Indeed, as we don’t have any information as to whether the generated “regression” tests can actually detect regression errors, we should de-emphasize this aspect in the paper, and focus mostly on the coverage achievements, while still mentioning the regression testing as the ultimate goal. Concretely speaking, I suggest to:

* Change title to “Automatic Test Completion via Mining Gigabytes of Dynamic Traces”, or “Automatic Generation of Missing Tests by Mining Gigabytes of Dynamic Traces”, or something like that
* ***Fixed. Please check the new title***
* Adding a paragraph at the beginning of the introduction which points out that a test suite that doesn’t have high code coverage only has limited regression-detecting capabilities
* ***Fixed***

-          Scott’s email address: [bwadswor@microsoft.com](mailto:bwadswor@microsoft.com)

***🡪 Fixed***

-          About Observer and state-modifying methods:

o   You say we only use pure observer methods to obtain values that get passed out via “out” parameters, to generate assertions for regression tests.  
I don’t see a reason why we could only do this for results of “pure” methods.  
So for the problem of collecting values as a test oracle, I wouldn’t introduce the whole concept of “observer” methods.

***🡪 Makes more sense. Fixed the issue with observer methods***

* However, I remember that we discussed ignoring calls to “pure” methods in the static analysis to detect duplicate PUTs. However, in the current version of the paper, I don’t see that this is mentioned in the treatment of duplicate PUTs.

🡪 ***Yes, initially we planned to do it. I have implemented that feature inside Shrinker but did not use it because it reduces the coverage as we may miss the PUTs with different observer methods.***

-          Definition of “Duplicate PUT”: You say that they must have the same “sequence of method calls”. What about the way the parameters are passed around? In fact, you later say that the methods are compared at the level of their MSIL instructions. However, you also say that the comparison at the level of MSIL instructions ignores “primitive values”. Why? Did we really do this? Did we do it to ignore details of “assertions” that Scott put into the code? If so, I’d say we just don’t talk about it at all.

***🡪 Yes you are right regarding ignoring primitive values. We did this for the values that Scott put in the code. I have removed the related sentences from the paper.***

-          The example PUT1 and PUT2 have control-flow in them, which is not representative of the actual dynamic traces. This was already a bit confusing in your presentation. I’d say let’s simplify the examples.

***🡪 Fixed. Moved the code with control into a method under test and made these example PUTs representative of our actual PUTs***

-          Definition of “Duplicate Seed Test”: Clarify what you mean by “execution path”, i.e.: starting from PUT, going through all (transitive) method calls.

***🡪 Fixed***

-          To motivate the minimization of duplicates, you could give some numbers: From how many to how many.

***🡪 Fixed***

-          When you describe the distributed setup, also give a reference to our TAP paper: There we used the same distributed setup, and it should be explained briefly. (Well, it was an older version of the distributed setup, but still.)

***🡪 Fixed***

-    Subject Applications: You motivate why we chose those 10 base class libraries. The main reason is that these base class libraries are shipped and maintained in different distributions (version 2 and version 4 of the “desktop CLR”, 32-bit / 64-bit, Silverlight, .NET Compact Framework, etc). Maintaining identical behavior and discovering discrepancies between these different distributions is paramount for the .NET product group.

***🡪 Fixed***

-    When you introduce the concept of “seed tests”, illustrate (at least verbally) how they are used by Pex: 1) Pex internally builds a tree of feasible execution paths; without seeds, it starts from the empty tree; with seeds, it starts from a pre-populated tree, 2) then Pex extends this tree by “flipping” individual branch nodes via constraint solving and actually test execution.

***🡪 Fixed***

Detailed suggestions

-          “of software development life cycle” -> “of the …”

***🡪 Fixed***

-          “A regression test is a unit test …” – many people might disagree. It’s just that in this paper, we will only deal with tests at the unit-level.

***🡪 Fixed***

-          “generate a minimal set” -> “…  a small set”

***🡪 Fixed***

-          “In our approach, we use Pex as an example state-of-the-art dynamic symbolic execution approach” -> “We use Pex, a state-of-the-art dynamic symbolic execution engine. However, our approach  is not specific to Pex, and it can be used in combination with any other test input generation engine.”.

***🡪 Fixed***

-          “writing good PUTs can still be challenging” -> “writing meaningful PUTs is often challenging”

***🡪 Fixed***

-          “PUTs require test scenarios” -> “PUTs require realistic test scenarios”

***🡪 Fixed***

-          “valid scenarios are quite small” -> “only few scenarios are meaningful in practice”

***🡪 Fixed***

-          “there exist three major categories of approaches that generate test scenarios” …, and then you even cite your previous work, which uses code mining. Does code mining fall into any of the three categories you mention? If not, why not make it “four major categories”, code mining being one of them?

***🡪 I cited the previous work because there we showed that these approaches are not effective. It is better not to mention more about our previous work here because that would give an immediate impression that there is no novelty in our new work. I had these experiences with my earlier papers. I changed the preceding sentence without mentioning our work explicitly.***

-          “happy paths” -> put into quotes, or make italic

***🡪 Fixed***

-          “we identify that even after minimization of PUTs and seed tests, the remaining number of PUTs can still be many and can take long time in exploring those PUTs.” -> “even after minimization of PUTs and seed tests, the number of remaining PUTs and seed tests can still be large, and it would take a long time (days or months) to explore those PUTs with dynamic symbolic execution on a single machine”

🡪 ***Fixed***

-           “These observer methods are put methods that do not change the state of their receiver or arguments” -> “All methods that return values can be regarded as observer methods. Call to pure methods that do not change the state of their receiver of arguments can be injected to enlarge the number of calls to observer methods.”

***🡪 Fixed. Removed definition of observer methods from the paper***

-          “1.5GB of dynamic traces” – that’s measuring the size of the C# source code, right?

***🡪 Fixed***

-          “A technique to record dynamic traces during program execution and generate PUTs and seed unit tests from recorded dynamic traces.” -> “Given a set of recorded dynamic traces, a technique to turn them into PUTs and seed unit tests.” (Let’s not claim what Scott did, as we don’t quite know what he is doing anyway. Also, collecting small unit tests from system tests was done before already. You can say that what we did was similar to this: <http://www.cs.washington.edu/homes/mernst/pubs/test-factoring-ase2005-abstract.html>)

***🡪 Fixed***

-          “From these PUTs, our approach … achieved a high code coverage” -> increased by how many percent?

***🡪 Fixed***

-          “that there many” -> “that there are many”

***🡪 Fixed***

-          “Our recorded traces include two kinds of methods” -> “In the following, we distinguish two kinds of methods in the recorded traces:”

***🡪 Removed observer methods from the paper***

-          “those seed tests exercise only the happy paths” -> “most seed tests only exercise common happy paths”

***🡪 Fixed***

-          “seed tests do not achieve high coverage of the code under test” -> “seed tests do not achieve high coverage of the corner cases and error handling of the code under test”

***🡪 Fixed***

-          You spell “OUT” parameters with upper-case letters, but it’s lower-case in C#.

***🡪 Fixed***

-          “can result in the corruption of the machine” -> “can cause external side effects, and change the machine configuration”

***🡪 Fixed***

-          “The reason for not able to present the coverage as a percentage of the number of blocks to the total number of blocks is that we do not know how many blocks in the codeunder test are actually reachable.” -> “We don’t give an upper bound on the number of reachable basic blocks, as we don’t know which blocks are actually reachable from the given scenarios.”

***🡪 Fixed***