Singularity Design Note

15

Testing

This design note first describes requirements on the design of Singularity to make it an environment which supports state-of-the-art testing. It then sketches the architecture of an API which could fulfill these requirements.

1. Introduction

Testing is a part of the software-development process which consistently has been treated as a stepchild in computer science. Over decades, academics concerned with correctness of systems have focused on formal verification techniques, which promise a complete approach to correctness instead of an approximate as it is provided by testing.

Only in recent years, the gap between verification and testing became smaller, on the one hand because verification approaches dropped formerly unquestioned requirements like completeness or even soundness, on the other hand since formal approaches have been more widely applied to testing itself, e.g. in the realm of model-based testing.

This design note aims at stating requirements on the Singularity design such that it supports the application of state-of-the-art testing techniques. It then sketches the architecture of the so-called *monitoring framework* which can fulfill most of those requirements.

2. Testing and Singularity

In order to understand what is needed in Singularity to support testing, it is useful to elaborate what we actually mean by testing. In general, we can say that testing is *driving a system through several execution scenarios and checking whether it behaves like expected.* In the subsequent sections, we break that down into the problems domains of *test configuration*, *test definition*, and *test execution*, and analyze the implications for the design of Singularity.

2.1. Test Configuration

Before we can run any kind of test, the “system-under-test” (SUT) needs to be described, configured and possibly deployed. The SUT can be a collection of applications, a single application, a process/component, a class/object, or just a single method. The SUT can run on a single node (machine), or on a network of nodes.

State-of-the-art in test configuration is represented by TTCN-3 – test configuration notation -- and the tools supporting it. TTCN-3 is a full-fledged programming language with constructs to launch processes over networks, deploy connections between them, monitor their interaction, etc. TTCN-3 is an ETSI standard (European Telecommunications Standards Institute) and in rising usage in the European telecommunication Industry. There is no US match for it.

Implications for Singularity Design

Application abstraction should exactly deliver us what we need for dealing with the problem of test configuration. It should be possible to setup an application manifest which reflects a particular test configuration of a collection of applications, a single application, a process/component, a class/object, or just a single function.

In particular, application abstraction should be able to setup a configuration for testing where most code comes from dependencies to other applications, and only some test control code is part of that "pseudo" application which's mere purpose is to test other applications.

When using application abstraction for testing, it might be possible that installation, updating, or un-installation are frequently called by some automated job (we are talking about a bandwidth of >100 installations per minute). If possible, these operations should be much faster as they are today under "Windows Installer".

2.2. Test Definition

A *test suite* defines which execution scenarios shall be tested when tests are executed. Traditionally, test suites are represented as sequences of inputs to the SUT, but testing of reactive, concurrent systems requires also considering asynchronous outputs, where the SUT may have the freedom to non-deterministically deliver different outputs in a given state. This lifts test suites from simple sequences to non-deterministic state machines, which can be e.g. represented by labeled transition systems or I/O automatons.

The inputs and outputs find in the representation of test suites may be build from messages send over channels, events like entering/exiting a method call, locking/unlocking a resource, etc. Inputs and outputs are not just labels, but terms which are built from the vocabulary of the SUT, including names of (potentially internal) methods, and parameters, given as ground values build from (potentially internal) types of the SUT, as well as free variables which are going to be bound to entities observed during test execution time.

In today's praxis of testing, the test suite is actually often given by a program, which encodes the state machine implicitly. These programs heavily rely on access to the internals of the SUT, which requires them either to be compiled together with the SUT, or using technologies like reflection to access the internals of the SUT. Alternatively, an application which is proactively "designed for testing" may explicitly expose hooks for letting the test engineer access its internals; however, this approach requires foreseeing all the needs to hook into the SUT, which is not feasible in praxis.

A test suite maybe designed manually, or derived from a test-generation technique, like model-based testing (MBT). The derivation of a test suite in MBT may happen "on-the-fly", i.e. interleafed with actual test execution, or offline. In either case, test derivation in MBT is based on exploring the state space of the model, by techniques similar to explicit state model-checking or symbolic model-checking. Tools like MSR's Spec Explorer allow the underlying model to be given in a C# programming style (actually, Spec#), and directly integrate into the domain of the SUT (using its types and methods). Exploration is done by execution on top of the CLR in these tools.

Implications for Singularity Design

Singularity must provide a way to define a test suite referring to internals of the SUT. Looking just at the external interfaces, are relying on that all necessary test hooks will be explicitly exposed, will not be sufficient (though this could be a better-then-nothing starting point).

In addition to enumerating methods, fields, channels and so on, it is also necessary to persist data in test suites (parameters of method calls). This calls eventually for an API similar to that provided by CLR reflection and CLR serialization. However, the API may be restricted to use for testing and debugging only, and not offered for meta-programming.

Regarding test derivation, in the long term it should be possible to run exploration and model-checking techniques for arbitrary MSIL code. A special "backtracking" runtime which includes symbolic computation should be deployable on top of Singularity's runtime (such a runtime is in development at MSR). The backtracking runtime should be able to pass-through some functionality – like garbage collection, kernel calls, etc. – to the underlying runtime. Whereas this kind of functionality can be relative easily implemented in off-the-shelf JVMs or CLRs by hooking into the class loader, the design of Singularity currently does not seem to provide such extensions of the runtime (it would be against the principles of Singularity to provide them).

2.3. Test Execution

Test execution means running a test suite in a given test configuration and analyzing and logging the results. It can be as simple as calling a function in the SUT and checking the return value, and -- in the presence of reactive, non-deterministic systems -- as complex as running a game strategy which tries to discover the behavior of the SUT by driving it to certain goal states, repeating the same experiment as long as some (usually stochastic) criterion is met.

Technically, test execution requires control of the SUT by feeding input to it and observation of the SUT by awaiting output. As discussed in the previous section, input and output can be messages on channels, as well as method calls on objects, or certain events like acquiring and releasing resources. Output might arrive in non-deterministic order and at any time at the test execution engine, which therefore usually buffers the output for consumption by the testing strategy.

Apart of observing the pure functional behavior as exposed in inputs and outputs, various other means are used in testing praxis to measure the success of testing. The most common is code coverage, which comes in the flavors of branch coverage and path coverage. Also, information about data access might be collected, as well as information about performance and memory usage.

Implications for Singularity Design

In order to execute a test suite, the test execution engine must be able to trigger inputs, i.e. send messages to (potentially internal) channels, and call methods (on potentially internal) objects. In order to observe output, there must be a mechanism which triggers notification of the test execution engine upon message arrival, method calls, and resource allocation. For realizing code coverage, according profiling mechanisms are needed. As for test definition, this eventually calls for an API as it is provided by CLR reflection (in order to trigger inputs). In addition, an API which provides the instrumentation of the SUT with probes and watches is needed.

3. Monitoring Framework

This section sketches a relative simple architecture which would allow realizing the requirements for advanced testing as sketched above. This is called the *monitoring framework.* As the name suggests, this framework could be not only used to realize testing, but also other applications like debuggers and tracing and profiling tools.

3.1. Basic Model

A monitoring application is a program which monitors other applications (e.g. for the purpose of testing). Monitoring applications run as normal user processes with particular access rights. They solely communicate with the kernel to obtain information, subscribe to events, and influence execution of other processes. Monitoring is thus a service provided on kernel level to processes. The kernel execution itself cannot be monitored.

3.2. Security Issues

Not every process can be monitored at every point of time. The fact that a process is monitored shall probably be reflected as a modifier on a process identity which changes dynamically at runtime. The change of identity reflects that the monitored process can behave differently than contracts have promised. A process should probably be able to reflect that it is monitored. Some processes might refuse to change their state from non-monitored to monitored once they started execution, since they might have accumulated confidential data during non-monitored execution time. Processes might refuse to initiate a communication channel or continue talking on an existing channel if the channels endpoint is in a monitored process.

3.3. Reflection

Since monitor applications do not know at compilation time about the structure of the programs they monitor, their needs to be a suitable data representation of this structure. This representation might or might not be aligned with user-level reflection support added to Singularity.

We can break down reflection support into static reflection (which just allows to enumerate the elements of a program) and dynamic reflection (which allows, at runtime, instrumenting those elements). While it might make sense to make a distinction here on the level of access rights, it is probably not advisable to provide two different APIs, since users are already acquainted with reflection APIs from the JVM and the CLR. Therefore it is suggested to have one common object model for static and dynamic reflection.

The reflection API needs to be able to enumerate the elements which constitute an application: types, fields, methods, properties, events, contracts and so on. This is supposedly very similar to the standard .NET reflection API.

In addition to standard reflection which exposes only declarations, they also need to be a way to represent program points (i.e. code), for installing watches (described later). For example, to monitor enter and exit points of method calls, or branches for coverage analysis, the program points associated with this code need to be addressed. It is not yet clear how to represent this information. One way is to expose the MSIL directly, another is to expose the logical structure (by means of a basic block control graph), and a third is to just enumerate source locations. In either case, it is crucial from the monitoring perspective that optimizing compilers can accurately calculate back from the generated code to the program point representation.

3.4. Probes

A *probe* is a piece of code which can be injected by a monitor application into a monitored process. A probe is defined by a Sing# statement which is compiled and executed "as is" it would live in the monitored process. The context of a probe can be process global (attached to a name space), instance based (attached to an object of a given class), or local (attached to a method). Within each context, the same language rules apply as if the probe was directly denoted in that context. For example, a probe attached to a method can access all global state, instance state (if the method is instance based), and local variables which are accessible for code written in that method.

A probe is not supposed to be able to change the state it accesses, however, for pragmatic reasons it might be desirable to allow probes to call into arbitrary methods of the process (following the context conditions of Sing#), which then can indirectly change state. Therefore, it might be more faithful to just assume that probes also can change state.

A probe is an object which is instantiated in the process in which it is injected. As that, the probe itself can have some internal state, like for example counters, which can be accessed from its defining statement.

A monitoring process can trigger execution of a probe in the monitored process whenever it has control over that process (i.e. when the monitored process is suspended). Probes are also used in watches (below) in which case their execution is triggered automatically.

3.5. Watches

A *watch* is a probe which is automatically executed when a given program point is hit. The probe's definition decides what happens in that case: it can suspend execution of the process, signaling a monitor application about that fact, or it can just maintain some summarization information defined with the probe, immediately continuing execution.

Watches, in combination with probes, are the silver bullet of the monitoring framework. They maybe used to realize observation of method enters and exists, code coverage, tracing, as well as conventional debugging functionality like breakpoints.

4. Conclusions

Testing and Singularity appear to be antagonistic. The requirements imposed by testing seem to largely contradict basic design principles of Singularity. This shouldn't come as a surprise. A test engineer would like to have full access to internals of implementations, driving and observing their behavior. The ultimate tester is indeed a hacker which tries to break a system (whether the attack is malicious or not does not matter, technically). Singularity tries to limit this kind of access.

It is possible to imagine a reduced scenario for testing under Singularity. This scenario would only include testing on communications observed on channels. Since channels are controlled by the kernel, and

communicate a restricted kind of data, interposition for testing could be easily implemented for them. While this scenario is not very interesting from a practical viewpoint on testing, it might provide a viable first start to install black-box, conformance testing in Singularity.

