aPET: A Test Case Generation Tool for Concurrent Objects [*]

Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa
Complutense University of Madrid, Spain
{elvira,puri,mzamalloa}@fdi.ucm.es

Peter Y.H. Wong
SDL Fredhopper, Amsterdam, The Netherlands
pwong@sdl.com

ABSTRACT

We present the concepts, usage and prototypical implementation of aPET, a test case generation tool for a distributed asynchronous language based on *concurrent objects*. The system receives as input a program, a selection of methods to be tested, and a set of parameters that include a selection of a coverage criterion. It yields as output a set of test cases which guarantee that the selected coverage criterion is achieved. aPET is completely integrated within the language's IDE via Eclipse. The generated test cases can be displayed in textual mode and, besides, it is possible to generate ABSUnit code (i.e., code runnable in a simple framework similar to JUnit to write repeatable tests). The information yield by aPET can be relevant to spot bugs during program development and also to perform regression testing.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.5 [**Testing and Debugging**]: [testing tools, symbolic execution]; D.2.6 [**Programming Environments**]: [integrated environments]

General Terms

Languages, Verification, Reliability

Keywords

Test case generation, Concurrency, Concurrent objects, Software testing, Symbolic execution

1. INTRODUCTION

It is widely recognized that writing concurrent programs is error-prone. The concurrency errors that programmers face are frequently related to undesired task interleavings, which may be the result of wrong synchronization. For instance, it frequently happens that, when a task suspends its execution, another task starts to execute and possibly modifies the global state without the programmer expecting it. This type of synchronization errors can lead to unpredicted behaviors, to non-termination, to erroneous computed results, etc. Also programmers are not always aware of the behavior of the task scheduler. The scheduler can give priority to a task without the programmer expecting it. Such task can modify the global state and give the same types of errors as above.

Several techniques are used in practice to ensure the reliability of concurrent programs. Software testing [6] is one of the techniques that is most widely used in practice. The basic idea is to use some sample of the data that a program is expected to handle to test its functional behavior. If the program produces correct results for the sample, it is assumed to be correct. Most current research focuses on the question of how to choose this sample. Test case generation (TCG) is the process of automatically generating *test inputs* for interesting test *coverage criteria*. The test inputs are constraints on the input arguments that identify classes of inputs that lead to the same output. The testing tool uses such test inputs to validate the functional behavior of programs. The coverage criteria ensure termination and also that the code is sufficiently exercised by the test inputs.

aPET is a TCG tool for a distributed asynchronous language based on *concurrent objects*, named ABS [11]. The *actor*-based paradigm [1] on which concurrent objects are based has evolved as a powerful computational model for defining distributed concurrent systems. Actors are the universal primitives of concurrent computation: in response to a message, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. *Concurrent objects* are actors which communicate via *asynchronous* method calls, e.g. "`o!m();`" denotes an asynchronous call from the **this** object to object `o`. Each concurrent object (both **this** and `o` in this case) allows at most one *active* task to execute within the object. The synchronization between the caller and the callee methods can be optionally performed when the result is necessary by means of *future variables*. For instance, "`f = o!m(); await f?;`" denotes that the caller synchronizes with the result of the execution of `m` such that at the **await** point if `m` has not finished, the **this** object releases the lock and another process which is pending to be executed on the **this** object can start its execution. Each object has an unbounded set of pending tasks. When the lock of an object is free, *any* task in the set of pending tasks can (non-deterministically) grab the lock and start to execute. Scheduling among the tasks of an

[*]This demo illustrates the implementation of some of the techniques presented at PADL'12 [2] and ICLP'12 [3].

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2494590

object is cooperative such that a task has to release the object lock explicitly using **await**. This is an essential difference with thread-based concurrency since unlike threads switching between tasks happens only at specific points which are explicit in the source code. Non-deterministic and cooperative scheduling are features essential in the development of our testing tool. The underlying concurrency model of actor languages also forms the basis of the programming languages Erlang [5] and Scala [9] that have gained in popularity, in part due to their support for scalable concurrency. There are also implementations of actor libraries for Java.

2. DESCRIPTION OF aPET

ABS is a concurrent object-oriented language which offers an Integrated Development Environment (IDE) for writing ABS programs. The IDE is realized as an Eclipse plugin that can be downloaded from `http://tools.hats-project.eu` and installed by using the standard Eclipse installation routine. Additionally, we have developed an `aPET` Eclipse plugin which is completely integrated within the main ABS plugin and downloadable from `http://costa.ls.fi.upm.es/apet`.

`aPET` follows the standard approach to TCG of performing *symbolic execution* of the program [12, 8] in which the contents of variables are expressions rather than concrete values. Symbolic execution produces a system of constraints over the input variables consisting of the conditions to execute the different paths. The conjunction of these constraints represents the equivalence class of inputs that would take this path. We refer to a *test case* as such conjunction of conditions or constraints which a tester will use to determine whether the application is working correctly. By *test inputs* we refer to concrete instantiations of the test cases which satisfy their constraints and that can be used to actually run the application or method under test.

2.1 System Parameters

The main novelty of `aPET` is on the parameters that it provides to simulate the concurrency features during the TCG process. Besides concurrency-related parameters, `aPET` includes options to control aliasing of references, advanced coverage criteria, domains for generating input values for Integers and for Strings, etc. As these parameters are rather standard, we omit explanations on them.

2.1.1 Coverage Criteria (CC)

An important problem in symbolic execution is that, since the input data is unknown, the execution tree to be traversed is in general infinite. Hence it is required to integrate a *termination criterion* which guarantees that the length of the paths traversed remains finite while at the same time an interesting set of test cases is generated such that certain code *coverage* is achieved. The following criteria are available:

Loop-k: In order to ensure termination of each task, we provide the loop-count criteria [10] which limits the number of times we iterate on loops to a threshold `k`.

Task-switching: Applying the loop-k CC to all tasks does not guarantee termination because we can switch from one task to another an infinite number of times. The task-switching criterion ensures *fairness* in the selection of objects whose tasks are being tested by limiting task switches on each object to a threshold `k`.

For both CC, the value of `k` must be provided by the user. Let us consider the following method `sum`, defined in class **A**, and `doSum`, defined in class **B**:

```
1  Int sum(Int n){
2    Int r=0; Fut<Int> f;
3    if (n == 0) r = 0;
4    else{
5     f = this ! sum(n-1);
6     await f?;
7     Int aux = f.get;
8     r = n + aux;
9    }
10   return r;
11 }
```

```
1  Int doSum(Int m, Int n){
2    A o = new A();
3    if (m > 0){
4      Fut<Int> g;
5      g = o ! sum(n);
6      await g?;
7      r = g.get;
8    }
9    else r = o.q();
10   return r;
11 }
```

When symbolically executing method `doSum`, the **if** branch invokes asynchronously method `sum` on object `o`. Once method `sum` starts to execute, a new task `sum` is created at line 5 and added to the queue of tasks of `o`. Limiting the number of loop iterations (in this case recursive calls) of each task does not ensure termination because each recursive call is a new task. As a consequence, symbolic execution of `sum` does not terminate and the **else** branch of `doSum` will be never executed (nor tested). The task switching criterion allows us to bound the number of task switching per object. In particular, if we set up a threshold of 1, we get the solutions $\langle m{>}0, n{=}0, r{=}0\rangle$ and $\langle m{\leq}0, n{=}\_, r{=}v\rangle$, where v is some value returned by `q`. The first solution corresponds to executing `o ! sum(0)`, which requires one task switching on `o`. The second one exercises the **else** branch of `doSum`.

2.1.2 Scheduling Policies

One of the core components of a concurrent system is the task scheduler which decides which task to run next among those available in the queue. For the sake of flexibility, ABS assumes that scheduling is non-deterministic such that any pending task can be selected. As fields can be accessed by all tasks, different behaviors can occur depending on the order in which tasks are scheduled in the object. `aPET` implements a parametric task scheduler which can be instantiated to adopt the following policies for each of the objects: (1) FIFO, (2) LIFO and (3) *priorities*.

As an example, consider the methods below that belong to the same class, where `a` is a class field. Let us symbolically execute method `check`. We first adopt a FIFO strategy. When the execution arrives to the **await** `g`?, the queue of tasks for object **this** will contain the three asynchronous calls `toOdd`, `toEven` and `oddEven`. Now, the current task in which the **await** `g`? is executing also has to go to the queue since the value of `g` is not ready. The FIFO strategy selects `toOdd` that changes the value of field `a` to 1. Next, `toEven` updates the field with the value 2. Then, `oddEven` returns 0 as result, and it is stored in the future variable `g`. Now the execution of the **await** can proceed and the method returns $r{=}0$ as result. If we adopt a LIFO strategy in the queue of object **this**, we will not be able to compute any solution, because when the task **await** `g`? is suspended, it is put to the queue and extracted again. This process will be repeated until the threshold for the task switching coverage criterion is exceeded. Since the return statement is never reached, no test cases are generated. Finally, we consider a strategy based on priorities in which we need to take into account all possible priority orders for the four tasks `toOdd`, `toEven`, `oddEven`, and **await** `g`?. Test cases are only produced if task `oddEven` has higher priority than the task suspended in **await** g?. In total, we obtain 12 test cases.

```
Unit toOdd(){
  this.a = this.a*2+1;
}
Unit toEven(){
  this.a = this.a*2;
}
Int oddEven(){
  Int r = 0;
  if (this.a == 0) r = 0;
  else r = this.a % 2;
  return r;
}
```

```
Int a; // field
Int check(){
  Int r = 0;
  this.a = 0;
  Fut<Int> g;
  this ! toOdd();
  this ! toEven();
  g = this ! oddEven();
  await g?;
  r = g.get;
  return r;
}
```

2.1.3 Task Interleavings

An important problem in TCG of concurrent languages is that, when a task suspends, there could be other tasks on the same object whose execution could interleave with it and modify the information stored in the heap. Thus, when a task is suspended we need to take into account the different methods that can modify the heap. The first thing is to fix the maximum queue length per object. Given a fixed length, aPET system considers interleavings with:

pruning1: any method that modifies object fields, either directly or transitively by calling other methods that modify them;

pruning2: those methods that modify fields directly;

pruning3: those methods that write directly on fields which are used (read or written) before an **await**, and read after an **await**.

Let us symbolically execute, setting as maximum queue length 2, method pruning that contains an **await** synchronizing the execution with the value of field b.

```
Unit setField_b(Int n){
 this.b = n;
}
Unit setField_c(Int n){
 this.c = n;
}
Unit setField_bc(){
 this.setField_b(30);
 this.setField_c(80);
}
```

```
Int b = -5; // field
Int c = -5; // field
Int pruning(){
 Int r = 0;
 this.b = -1;
 this.c = 40;
 await (this.b > 0)?;
 r = this.b + 5;
 return r;
}
```

If we assume that the queue of pending tasks for the **this** object is empty when executing the **await** instruction, no test cases are computed. Using *pruning1*, symbolic execution assumes that methods setField_b, setField_c and setField_bc can be in the queue of **this**. Among the 12 test cases computed by aPET, one of them corresponds to the solution r=35, *this.b=30* and *this.c=80*, i.e., the last method executed in the queue is setField_bc. By using *pruning2*, we only consider interleavings with setField_b and setField_c. Thus, the solution shown before for *pruning1* is not computed for this pruning. One of test cases is *r=this.b* +5 together with the constraint *this.b>0*. No constraints on *this.c* are imposed. In total, *pruning2* computes 5 test cases which subsume those of *pruning1*. The intuition is that by considering the method that modifies the field alone, we execute it from a more general context, while its execution from another one will be just more specific. Finally *pruning3* only assumes interleaving with setField_b. The intuition for this option is that, if a field has not been accessed before the **await** then there is no information about the field. Thus, the context with no interleaving corresponds to the most general possible context for the field. If it is not read afterwards information about the field does not matter.

2.2 System Output

The test cases computed by aPET are provided in two different formats: (1) a textual mode which is meant to be used during software development to spot bugs and (2) ABSUnit code runnable in a simple framework similar to JUnit meant to be used for regression testing.

2.2.1 Test Cases in Textual Format

In the textual mode, we show the constraints on the input arguments and the returned value, as well as the initial and final states. The states are the lists of objects that have been accessed, where each object encapsulates information about its class and set of fields. Additionally, when we use a task scheduler based on priorities, the system returns also the priorities assigned to each task. As an example, for method check of Sec. 2.1.2, bounding the number of loop iterations to 1 and the number of task switching to 4, we get 12 different test cases. For all test cases *Args* = [*This*] (i.e., the only input argument is the **this** object) and the constraint store is [ ]. In 6 test cases oddEven has the maximum priority and, in 3 of them, odd has the maximum priority and, in the others, even has it. We show two test cases which compute the same returned value R but differ on the final heaps, where $\mathtt{S_i}$ and $\mathtt{S_f}$ stand for the initial and final heap respectively.

Args	=	[This] R = 0 C = [ ]
$\mathtt{S_i}$	=	[(0, obj('Check', [field(a, 0)], [ ])|0s]
$\mathtt{S_f}$	=	[(0, obj('Check', [field(a, 2)], [ ])|0s]
Priorities	=	odd,even,oddEven,await
Args	=	[This] R = 0 C = [ ]
$\mathtt{S_i}$	=	[(0, obj('Check', [field(a, 0)], [ ])|0s]
$\mathtt{S_f}$	=	[(0, obj('Check', [field(a, 1)], [ ])|0s]
Priorities	=	even,oddEven,await,odd

2.2.2 ABSUnit Code

The test cases obtained by aPET can be used to generate ABSUnit code, i.e., repeatable tests typically used for regression testing. Here we show the ABSUnit code generated from the first test case of the method check above, where the priority is odd, even, oddEven and await. The test method is named testCheck.

```
[Fixture] interface CheckTest {
  [Test] Unit testCheck();
}
[Suite]
class CheckTestImpl implements CheckTest {
  Check c; ABSAssert aut;
  { aut = new ABSAssertImpl(); }
  Unit testCheck() {
    this.setHeap(); Int r = c.check();
    aut.assertTrue(r == 0); this.assertHeap();
  }
  Unit setHeap() { }
  Unit assertHeap() { }
}
```

ABSUnit provides annotations (in brackets) to annotate various parts of the generated test codes. In this case the method is declared as a *test method* at the level of the interface. At the level of method implementation, it first invokes `setHeap` to set up the initial heap, which consists of object `c` of type `Check`. Next, method `check` is called on `c` and asserts that the return value is as expected. It also invokes the generated method `assertHeap` to assert that the invocation of `check` changed the heap as expected.

To correctly set up the initial heap and assert the final heap, we employ ABS's delta modules [7]. Delta modules realize code reuse in the paradigm of Delta-Oriented Programming [13]. A delta module is a named entity that describes the code changes associated with the realization of new features. A delta module may add, remove and modify existing classes, fields, interfaces and methods. Specifically two delta modules are generated for executing the test cases. The first of these, `MDeltaForCheck` completes existing interfaces and classes to permit easy setup of their initial state. For example, it provides a getter method for the object field `f`. The second delta, `TestDelta` modifies the methods `setHeap` and `assertHeap` to set up the initial heap and check the final heap. Here `TestDelta` initializes object `c` and asserts that the field `f` of `c` is updated to 2 after `check` is executed.

```
delta MDeltaForCheck;
adds interface MCheck
      extends Check {
  Int getf();
}
modifies class Check
      adds MCheck {
  adds Int getf() {
        return this.f; }
}
```

```
delta TestDelta;
modifies class CheckTestImpl{
 modifies Unit setHeap() {
    c = new Check();
 }
 modifies Unit assertHeap(){
    Int x = c.getf();
    aut.assertTrue(x == 2);
 }
}
```

3. CONCLUSIONS AND RELATED TOOLS

We have presented a state-of-the-art TCG tool for concurrent objects which features advanced options to control the concurrency behavior during the testing process, namely we can adopt different scheduling policies, simulate different tasks interleavings, and use a coverage criterion which ensures task scheduling fairness. Java Path Finder (JFP) [14] is a symbolic model checker for multi-threaded Java. It tries all possible scheduling sequences and interleavings. Thus, the user has less control during TCG than in our framework, since we can set up specific policies and select different criteria for task interleavings. Having fine-grained control is especially important when using TCG for bug discovery during software development. There are other tools (like MultithreadedTC) in which the user must specify the specific interleaving to be exercised. This lack of automation could be a handicap for the use of the tool by non-experts.

aPET uses the symbolic execution engine of the PET system [4]. This is a generic engine which can be applied to programs originating from different languages by first translating them to equivalent constraint logic programs (CLP) and then using the symbolic execution engine of PET for such CLP programs. We have translated ABS programs to CLP and then implemented the particular concurrency model used in ABS by means of CLP predicates. Also we have added our coverage criteria to the basic engine. Pex [15] is one of the most powerful tools for TCG, but it currently handles only sequential programs.

4. ACKNOWLEDGMENTS

This work was funded partially by the projects ICT-231620, TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and S2009TIC-1465.

5. REFERENCES

[1] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[2] E. Albert, P. Arenas, and M. Gómez-Zamalloa. Symbolic Execution of Concurrent Objects in CLP. In *Proc. of PADL'12*, *LNCS* 7149, pages 123–137. Springer, 2012.

[3] E. Albert, P. Arenas, and M. Gómez-Zamalloa. Towards Testing Concurrent Objects in CLP. In *Proc. of ICLP'12*, *LIPIcs* 17, pages 98–108. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[4] E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *Proc. of PEPM'10*, pages 25–28. ACM Press, 2010.

[5] J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proc. of ICSE'11*, pages 1066–1071. ACM, 2011.

[7] D. Clarke, N.Diakov, R. Hähnle, E. Broch Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In *Proc. of SFM'11*, *LNCS* 6659, pages 417–457. Springer, 2011.

[8] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[9] P. Haller and M. Odersky. Scala actors: Unifying Thread-Based and Event-Based Programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[10] W.E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[11] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10*, *LNCS* 6957, pages 142–164. Springer, 2012.

[12] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[13] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Proc. of SPLC'10*, *LNCS* 6287, pages 77–91, 2010.

[14] S. Shivaprasad and N. Prasad. Unit Testing Concurrent Java Programs. *Journal of Computer Applications*, 67(10):49–62, 2013.

[15] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In *Proc. of TAP'08*, *LNCS* 4966, pages 134–153. Springer, 2008.