# Model-based testing without a model: Assessing portability in the Seattle testbed

Justin Cappos
*University of Washington*
*Seattle, Washington USA*

Jonathan Jacky
*University of Washington*
*Seattle, Washington USA*

**Abstract**

Despite widespread OS, network, and hardware heterogeneity, there has been a lack of research into quantifying and improving portability of a programming environment. We have constructed a distributed testbed called Seattle built on a platform-independent programming API that is implemented on different operating systems and architectures. Our goal is to show that applications written to our API will be portable.

In this work, we use an instrumented version of the programming environment for testing purposes. The instrumentation allows us to gather traces of actual program behavior from a running implementation. These traces can be used across different versions of the implementation exactly as if they were test cases generated offline from a model program, so we can commence testing using model based testing tools, without constructing a model program.

Such offline testing is only effective in scenarios where traces are expected to be reproducible (deterministic). Where reproducibility is not expected, for instance due to nondeterminism in the network environment, we must resort to on-the-fly testing, which does require a model program. To validate this model program, we can use the recorded traces of actual behavior. Validating with captured traces should provide greater coverage than we could achieve by validating only with traces constructed *a priori*.

## 1 Introduction

While most programmers understand the concept of portability, there has been surprisingly little research into how to quantify and improve this aspect of a programming environment. The result of this is that the state-of-the-art in portability is "write once, debug everywhere" [5, 9, 4], as is provided by Java and .NET. Portability is an important problem both for platform developers and application programmers.

We confront the portability problem in our practically deployed peer-to-peer network research testbed called Seattle [3, 17]. Researchers write applications in a Python [7] subset that runs on the Seattle testbed to perform tasks like measuring the Internet, running peer-to-peer software on representative end hosts, evaluating overlays, and studying real user availability and mobility patterns. We provide implementations of the Seattle testbed that run on many diverse platforms, including a wide variety of desktops, laptops, and phones. For example, the operating systems currently supported include Windows XP, Windows Vista, Linux, BSD, Mac OS X, as well as some mobile variants.

It is our goal that researchers' applications should be portable: the same application code should run and behave the same on all the platforms that Seattle supports. It is infeasible to require that each application programmer port and test their program on every platform where it might run. We strive to provide strong assurances that applications that run on Seattle will behave the same on the different platforms.

For each platform, there is a different Seattle implementation. Each implementation includes code that, in effect, translates application calls to the Seattle API into calls to the platform-specfic operating system APIs, then translates the results back to the values returned by the Seattle API. These translations deal in part with syntax: the API calls, the numbers and types of their arguments and results. They also deal with semantics, or behavior: the actual values of the arguments and results, and the ordering and timing of calls.

Another complicating factor is nondeterminism that arises from differences in the network environment. Even in a situation where two systems are completely identical in hardware and software configuration, their network characteristics will differ. The systems will have different IP addresses, will have different packet loss / delay characteristics, and may contact different DNS servers. Each of these types of heterogeneity may impact the behavior of an executing program and cause it

to behave differently.

We anticipate that, despite our efforts, our Seattle implementations might exhibit portability bugs: differences in behavior on different platforms.The platform-specific APIs are complicated and some are closed, so we cannot rely on inspection or analysis to prevent such errors. We must resort to testing. Testing is the only assurance method that actually executes the implementation in an environment similar to where it will be used, so it checks our assumptions about the execution environment — which are exactly what is at issue here.

We needed a testing technology that could generate lots of tests automatically and deal with nondeterminism, so we chose *model-based testing*. Model-based testing is an automated testing technology that uses an executable specification called a *model program* as both a test case generator and an oracle [20, 13]. As a bonus, the model program can also serve as a formal specification of the Seattle API, expressing the well-defined semantics to which we aspire. Then can use the framework's analysis tools (including a kind of model checker) to check properties of our design, using the same models that we use for testing. Since the Seattle testbed is written in Python, we use the PyModel framework [16].

It is necessary to validate a model program — to show that it describes the intended behaviors. The usual approach is to check the model program with short samples of behavior which are known *a priori* (from a written or implicit specification) to be allowed or forbidden. It can be difficult to know whether a collection of such samples provides adequate coverage, exhibiting the range of behaviors that implementations will exhibit "in the wild".

Our situation is a bit unusual because we have to test several implementations, not just one. Moreover, an important requirement is that the several implementations must behave the same as each other, not just that all conform to the same specification. (It would be possible to write a specification loose enough that several implementations would conform, yet still behave so differently that they violate reasonable expectations of portability).

We can turn this situation to our advantage. We can (provisionally) choose any one implementation to be the reference implementation, considering it to be an executable specification to which the others must conform. It is not unsound for us to use one (possibly erroneous) implementation to check another, because our purpose here is to assess portability, rather than to check conformance to a specification. We are trying to expose differences in behavior between different implementations. Any difference is unexpected and requires investigation. It is not necessary at this stage to identify which implementation is correct.

Considering an implementation to be an executable specification places it in a role similar to a model program. In fact, we can exploit this similarity to turn an implementation into an offline test generator. We use the existing Seattle interposition technology to collect samples of behavior called *traces* in the same format that the model-based testing framework's offline test generator writes test suites. These traces can then be played through the framework's test runner in the same way, to provide a kind of model-based testing without a model.

Captured traces complement the model program — they do not replace it. We expect that the captured traces, since they record actual system behavior "in the wild", may exhibit features and expose errors that are not covered by small test cases constructed by hand from a written (or implicit) specification, or generated from a model program based on those same sources. The model program is still valuable as a formal specification that expresses the intended semantics, for model-checking analysis, and it is essential for on-the-fly testing, which is needed to handle nondeterminism.

Offline testing using stored traces, whether captured or generated by a model, is only effective in scenarios where traces are expected to be reproducible (deterministic). Where reproducibility is not expected, for instance due to nondeterminism in the network environment, we must resort to on-the-fly testing, which does require a model program. To validate this model program, we can still use the recorded traces of actual behavior. Validating with captured traces should provide greater coverage — and more confidence — than we could achieve by validating only with traces constructed *a priori*.

### 1.1 The innovations in this work

Our contribution (beyond applying model-based testing to this kind of system) is to add two new techniques to the model-based testing toolkit: using captured traces to validate models and using traces as offline test suites. The idea of using samples of observed behavior (traces) in assurance activities has many precursors. The particular engineering innovation here is integrating trace capture and replay into an existing model-based testing framework, using its interposition facility to capture the traces and its test runner to execute them as test suites. This makes testing with captured traces convenient and accessible. We believe this makes a significant addition to the software tester's toolkit when portability is at issue.

This report describes the rationale and supporting techology for our approach. At this time we have performed some preliminary experiments to establish the technical feasibility of this approach, but do not yet have sufficient experimental data to report here. We are eager to learn whether testing with captured traces exposes portability errors that are not detected by traces generated from a model based on the written specification.

### 1.2 Roadmap

The remainder of the paper is organized as follows. First, Section 2 describes the Seattle platform in more detail, focusing on its programming API. In Section 3, we discuss how behavior is represented by traces. Then, in Section 4 we precisely define portability in terms of traces. Following this, Section 5 describes how a model-based testing framework uses traces in offline and on-the-fly testing. Validation for deterministic programs by utilizing multiple implementations is discussed in Section 6. Then Section 7 discusses how we can utilize traces to evaluate nondeterministic portions of the API. Section 8 shows how our technique relates to traditional on-the-fly testing techniques. In Section 9, we discuss related work. Section 10 concludes.

## 2 Seattle Testbed

In order to understand how evaluate the portability of the Seattle API, one must first understand the programming environment [10]. After examining the entire API at a high level, a subset of the API is described in detail. Programs using this subset will be used in examples throughout in the paper. Then we describe the interposition mechanism which is used to gather API traces and validate portability.

### 2.1 API Overview

A program that executes in Seattle is handled by two separate components, the *interpreter* and the sandbox kernel. The computational portions of the program are handled through access to a subset of the normal Python interpreter [7]. Each Seattle program can freely use a set of 87 symbols mapped in by the Python interpreter for basic computations such as comparison to None, type conversion, and basic math and list operations. This portion of the API is comprised of functionality that is simplistic and computational in nature. The exposed portions of the interpreter do not access complex resources like the file system or network. As a result, any program that relies solely on the interpreter is deterministic.

Operations that involve the operating system may only be issued through the sandbox kernel. The sandbox kernel exposes a set of 32 functions that behave in a platform independent manner. These functions can be summarized as follows:

- Six file I/O functions involving access to a sandbox-specific directory on the file system. These deterministic functions allow the user to open a file, read at a location in the file, write at a location in the file, close the file, delete a file, and list the files in the sandbox. These functions are described in more detail below.
- Three deterministic functions that manipulate lock objects. The first function creates and returns a lock object. The lock object has methods to acquire or release the lock.
- Two deterministic functions that provide debugging information. The first function returns a string to describe the last error's stack trace. The second function lists the current thread's name.
- Three thread-related functions: a function to create a new thread of execution for a function, a function to sleep the current thread, and a function to force all threads to exit. These functions are deterministic, but the creation of additional threads may lead to data races if the underlying program has them.
- Two functions that validate and execute dynamic code. This allows mechanisms like import and eval to be constructed. The validation function is deterministic, but the execution of dynamic code is deterministic when the code that is being executed is also deterministic.
- Thirteen network functions, to perform DNS lookups, obtain the local IP address, and send / receive TCP and UDP traffic. All of these functions are non-deterministic. The non-TCP functions will be described in more detail later.
- Two non-deterministic functions that provide accounting information. One function returns resource utilization information and the other returns the elapsed time since the program started.
- A function to return random bytes suitable for cryptographic use. This function is obviously non-deterministic.

It is important to note that the sandbox kernel is the only interface by which a program can utilize resources on a user's machine. Using the above interface, we have built standard libraries that reconstruct common language functionality. Utilizing the minimal functionality provided by the sandbox kernel we were able to provide access to large amounts of Python functionality including `print`, `eval`, traceback handling, basic file I/O, and many types of introspection. In addition, all of the standard libraries are also implemented on top of the sandbox kernel. This includes cryptographic libraries, XML parsing, RPC, serialization, NAT traversal, HTTP client / server code, argument parsing, advanced synchronization primitives, and a variety of encoding schemes. The fact that a large number of programs all use a small, well-defined interface makes Seattle a good test environment for examining portability.

There are some portions of the sandbox state which may be set before the program begins executing. The

sandbox can be given a set of command line arguments. Additionally, the file system can be manipulated externally. This allows an external developer the ability to add the file that should be executed. Similarly, the developer can upload, download, or remove other files in the storage area of the sandbox. However, these facilities are disabled while the sandbox is running. Thus, the storage area of an executing sandbox will change only when the sandboxed program modifies it.

### 2.2 API Subset

In order to make examples in the remainder of the paper more concrete, we will describe a subset of both the deterministic and non-deterministic portions of the API in greater detail. The first six functions we describe comprise the file system API. These functions can be described as follows:

- **listfiles()** Returns a list of the filenames in the sandbox's storage area.
- **removefile(filename)** Removes a file from the sandbox's storage area. An exception is raised if the filename does not exist, is invalid, or if the file listed is currently open.
- **openfile(filename, createbool)** Opens a file, possibly creating the file if it does not exist. An exception is raised if the filename is invalid, the file is already open, or the file is not found but createbool is False. Returns a simplefile object.
- **simplefile.readat(sizelimit, offset)** Returns up to sizelimit bytes from the location offset in the file (less may be read due to EOF). If size is None, the entire file after the offset is read. If the offset or size are negative or the wrong type an exception is raised. An exception is also raised when offset is past the end of the file or if the file is closed.
- **simplefile.writeat(data, offset)** Writes data bytes at the location offset in the file. If the offset is negative or either argument is the wrong type an exception is raised. An exception is also raised when offset is past the end of the file or if the file is closed.
- **simplefile.close()** Close the file. A closed file cannot be read or written any more.

In addition to the deterministic file system functions, we describe a portion of the non-deterministic API. This portion of the API covers the UDP networking portion of the system and also includes several helper networking functions.

- **gethostbyname(hostname)** Provides the IP address given a string containing a hostname. The IPv4 address is returned as a string, such as `100.50.200.5`. If the hostname is an IPv4 address it is returned unchanged. A exception is raised if the hostname is not a string or if the address cannot be resolved for any reason.
- **getmyip()** Returns the localhost's "Internet facing" IP address as a string. It will raise an exception on hosts that are not connected to the Internet.
- **sendmessage(destip, destport, message, localip, localport)** Sends a UDP message to a destination host / port using a specified localip and localport. Returns the number of bytes sent which may be less than the entire message. The IP addresses must be strings containing valid IPs (not hostnames), the message must be a string, and the ports must be integers between 1 and 65535. This function can raise exceptions when the arguments are invalid or the localip argument isn't a local IP.
- **listenformessage(localip, localport)** Registers a callback that will be called upon receipt of an incoming UDP message on the specified IP and port. If `listenformessage` is called multiple times on the same ip and port without the first udpserversocket being closed, the second call will have an exception. This function will raise an exception if it would need to block while waiting for the previous connection to be cleaned up, if the arguments are invalid, or if the localip isn't a valid, local IP address. This function returns a `udpserversocket` object which is used by the next two functions.
- **udpserversocket.getmessage()** Receives a message that was sent to an IP and port. If the `udpserversocket` was previously closed, an exception is raised. An exception is also raised if the function would block.
- **udpserversocket.close()** Closes the `udpserversocket`. Returns True on the first call and False on any subsequent calls.

These functions are used by many libraries to perform high-level network operations. For example, our library that queries the time via NTP will look up a time server's IP using its hostname (`gethostbyname`). Next, the node will acquire its public facing IP address (`getmyip`) and send a message with the NTP request to the NTP server (`sendmessage`). Then the node will begin to listen for the upcoming UDP reply (`listenformessage`) and begin polling until it receives the reply (`udpserversocket.getmessage`). Following this, the server stops listening on the UDP server socket (`udpserversocket.close`).

```
(listfiles_start, ()),
(listfiles_finish, (["junk.testfile"],)),
(removefile_start, ("junk.testfile",)),
(removefile_finish, ()),
(open_start, ("junk.testfile", True)),
(open_finish, (fileobject0)),
(filewriteat_start, (fileobject0, "hello world!!!", 0)),
(filewriteat_finish, ()),
(filewriteat_start, (fileobject0, "ked", 9)),
(filewriteat_finish, ()),
(filereadat_start, (fileobject0, None, 0)),
(filereadat_finish, ("hello worked!!"),
(fileclose_start, (fileobject0)),
(fileclose_finish, ())
```

Figure 1: Trace collected from an instrumented Seattle API implementation, showing file system activity

### 2.3 API Interposition

The Seattle sandbox has a mechanism which allows a programmer to interpose on calls to the sandbox kernel. Essentially, a programmer may remap a user program's API to use a separate set of functions instead of directly calling the sandbox kernel. This is used to enforce security policies as well as perform debugging and forensic logging. In this work, we utilize the interposition mechanism to validate the portability of the system as a program executes. This is performed using a trace that is generated by interposing on each of the kernel's calls. The trace logs the call type and call arguments and then issues the call. Upon return, the return or exception information is also logged.

## 3 Traces

Traces describe samples of behavior. They are ubiquitous in model-based testing; most activities either generate or use traces. In this work, we use traces both as test cases and as standards for validation.

A trace is a sequence of units called *actions*. In this project, the actions are calls and returns of the Seattle API. Figure 1 shows a trace that records some calls to the Seattle file system API discussed in section 2. This trace is in the format written by our instrumentation, which is also the format used for offline test suites by our model-based testing framework. Each API call and return are separate actions, indicated by `start` and `finish` in the action names. Each action also includes arguments in a (possibly empty) tuple. We must distinguish *controllable actions* (the calls) from *observable actions* (the returns). We split each API call into two actions this way to account for the possibility that a call might not return, and also to support threading. Notice that the return values of each API call (controllable action) appear as arguments of the accompanying return (observable action). The argument values of controllable actions are under the tester's control, while argument values of observable actions are not.

This trace is deterministic, in the sense that persistent

```
(getmyip_start, ()),
(getmyip_finish, ("128.208.3.72",)),
(open_start, ("ntpservername", True)),
(open_finish, (fileobject0)),
(filereadat_start, (fileobject0, None, 0)),
(filereadat_finish, ("time-a.nist.gov"),
(fileclose_start, (fileobject0)),
(fileclose_finish, ())
(gethostbyname_start, ("time-a.nist.gov",)),
(gethostbyname_finish, ("129.6.15.28",)),
(sendmessage_start, ("129.6.15.28", 123, \
     "\x1b\x00...", "128.208.3.72", 12345)),
(sendmessage_finish, (48,)),
(listenformessage_start, ("128.208.3.72", 12345)),
(listenformessage_finish, (udpserversocket0,)),
(udpsockgetmessage_start, (udpserversocket0,) , None),
(udpsockgetmessage_finish, ("129.6.15.28", 123, \
     "\x1c\x01...")),
(udpsockclose_start, (udpserversocket0,)),
(udpsockclose_finish, (True,))
```

Figure 2: Trace collected from an instrumented Seattle API implementation, showing network activity

data is assigned in controllable actions, and is therefore under the tester's control, so it can be made repeatable. For example, notice how *hello* in the argument of the controllable action `writeat_start` later reappears in the return value (that is, the argument) of the observable action `readat_finish`. If the program that generated this trace was run on a different Seattle node, the same trace will be generated again.

Figure 2 shows a trace that records the NTP scenario discussed in section 2. This trace is *non*-deterministic, in the sense that persistent data first appears in observable actions, and is therefore *not* under the tester's control, so it cannot be made repeatable. For example, notice how the IP addresses *128.208.3.72* and *129.6.15.28* appear in the return values (that is, the arguments) of the observable actions `getmyip_finish` and `gethostbyname_finish`, respectively, are later used in the arguments of the controllable action `sendmessage_start`. If the program that generated this trace was run again, especially on a different Seattle node, it is likely that different IP addresses would appear.

## 4 Understanding Portability

This section discusses portability in order to convey the property we would like to obtain and then defines a property called practical portability that we will ensure Seattle meets.

A program $P$ (such as the Seattle API) can be said to be *perfectly portable* between environments $E_1, E_2, \ldots E_n$ if every input to $P$ (a trace including particular controllable actions) produces an identical trace (including the same observable actions) on $E_1, E_2, \ldots E_n$. This means that an executed program will behave the same on each run in the supported environments. However, this definition is unverifiable for two reasons.

- **Input Verification.** In practice it is impractical to test every possible input for a program.
- **Nondeterministic Calls.** The requirement of strict trace matching will not work for programs that use *nondeterministic calls*: controllable actions where the following observable actions are expected to vary across different executions. Nondeterministic calls are those affected by network behavior, hardware randomness, the local time, or scheduling (for a program which has data races).

We can define a looser property to allow us to analyze programs in practical scenarios.

**Definition 1** *A program $P$ is* practically portable *if in every* given *trace where the response to every nondeterministic call is the same as in the reference implementation, the entire trace is the same.*

There are two key parts to this definition. First, the definition limits the set of inputs to be those that the program is run with. This allows the verification to avoid reasoning about every possible program input. Second, the nondeterministic calls are allowed to vary across different runs. However, subsequent processing of the values that nondeterministically appear in observable actions is the same.

# 5 Model-based testing

Model-based testing is an automated testing technology that uses an executable specification called a *model program* as both a test case generator and an oracle [20, 13]. Model-based testing works with traces, as appear in Figures 1 and 2. Model-based testing can only test the items that appear in the traces: the calls in the Seattle API. Model-based testing does not test the process that creates the traces: the application program and the interpreter that executes it (except for the interpreter's processing of the calls to the Seattle API).

## 5.1 Model programs

A model program is a kind of executable specification. Developers or test engineers must write a model program for each implementation program (for example, the Seattle API) that they wish to analyze or test. By using various tools from the model-based testing framework, the same model program can support analyses by a technique similar to model-checking, can generate test suites (collections of traces) offline or on-the-fly, can act as an oracle (it can check traces).

A model program models the implementation as a collection of guarded update rules. For each action in the implementation (for example, `listfiles_start`, `listfiles_finish`, `removefile_start`, etc. in Fig. 1) there are two functions in the model program: a *guard* and an *update rule*. There are also some state variables. The guard (also called an *enabling condition*) is a Boolean function of the action arguments and state variables that returns True when that action is enabled in that state with those arguments. The update rule can update the state variables, possibly using the values of the action arguments. To generate a trace (a test run), the test generator starts in the initial state, chooses an enabled action, executes the update rule, then chooses an action which is enabled in the new state, etc. In general, several actions are enabled in each state; the generator uses a programmable strategy to select one. To check a trace, the oracle successively checks whether each action in the trace is enabled in the current state, and if so, updates the state by executing the update rule; otherwise, it indicates the action is forbidden, rejects the trace, and exits. To analyze the model, the analyzer uses a method similar to model checking called *exploration*: it builds a finite state machine (FSM) from the model program by selecting several enabled actions in each state (by some programmable criteria), computing all their next states (backtracking as needed), continuing until some stopping condition is reached. The generated FSM can then be searched to check various properties. In practice, properties to be checked can often be coded in a way that can be used during exploration to efficiently limit and direct the search, using a method called *composition* that is a generalization of computing the intersection of finite automata.

## 5.2 Offline testing

Offline testing proceeds in two stages. In the first stage, the framework's *offline test generator* produces a trace from a model program. In the second stage, the framework's *tester* or *test runner* causes the implementation to execute each controllable action in the trace (it calls the Seattle API), and checks whether the implementation performs each observable action in the trace (it checks the return values). When the implementation returns a result which is different from the one in the captured trace, the test runner indicates a test failure. This second stage does not require a model because all the needed information is in the traces. Offline testing is effective where traces are expected to be reproducible (deterministic).

## 5.3 On-the-fly testing

*On-the-fly testing* is needed in scenarios where reproducibility is not expected, due to nondeterminism (in the network environment, for example). To perform on-the-fly testing, the test runner does not use a pre-computed trace, instead it uses the model program to generate the trace as the test run executes. The test runner executes the model program during the test run in order to choose controllable actions to execute in the implementation, and

also executes the model program to check the results (observable actions) from the implementation. The test runner captures data from observable actions and uses that data in subsequent controllable actions. When the implementation executes an observable action that should not be enabled in the current state (according to the model), the test runner indicates a test failure. The test runner records the trace of the on-the-fly test run as it executes, so test failures can be investigated.

### 5.4 Validation

It is necessary to validate a model program: to show that it exhibits the intended behaviors. The usual way to do this is to use the model program as and oracle and see if it accepts (or rejects) sample traces the are allowed (or forbidden), constructed *a priori*, guided by a written or implicit specification. An innovation of this work is to also check models with traces captured from an instrumented platform.

## 6 Deterministic API Evaluation

We can validate the portability of deterministic programs, such as the trace of file system activity in Figure 1. Recall that the API calls in this trace are deterministic, in the sense that persistent data is assigned in controllable actions, and is therefore under the tester's control, so it can be made repeatable.

We can test the deterministic calls in the Seattle API by using traces like these as offline test suites with our model-based testing framework. We select a particular implementation (on one particular platform) to be the *reference implementation* for purposes of testing portability. We use our interposition instrumentation to collect traces from the reference implementation while it is running various application programs "in the wild". In effect, we use the instrumented reference implementation as our offline test generator, so we do not need a model program for these tests. This allows us to utilize trace data from real programs that are longer and more complex than we would generate a priori.

Then we execute these captured traces as offline tests in another implementation. We call this variant of offline testing by the name *trace replay*. Portability problems are indicated by test failures during trace replay. Each test failure is an incident where the implementation under test behaves differently than the reference implementation.

Note that there is nothing special about the reference implementation. In many cases we run the program we want to verify across a variety of systems with different heterogeneity properties. This ensures that the behavior is deterministic across all tested implementations.

## 7 Nondeterministic API Evaluation

For programs that have non-determinism, such as the trace of UDP activity in Figure 2, we cannot leverage the same techniques as for deterministic programs. Recall that the API calls in this trace are *non*-deterministic, in the sense that persistent data first appears in observable actions, and is therefore *not* under the tester's control, so it cannot be made repeatable.

In general, offline testing, which uses pre-computed traces, or (in our project) recorded traces, cannot be used to test API calls that are not expected to be repeatable.

To test these calls, we must resort to on-the-fly testing, which does require a model program. We write the model program, based on the written and implicit specification (similar to the description in section 2 here). To validate this model program, we can use our captured traces of actual behavior from the reference implementation. We use the framework's analyzer to check whether the captured traces are accepted by the model program. The model should accept all of the traces captured from the reference implementation when it is executing applications "in the wild". If necessary, we revise the model program until it accepts all of these traces. We call this *empirical validation*. This provides greater confidence in the portability of the Seattle testbed than we could achieve by relying solely on constructed traces, which we now call *a priori validation*. (We also ensure that the model is not too permissive by confirming it does not accept forbidden traces we produce by altering captured traces).

Then we use the model program to execute on-the-fly tests with the empirically validated model on a different implementation. Portability problems are indicated by test failures during on-the-fly testing. Since the model was empirically validated on the reference implementation, which is considered the ultimate oracle for these tests, a failure indicates that the behavior of the implementation under test may differ from the behavior of the reference implementation. In general an on-the-fly test run is not reproducible on the reference implementation, so we must investigate the trace of each failing test run to decide whether it indicates a portability problem.

## 8 Integration with model-based testing

A model-based testing framework provides a versatile collection of tools that can be used in various modeling, validation, and testing activities. Here we present a systematic classification of these techniques in order to show how this project relates to other work.

The entries in Table 1 show model-based testing activities with their inputs and outputs. Each entry in the table represents a particular activity that might be provided by a tool in a framework. The entries marked with asterisks show the activities which are the innovations reported in

| | Generates traces by | Uses synthesized traces for | Uses captured traces for |
|---|---|---|---|
| Impl. only | Trace capture ∗ | Offline testing | Trace replay ∗ |
| Model only | Offline test generation | *a priori* Validation | Empirical validation ∗ (check the model) — — — — — Passive testing (check the traces) |
| Impl. & Model | On-the-fly testing (control some actions) — — — — — Run-time verification (observe all actions) | | |

Table 1: Model-based testing activities. Areas covered by this work are marked with a ∗

this paper.

Each row in the table represents the kind of program that is the input to the activities in that row. An *implementation* is the program under test; in this work each version of the Seattle API running on a particular platform is an implementation. A *model program* is an executable specification used as a test generator and test oracle. There are three rows in the table because an activity may use one or the other, or both.

Each column in the table represents the generation or use of traces by the activities in that row. *Synthesized* traces are generated from a model program (or created "by hand"). *Captured* traces are collected by instrumentation from a running implementation. Activities in the first column generate traces, those in the second column use synthesized traces, and those in the third use captured traces.

Let us briefly describe all the activities in the table, in row order:

The first row describes activities that only require an implementation, but not a model program. We perform *trace capture* to collect traces from an instrumented implementation. In *offline testing*, the framework's test runner tool uses synthesized traces to test an implementation. We perform *trace replay*, a variant of offline testing that uses captured traces instead of synthesized traces.

The second row describes activities that only a require a model program, but not an implementation. The framework's *offline test generator* creates a synthesized trace from a model program. Another two activities use traces to validate model programs. To perform *a priori validation*, the analyzer checks a model program with traces constructed "by hand" from a written or implicit specification. *Empirical validation* checks the model program with captured traces. Empirical validation is similar to *passive testing*, where log files are checked against a model program. However, the purpose of empirical validation is to check the model program, while the purpose of passive testing is to check the traces.

The third row describes activities that require both a model program and an implementation. *On-the-fly* testing executes the model program and the implementation during the test run in order to choose actions and check the returned results. *Run-time verification* is a variant of on-the-fly testing which similar to passive testing. It considers all actions to be observable, and checks the observed behavior for conformance. Both activities generate a trace. The test outcome is computed during the test run, but the trace is still useful for failure investigation.

The last two columns in the third row are empty. We are not aware of any tools or activities that cover these combinations.

It is informative to consider the table in column order. Traditional offline testing and *a priori* validation occupy the second column for synthesized traces, as is appropriate for checking conformance to a written or implicit specification. Our trace replay and empirical testing occupy the third column for captured traces, as is appropriate for checking portability.

## 9 Related work

The idea of using samples of observed behavior (traces) in assurance activities has many precursors. As far as we know, we are the first to use captured traces as offline test suites in a model-based testing framework, or to use captured traces to validate model programs that are used for on-the-fly testing. There are some similar predecessors in the model-based testing community, which we review here.

Most of the model-based testing literature (reviewed in [20, 13]) emphasizes offline test generation, offline testing, and on-the-fly testing; some also recommends *a priori* validation. These techniques have been used in many projects; a large example is by Grieskamp, et al. [12], who used these methods to confirm that Windows protocol implementations conform to their published specifications.

Mariani [15] and Ulrich and Petrenko [19] report on techniques similar to our trace capture followed by empirical validation. Mariani even used trace data for subsequent tests, as we do in our trace replay. But both works used information extracted from captured traces rather than the traces themselves, and used purpose-built software rather than a model-based testing framework.

Some work has explored the techniques at other entries in our Table 1. Barnett and Schulte [1, 2] and Colin and Mariani [6] report on run-time verification, using technology similar to our interposition. Lee, et al [14] discuss passive testing.

Xie and Notkin [21] proposed a model learning technique that uses test runs (like our captured traces) to infer and iteratively improve a model used for automatic test generation.

There are several popular programming languages that attempt to provide platform independence including Java, Flash, and JavaScript. Portability is most frequently tested via a set of unit tests which an implementation must pass. The use of unit tests is clearly not adequate. In the case of Java [11, 18], the portability issues are well documented [5, 9, 4]. JavaScript and Flash / ActionScript both have to contend with a different source of heterogeneity, the web browser. The functionality exposed by the web browser leads to portability problems which can be hard for developers or users to diagnose. For instance, Windows users running Firefox had issues playing videos when using versions earlier than Flash 10 [8]. Our approach to portability differs in that we use traces gathered from real programs running on deployed systems to validate the portability of the system. This allows strong verification of the deterministic portions of programs when running in real scenarios.

## 10 Conclusion

In this work, we used techniques from the model-based testing community to evaluate portability of implementations within the Seattle testbed. We define a goal called practical portability and then verify that programs we run meet this standard using program traces. We validate that any entirely deterministic program generates the same program trace across different implementations. This trace validation mechanism allows us to validate the sandbox's API either concurrently or independently from the interpreter behavior.

For programs that have non-deterministic calls, we describe techniques for using traces that can be used to validate the program behavior. We use traces gathered in production to drive the construction and validation of a model of the nondeterministic portion of the API. This model can be used across different implementations for on-the-fly testing.

This report describes the rationale and supporting techology for our approach. At this writing we have performed some preliminary experiments to establish the technical feasibility of this approach, but we do not yet have sufficient experimental data to report here. We are eager to learn whether testing with captured traces exposes portability errors that are not detected by traces generated from a model based on the written specification.

## References


[1] BARNETT, M., AND SCHULTE, W. Spying on components: A runtime verification technique, 2001. OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems.

[2] BARNETT, M., AND SCHULTE, W. Runtime verification of .NET contracts. *Journal of Systems and Software 65*, 3 (2003), 199–208.

[3] CAPPOS, J., BESCHASTNIKH, I., KRISHNAMURTHY, A., AND ANDERSON, T. Seattle: a platform for educational cloud computing. *SIGCSE Bull. 41*, 1 (2009), 111–115.

[4] CHANEZON, P. Write Once, Run Anywhere: the devil is in the details, Oct 2006. `http://wordpress.chanezon.com/?p=7`.

[5] CHARNY, B. Write once, run anywhere not working for phones, Jul 2005. `http://mcall.com.com/Write-once,-run-anywhere-not-working-for-phones/2100-1037_3-5788766.html`.

[6] COLIN, S., AND MARIANI, L. Run-time verification. In *Model-Based Testing of Reactive Systems* (2004), M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., vol. 3472 of *Lecture Notes in Computer Science*, Springer, pp. 525–555.

[7] Python Programming Language. `http://www.python.org/`.

[8] Flash 10 Released - Finally, Flash Videos In Firefox Work Again! `http://www.readwriteweb.com/archives/flash_10_released_finally_flash_works_in_firefox_again.php`. Accessed Jun 3, 2010.

[9] FRUHLINGER, J. LWUIT: Write once, run anywhere (mobile) (hopefully), Aug 2008. `http://www.javaworld.com/community/node/1113`.

[10] FutureRepyAPI – Seattle. `https://seattle.cs.washington.edu/wiki/FutureRepyAPI`. Accessed April 15, 2010.

[11] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *Java (TM) Language Specification, The (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[12] GRIESKAMP, W., KICILLOF, N., MACDONALD, D., NANDAN, A., STOBIE, K., AND WURDEN, F. L. Model-based quality assurance of Windows protocol documentation. In *ICST* (2008), IEEE Computer Society, pp. 502–506.

[13] JACKY, J., VEANES, M., CAMPBELL, C., AND SCHULTE, W. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.

[14] LEE, D., CHEN, D., HAO, R., MILLER, R. E., WU, J., AND YIN, X. Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Trans. Netw. 14*, 2 (2006), 424–437.

[15] MARIANI, L. Behavior capture and test for verifying evolving component-based systems. In *ICSE* (2004), IEEE Computer Society, pp. 78–80.

[16] PyModel: Model-based testing in Python. `http://staff.washington.edu/jon/pymodel/www/`. Accessed March 23, 2010.

[17] Seattle: Open Peer-to-Peer Computing. `http://seattle.cs.washington.edu/`. Accessed April 3, 2010.

[18] Sun Java J2EE – Compatibility & Java Verification. `http://java.sun.com/j2ee/verified/`.

[19] ULRICH, A., AND PETRENKO, A. Reverse engineering models from traces to validate distributed systems - an industrial case study. In *ECMDA-FA* (2007), D. H. Akehurst, R. Vogel, and R. F. Paige, Eds., vol. 4530 of *Lecture Notes in Computer Science*, Springer, pp. 184–193.

[20] UTTING, M., AND LEGEARD, B. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[21] XIE, T., AND NOTKIN, D. Mutually enhancing test generation and specification inference. In *FATES* (2003), A. Petrenko and A. Ulrich, Eds., vol. 2931 of *Lecture Notes in Computer Science*, Springer, pp. 60–69.