, and one with three `Node` objects assigned to the variable `nodes`. Additionally, the `Node` class domain is created using the `none` argument set to `True`, meaning that the `nodes` `ClassDomain` instance will also allow Python's `None` object to be a possible value. Both of these created `ClassDomain` instances are stored within the `Factory` instance.

Although not used in this example, it is possible to create multiple `ClassDomain` instances of the same class. The fields for each class domain can be finitized independently, with state maintained separately in each `ClassDomain` instance without interference. In the next section, we show how the fields of the classes represented by `ClassDomain` objects are finitized.

## 3.4 Finitization and Instrumentation

With the class domains created from the previous section, we now finitize fields for the `BinaryTree` and `Node` class domains that were created and stored to the `tree` and `nodes` variables, respectively. Listing 4 shows this finitization. Using the `set` method of the `tree` and `nodes` `ClassDomain` instances from Listing 3, we set the field domain for the `tree` attribute on the `Factory` instance itself, as well as the field domains for the `left` and `right` attributes of the `Node` class.

In this example, both the `left` and `right` attributes of the `nodes` `ClassDomain` instance are set to the `nodes` `ClassDomain` itself. From Listing 3, recall that `nodes` represents a collection of four objects: the `None` object and three concrete `Node` instances.

```
18 f.set('tree', tree)
19 nodes.set('left', nodes)
20 nodes.set('right', nodes)
```

**Listing 4: Finitization and instrumentation of the `tree` attribute on the `Factory` instance and the `left` and `right` attributes of the `nodes` `ClassDomain` instance.**

In general, the specified field domains can be any Python collection object containing any number and any type of values. Behind the scenes, each call of the `set` method performs the following actions:

- A `FieldDomain` instance is created and all values given are stored, separated by type, within the `FieldDomain` instance.
- The newly created `FieldDomain` instance is stored within the calling `ClassDomain` instance, referenceable by the name of the field given as the first argument to `set`.
- The class represented by the calling `ClassDomain` instance is instrumented with a created `FieldDescriptor` instance.

The `FieldDescriptor` instance is what controls and records get/set accesses to a finitized field for each concrete instance within a `ClassDomain`. It is implemented as a Python descriptor, an object that defines special `__get__` and `__set__` methods that act as hooks into Python's standard get and set actions for instance attributes. In our implementation, `FieldDescriptor` instances maintain a reference back to a `Warehouse` instance, from which they lookup the value of the finitized field based on the state stored within the `Factory` instance that maintains its state. Upon access to the field, the `FieldDescriptor` instance also records the access on the correct `Factory` instance. Section 3.7 shows how these access recordings are used to help prune the input search space.

## 3.5 Integration with Python's `unittest` library

Python's standard library includes a unit test library (named `unittest`, and sometimes referred to as PyUnit), which is based on Java's JUnit. We have defined a custom `TestCase` class that inherits from the standard `unittest.TestCase` class and provides a framework for automated testing with generated inputs.

Listing 5 shows the interface for defining custom test case classes that make use of PyIG's automated input generation. With a class that inherits from `inputgen.TestCase`, below is a description of the required methods:

- The `repOK` method takes a `Factory` instance as its argument and returns `True` or `False` depending on whether or not the desired objects attached to the `Factory` instance meet some user-defined constraints, class invariant, and/or assertions.
- The `fin` method generates a `Factory` object and specifies the finitization of classes and fields. This is essentially what has been shown throughout the binary tree example from the previous sections, where the `Factory` object constructed in those examples would be the value returned by the `fin` method.
- The `run_method` method takes a `Factory` object as its argument and calls the desired method under test, using any objects available on the `Factory` instance.

```
class CustomTestCase(inputgen.TestCase):

    def repOK(self, factory):
        # ...

    def fin(self):
        f = inputgen.Factory()
        # ...
        return f

    def run_method(self, factory):
        # ...
```

**Listing 5: Interface for defining test cases that make use of automated input generation.**

## 3.6 Performing the Search

The key piece that ties the `TestCase` methods (Section 3.5) together lies within the `test` method of our custom `TestCase` class. The following actions are performed within this method:

1. A `Factory` instance is created by calling the `fin` method.
2. The `Factory` instance is initialized for search exploration with a call to `Factory.initialize`, which triggers creation of `Field` instances for maintaining the state of each finitized field, on each concrete instance, from all constructed class domains.
3. The `Factory` instance is passed to a call of the `repOK` method, which determines if the current set of objects/values represented by the `Factory` instance's state meet the desired assertions or class invariant.
4. If `repOK` returns `True`, then `run_method` is called with the `Factory` instance as its argument.
5. The `Factory` instance's state is advanced to the next input combination.
6. Steps 3-5 are repeated until all input combinations are exhausted.

Internally, state is kept with a list of indicies, one index for each finitized field. Each field's index refers to the current state of the field, an index into the list of the field's possible values stored within its associated `FieldDomain` instance.

## 3.7 Pruning the Input Search Space

Two major features of Korat that we have also implemented for PyIG are backtracking and non-isomorphism. The main algorithms for our implementations of these features live in the `Factory` class and are essentially a direct port to Python from the pseudo code and descriptions within the Korat MIT technical report [4].

For backtracking, the `Factory` instance records all field accesses during each execution of the `repOK` method, storing the accesses in the order of last accessed. When a `repOK` execution returns `False`, the algorithm backtracks to the last accessed field, incrementing to the next finitized value for that field. Once all finitized values of a field have been tried, the search continues with the remaining accessed fields in order of last access. Since all combinations of non-accessed fields are ignored, the input search space may be drastically reduced in comparison to the full search space that includes all combinations of all finitized fields, accessed or not.

In PyIG, we have also implemented the generation of non-isomorphic inputs, or inputs that are unique in structure. The key to our implementation for non-isomorphic inputs lies in the fact that the `FieldDomain` class stores each finitized field's values separated by type. When the next finitized value for a field is to be generated, the `FieldDomain` instance will skip ahead to the next value that is not the same type as the current value.

In some instances, it is desired that a `FieldDomain` instance try all possible finitized values for the field instead of skipping ahead to the next value of a different type. For instance, the `size` field for the binary tree example should try all possible finitized values because we are interested in isomorphic binary trees at every size from zero to the given maximum number of nodes. We have added support for this behavior with an `all` parameter for the `ClassDomain.set` method. Listing 6 shows how the `size` field is finitized in the binary tree example so that all possible size values are tried.

## 3.8 Parallel Execution

As the number of CPU cores on today's workstations increase, parallel execution becomes much more important for improving the performance of a software system. Since

```
sizes = range(0, max_size + 1)
tree.set("size", sizes, all=True)
```

**Listing 6: Disabling non-isomorphic input generation for a field by using `all=True` as a parameter to the `ClassDomain.set` method.**

Python's global interpreter lock (GIL) prevents multiple threads from running in parallel [5], we chose to utilize multiple processes instead (using Python's built-in `multiprocessing` library[5]).

The main code for parallel execution is built into our custom `TestCase` class (see Section 3.5). One process is started for each CPU detected on the machine running PyIG, and all processes share a single work queue for adding and removing work units. Each work unit consists of a set of starting indicies for the finitized fields and a list of accessed fields' indicies. The list of accessed fields' indicies acts as the stopping point for that unit of work, i.e. a process will continue generating inputs until its internal state of accessed fields matches the work unit's given list of accessed fields.

To begin work, a single work unit is pushed onto the queue with indicies all at zero and an empty list of accessed fields. When a process fetches a work unit from the queue, it will initialize its `Factory` instance's state to the given indicies and, as stated above, will continue generating input combinations until the list of accessed fields has worked down to the given list of accessed fields. Therefore, starting with the single work unit of zeroed indicies and an empty accessed field list is all that is needed to cause generation of all input combinations.

However, in order to break the work units up amongst all the running processes, the search will watch the list of accessed fields until the list grows in size. At that point, the process stops execution of its current work unit and adds two new work units to the queue. Given that:

- $I$ represents the current state of field indicies,
- $A$ represents the previous list of accessed fields,
- $A'$ represents the current, now larger list of accessed fields,
- $A'_t$ represents $A'$ truncated so that the length of $A'_t$ is equal to the length of $A$,
- $A_0$ represents the original list of accessed fields specified by the current work unit,
- and *next_indicies* is a pseudo function that returns the next set of indicies (given a starting set of indicies, a starting list of accessed fields, and a stopping list of accessed fields) or `None` if there is no next set of indicies,

[5]`http://docs.python.org/library/multiprocessing.html`

then the two work units added to the queue are:

1. $(next\_indicies(I, A', A_0), A'_t)$
2. $(next\_indicies(I, A, A_0), A_0)$

In other words, the set of indicies for the first work unit is the next set of indicies from the current state, given the new, longer list of accessed fields. The stopping condition for the first work unit is the new, longer list of accessed fields truncated to the length of the previous list of accessed fields. The set of indicies for the second work unit is the next set of indicies from the current state, given the previous list of accessed fields. The stopping condition for the second work unit is the original list of accessed fields specified by the current work unit. Additionally, the second work unit will only get added if there *is* a next set of indicies, i.e. search has not reached the list-of-accessed-fields stopping condition.

For example, looking within an execution of the binary tree example for four `Node` objects, we have the set of indicies and accessed fields shown in Listing 7. From line 2 to line 3, the list of accessed fields grows; thus, execution stops and new work units are added to the work queue. The first work unit added to the queue processes from line 4 until before line 6, when the list of accessed fields reduces back down to its original length at the time it was added to the queue. The second work unit added to the queue processes from line 6 until the list of accessed fields reduces down to its stopping condition (at some point in time after the last line shown in Listing 7).

While not quite a linear performance improvement for each additional process, the time taken to run the binary tree example reduced by roughly three and a half times on the authors' quad-core workstation when going from single process execution to multiprocess execution.

## 4. EXAMPLES

The binary tree example used throughout this paper is a Python port of the binary tree Java example from the Korat source code [6], and can be seen in its entirety in Appendix A.1. In sections 4.1 and 4.2, we demonstrate the use of PyIG on two non-data-structure examples: a form validation example using a popular Python Web framework and a structured document example, respectively.

### 4.1 Web Framework Form Example

In this section, we demonstrate the use of PyIG for generating inputs that pass validation checks for a Web form object from the Django Web framework[6]. Listing 8 shows a snippet of code for this example (the full code listing can be found in Appendix A.2).

Line 1 defines the form object, `MyForm`, containing:

- A character field, named `name`, that has a maximum length of 10 characters

[6]`http://www.djangoproject.com/`

```
1 <Factory, indicies: (0,1,0,0,2,1,0,0,0,0,0), accessed: (0,1,3,4,5)>
2 <Factory, indicies: (0,1,0,0,2,2,0,0,0,0,0), accessed: (0,1,3,4,5)>
3 <Factory, indicies: (0,1,0,0,2,3,0,0,0,0,0), accessed: (0,1,3,4,5,6,7,8,2)>
4 <Factory, indicies: (0,1,1,2,0,0,0,0,0,0,0), accessed: (0,1,3,4,5,6,2)>
5 ...
6 <Factory, indicies: (0,1,0,2,0,1,0,0,0,0,0), accessed: (0,1,3,4,5)>
7 <Factory, indicies: (0,1,0,2,0,2,0,0,0,0,0), accessed: (0,1,3,4,5)>
```

**Listing 7: An example input space search. At line 3, where the list of accessed field grows larger, execution on the current work unit stops and two new work units are added to the work queue.**

- An integer field, named `age`, that has a minimum value of 0 and a maximum value of 10

In the `fin` method, we finitize three fields on the `Factory` instance: `form`, `name`, and `age`. The `repOK` method uses these fields to set data on the form instance in the format it is expecting and then calls the form's `is_valid` method to determine if the data contained in the form object is valid according to its fields and their specifications.

```python
class MyForm(forms.Form):

    name = forms.CharField(max_length=10)
    age = forms.IntegerField(min_value=0,
        max_value=10)


class FormExample(inputgen.TestCase):

    @staticmethod
    def repOK(factory):
        form = factory.form
        form.data = {'name': factory.name,
            'age': factory.age}
        form.is_bound = True
        form._errors = None
        return form.is_valid()

    @staticmethod
    def fin():
        f = inputgen.Factory(
            iso_breaking_enabled=False)
        form = f.create(MyForm, init=True)
        f.set('form', form)

        names = ['fred', 'bob', '
            areallylongname']
        ages = [-3, 0, 3 , 99, 1000, '     
            5554   ', '5', 'notanumber']
        f.set('name', names)
        f.set('age', ages)
        return f
```

**Listing 8: An example that demonstrates using PyIG to generate valid inputs for a Web form object.**

### 4.2 XHTML Document Example

In this section, we demonstrate the use of PyIG for generating a valid-structured XHTML[7] document. Listing 9 shows a snippet of code for this example (the full code listing can be found in Appendix A.3).

In the `fin` method, we finitize the `doc` field on the `Factory` instance, as well as a variable number of fields that depends on the value given in the `fin` method's `size` parameter. We finitize `size` number of fields, named `pos0`, `pos1`, etc., to a list of possible strings including the `<p>` and `<b>` XHTML tags and a few simple words. The `XHTMLExample.repOK` method then calls `Document.repOK` through the `Document` instance stored in the `doc` field. `Document.repOK` concatenates all of the `posX` fields, inserts the generated string into the body of a simple XHTML template, and then validates the resulting content with the lxml library[8] and XHTML 1.0's document type definition (DTD)[9].

## 5. FUTURE WORK

Ideas for future enhancements of PyIG include:

- Pre- and post-condition assertion integration into our custom unit test class.
- A proper command line interface for executing PyIG and specifying runtime options.
- Additional Korat features besides the backtracking and non-isomorphism already implemented.
- Support for finitization of more than just class attributes, e.g. list item lookup and dictionary key lookup.

## 6. CONCLUSIONS

Using the Python language, we have implemented an automated input generation tool, named PyIG. We have illustrated the implementation and usefulness of this tool with three examples, including a complex data structure example and two non-data-structure examples. Our tool implements the backtracking and non-isomorphism features from Korat, which we demonstrated with a binary tree example from the Korat source code. We also used the binary tree example to demonstrate how our tool implements finitization and instrumentation. In the two non-data-structure examples, we showed how PyIG was used to generate valid Web form data inputs and valid XHTML documents. Additionally, we contributed a method and implementation of parallel execution for input generation, which significantly reduces execution

[7]`http://www.w3.org/TR/xhtml1/`
[8]`http://codespeak.net/lxml/`
[9]`http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd`

```python
class Document(object):

    def output(self):
        content = ''.join(getattr(self, 'pos%s' % i) for i in xrange(self.size))
        return template % content

    def repOK(self):
        """Validate using lxml."""
        output = self.output()
        try:
            root = etree.XML(output)
        except etree.XMLSyntaxError:
            return False
        return dtd.validate(root)


class XHTMLExample(inputgen.TestCase):

    @staticmethod
    def repOK(factory):
        return factory.doc.repOK()

    @staticmethod
    def fin(size=5):
        f = inputgen.Factory(iso_breaking_enabled=False)
        doc = f.create(Document)
        f.set('doc', doc)

        tags = ['<p>', '</p>', '<b>', '</b>']
        content = ['text1', 'text2', 'text3']

        doc.set('size', [size])
        for i in xrange(size):
            doc.set('pos%s' % i, tags + content)
        return f
```

**Listing 9: An example that demonstrates using PyIG to generate valid XHTML documents.**

time. With PyIG, developers are able to automatically generate large numbers of test case inputs with minimal time and effort.

## 7. REFERENCES

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM.

[2] I. Craggs, M. Sardis, and T. Heuillard. Agedis case studies: Model-based testing in industry. In *1st European Conference on Model Driven Software Engineering*, 2003.

[3] D. R. Hackner and A. M. Memon. Test case generator for guitar. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, pages 959–960, New York, NY, USA, 2008. ACM.

[4] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. In *Technical Report MIT-LCS-TR-921*, 2003.

[5] N. Matloff. *Programming on Parallel Machines.* `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`.

[6] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat source code. `https://korat.svn.sourceforge.net/svnroot/korat/trunk/`, Nov. 2010.

[7] W. Swain and S. Scott. Model-based statistical testing of a cluster utility. In V. Sunderam, G. van Albada, P. Sloot, and J. Dongarra, editors, *Computational Science - ICCS 2005*, volume 3514 of *Lecture Notes in Computer Science*, pages 443–450. Springer Berlin Heidelberg, 2005. 10.1007/11428831_55.

[8] G. Tassey. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project Number 7007.011, 2002.

[9] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2004. ACM.

## APPENDIX

## A. EXAMPLE CODE

This appendix presents the entire code for each of the three examples described in Section 4.

### A.1 Binary Tree Example

```python
"""
A binary tree example ported from an
    example found in the Korat source code.
"""

from collections import deque

import inputgen


class BinaryTree(object):
    def __init__(self):
        self.root = None
        self.size = 0

    def repOK(self):
        if not self.root:
            return self.size == 0
        # checks that tree has no cycle
        visited = set()
        visited.add(self.root)
        worklist = deque()
        worklist.append(self.root)
        while worklist:
            current = worklist.popleft()
            if current.left:
                if current.left in visited:
                    return False
                visited.add(current.left)
                worklist.append(current.
                    left)
            if current.right:
                if current.right in visited
                    :
                    return False
                visited.add(current.right)
                worklist.append(current.
                    right)
        # checks that size is consistent
        return len(visited) == self.size


class Node(object):

    def __init__(self, left=None, right=
        None):
        """
        Create a Node object.  left and
            right are optional and should
            be Node
        objects themselves.
        """
        self.left = left
        self.right = right



class BinaryTreeExample(inputgen.TestCase):

    @staticmethod
    def repOK(factory):
        return factory.tree.repOK()

    @staticmethod
    def fin(num_nodes=5, max_size=5):
        f = inputgen.Factory()
        tree = f.create(BinaryTree)
        f.set('tree', tree)

        nodes = f.create(Node, num_nodes,
            none=True)
        nodes.set('left', nodes)
        nodes.set('right', nodes)

        tree.set('root', nodes)
        sizes = range(0, max_size + 1)
        tree.set('size', sizes, all=True)
        return f

    def run_method(self, obj):
        pass
```

## A.2 Web Framework Form Example

```
"""
An example involving the validation of a
    Django Form object.  This example
requires that you have Django installed:
    http://www.djangoproject.com/
"""

from django.conf import settings
settings.configure()

from django import forms

import inputgen


class MyForm(forms.Form):

    name = forms.CharField(max_length=10)
    age = forms.IntegerField(min_value=0,
        max_value=10)



class FormExample(inputgen.TestCase):

    @staticmethod
    def repOK(factory):
        form = factory.form
        form.data = {'name': factory.name,
            'age': factory.age}
        form.is_bound = True
        form._errors = None
        return form.is_valid()

    @staticmethod
    def fin():
        f = inputgen.Factory(
            iso_breaking_enabled=False)
        form = f.create(MyForm, init=True)
        f.set('form', form)

        names = ['fred', 'bob', '
            areallylongname']
        ages = [-3, 0, 3 , 99, 1000, '␣␣␣␣␣
            5554␣␣␣', '5', 'notanumber']
        f.set('name', names)
        f.set('age', ages)
        return f

    def run_method(self, factory):
        pass
```

## A.3 XHTML Document Example

```
"""
An example involving the construction of a
    valid XHTML document.
This example requires installation of the
    lxml library:
http://codespeak.net/lxml/
"""

import os

from lxml import etree
```

```
import inputgen


parent_dir = os.path.dirname(__file__)
dtd = etree.DTD(open(os.path.join(
    parent_dir, 'xhtml1-strict.dtd')))


template = """\
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
    1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1
        -strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title></title>
  </head>
  <body>
    %s
  </body>
</html>
"""


class Document(object):

    def output(self):
        content = ''.join(getattr(self, '
            pos%s' % i) for i in xrange(
            self.size))
        return template % content

    def repOK(self):
        """Validate using lxml."""
        output = self.output()
        try:
            root = etree.XML(output)
        except etree.XMLSyntaxError:
            return False
        return dtd.validate(root)



class XHTMLExample(inputgen.TestCase):

    @staticmethod
    def repOK(factory):
        return factory.doc.repOK()

    @staticmethod
    def fin(size=5):
        f = inputgen.Factory(
            iso_breaking_enabled=False)
        doc = f.create(Document)
        f.set('doc', doc)

        tags = ['<p>', '</p>', '<b>', '</b>
            ']
        content = ['text1', 'text2', 'text3
            ']

        doc.set('size', [size])
        for i in xrange(size):
            doc.set('pos%s' % i, tags +
                content)
        return f

    def run_method(self, factory):
        pass
```