Extensible Synthetic File Servers?
or: Structuring the Glue between Tester and System Under Test

(work in progress)

Axel Belinfante
University of Twente
Axel.Belinfante@cs.utwente.nl

ABSTRACT

We discuss a few simple scenarios of how we can design and develop a compositional synthetic file server that gives access to external processes – in particular, in the context of testing, gives access to the system under test – such that certain parts of said synthethic file server can be prepared as off-the-shelf components to which other specifically written parts can be added in a kind of plug-and-play fashion.

The approaches only deal with the problem of accessing the system under test from the point of view of offered functionality, and compositionality, but do not consider efficiency or performance.

The study is rather preliminary, and only very limited practical experiments have been performed.

1. Introduction and motivation

In our contribution to the first IWP9 in 2006 [1] we gave a description of the model-based testing [2] approach that we use, including an overview of our test tool architecture. We do model-based black box conformance testing of reactive systems. We replace the usual environment to the *system under test* (SUT) by one that is under control of the test tool. The test tool mimics the environment that the SUT expects and interacts with it. In each test step either a stimulus, obtained from the model, is applied (input is given) to the SUT, or an observation (output) is obtained from the SUT, and it is checked whether it was expected by the model - if not, an error is found.

The high-level architecture of the test tool is depicted in Figure 1.

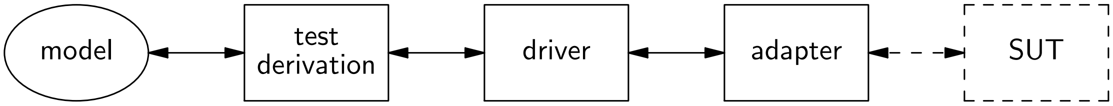

Figure 1: Complete tester with implementation under test.

The *test derivation* component computes from the model the stimuli that can be given, as well as the expected observations. The *adapter* provides the access to the SUT. The *driver* component provides the 'main loop' of the tester. It uses the test derivation component and the adapter to do its work. In our IWP9 2006 constribution we focused on the the test derivation component, and gave a decomposition of it. Now we focus on the adapter component.

To be able to interact with an SUT an adapter has to speak the right protocols or to implement the right API functions. In the models the interactions between SUT and its environment are described in an abstract way, and these abstract representations are also used on the interface between the adapter and the driver. Hence, the adapter also has be able to relate its actual interactions with the SUT to these abstract representations. Different models of the same system may use different abstract representations for the same real-world interaction. This makes the adapter specialised in two ways:

1. towards a particular SUT (or SUT family) with which it has to interact, and
2. towards a particular abstract interactions representation of the model (or family of models) of which the test cases are derived.

Whereas for all other test tool components in the architecture we can have generic implementations, we do not intend to have a single generic adapter implementation. That seems simply infeasable. Instead we would like to have an adapter framework that allows us to easily and quickly develop a specialised adapter for a case at hand by simply plugging together already existing components with newly developed SUT- or model-specific ones.

In order to make progress in turning this idea into reality we will define a generic interface between the adapter and the driver, and study a generic adapter architecture to identify those components for which we can create generic, reusable implementations, as well as those that typically will have to be specifically implemented. For this paper our aim is to study in particular how we can realize such a "plug and play" approach using the mechanisms that Plan 9 and Inferno offer us, and briefly look at existing end-user extentable programs in Plan 9.

1.1. Adapter interfaces

In Figure 2 we see that the adapter only interfaces with the driver and with the SUT. The adapter may have multiple interfaces with the SUT, depending on how the SUT interacts with its environment on the one hand, and depending on what kind of environment we, as tester, want to provide to the SUT on the other hand. These interfaces will vary from one SUT to another, or even with the same SUT, they may vary from one test setup to another one. We refer to these interfaces as PCOs (*Points of Control and Observation*).

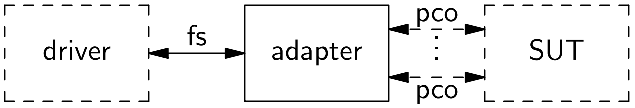

Figure 2: Interfaces of the Adapter.

We already decided on the nature of the interface that the adapter provides to the driver in [1]: it is a file system interface (*fs* in Figure 2). The file server interface consists of two files: write-only file *input* and read-only file *output*. When a stimulus in abstract form (of a test case) is written to *input* the adapter will issue the corresponding stimulus to the SUT (initiate the corresponding interaction with the SUT). When *output* is read the adapter returns an observation in abstract form that can be passed to the test derivation component, to be checked. The stimuli and observations passed over the interface are in the abstract model-specific representation.

1.2. Adapter functional decomposition

We now decompose the adapter. A high-level decomposition indentifies the following functionality in the adapter, as depicted in Figure 3:

- file server interface to driver (fs)
- handling of SUT interface at PCOs
- mapping between abstract representation used at driver interface and concrete representation used at PCO interfaces
- observation queue, to store observations until driver requests them

In Figure 3 we already made a further decomposition step: each PCO (each interface by which we access the SUT) has its own handler. We can similarly further decompose the mapping component, for example to have a separate mapping component for each individual PCO.

This decomposition allows us to make the following observations. Firstly, the interface to the driver can be provided by a generic adapter component. The data exchanged over that interface will vary, but the interface itself will not. Secondly, the particular PCO handlers are likely to be reusable, at least for those interfaces to the SUT where the interaction consists of exchanging messages. Thirdly, the mapping itself is likely to be specific, and thus not directly reusable. It may however be possible to have generic mapping functionality that is instantiated for a specific mapping.

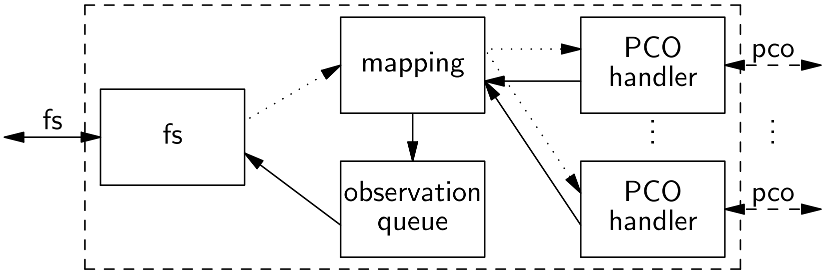

Figure 3: Functional Decomposition of the Adapter. For clarity we use dotted arrows for the flow of stimuli and solid arrows for the flow of observations.

1.3. Refining adapter functional decomposition

We look at how we can refine the functional decomposition along the following axes:

1. by deciding which methods of interaction we will support,
2. by splitting components such that data flows through them in only one direction,
3. by looking at concurrency and blocking,
4. by identifying multiplexing and demultiplexing components.

We discuss each of these separately below. Before we do that we first depict the resulting design in Figure 4. Three components in the Figure we only briefly mention here. Component *datagen* is used to build the mapping table represented by *datamap* in a demand driven (lazy) way while the test execution run takes place. Component *dynconfig* is used to dynamically reconfigure the adapter by activating or even adding PCO handlers and encoders and decoders in the course of a test execution run. Component *timer* is used to see if the system under test does respond within a reasonable amount of time to a stimulus that was given.

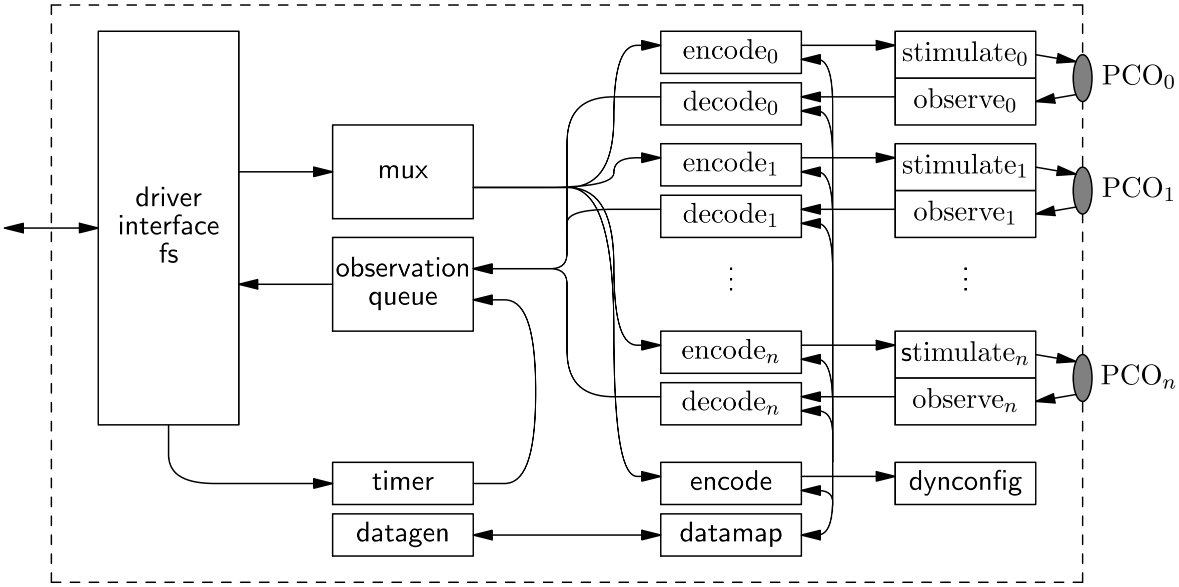

Figure 4: Full decomposition of adapter with per PCO separate translation (encode_i, decode_i) and interaction ($\mathrm{stimulate}_i$, $\mathrm{observe}_i$), a dynamic reconfiguration component (dynconfig) with its own translator (encode), a multiplexer that directs stimuli to the right translator (for PCO or for dynconfig), a queue to store observations, a timer to observe quiescence, a data map that holds the atomic data translation mapping, and a data generator that produces data values to extend the data translation mapping on demand.

1.3.1. Interaction methods

For now we only support message based interaction with the SUT, like via network protocols, or through the interaction with a program over its standard input and output. For a number of network protocols Plan 9 already has a uniform interface via the files under /net. We may be able to extend our approach to other interaction methods by using message based interaction with proxy components that participate in the non-message based interaction. We will not discuss this further here.

The functionality that we need for connection-oriented network protocols differs slightly from that needed for connection-less ones. The reason for that is that we typically associate a PCO with each established connection such that once the connection has been established we only need to pass the message data through the associated translation and interaction components. The connection information, like connection endpoint addresses, is only necessary during connection set up. For connection-less interaction we typically associate a PCO with each network access point of the tester such that the message data must be accompanied by at least the remote network access point address; the PCO will know its own address.

In case of interaction with a program, what corresponds to connection set up is starting the program. When we start a program we have to provide it with configuration information. The configuration information may need to contain information about network addresses for other connections that the program has to establish with its environment. If the addresses refer to network access points (PCOs) of the tester, their addresses may not be known until the network access points (PCOs) have been set up. What we learn from this is that we must have a means to specify the order in which PCOs are set up and configured.

1.3.2. Data flow

We refine the decomposition by splitting the single mapping component of Figure 3 into multiple ones, one per PCO, which we once more split according to the direction of the data flow. This gives us separate per-PCO encoders and decoders. The mapping that is built into these encoders and decoders is parameterized to such that test execution run time parameters, like network addresses or test data values, can be changed from one test run to another. Component *datamap* holds mapping tables for such run time parameters. The datamap component is consulted by the per-PCO encodeers and decoders, such that each of them uses the same parameter values.

Conceptually we also split each of the PCO handlers according to the direction of the data flow into separate stimulator and observer components.

1.3.3. Concurrency and blocking

In general, all handlers of our external interfaces must run independent from each other, to avoid the blocking of one making affecting the ability to interact for another. Thus, the PCO observer components need to be able to accept data from the SUT at all time, independent from each other, and from the interface to the driver.

1.3.4. Multiplexing and demultiplexing

We now have for each PCO a decoder that delivers observations to the observation queue. per-PCO mapping decoder. Conceptually thus the observation queue component contains demultiplexing functionality to pass the observations that it receives from the decoders one by one to the observation queue. For stimuli we have the opposite situation: a stimulus received from the driver must be handed over to the encoder for the right PCO. We add a *mux* component to do this.

2. Implementation scenarios

We now discuss various scenarios for the implementation of the components and their interfaces. This is where the “work in progress” part of this paper applies: our quest for the most appropriate approach in a Plan 9 environment is not over yet, although we believe that we are making progress.

We start by describing our current design, after which we discuss alternatives.

2.1. Single synthetic file server with plugin programs

We design the adapter as a single synthetic file server that is parameterized: the translation and interaction components – the components that typically vary from one test execution set up to the next – are not implemented in the file server itself, but provided to it in the form of separate plug in programs. The single file server provides the basic adapter framework. It implements the generic components of our design, i.e. the interface to the driver, the multiplexer and demultiplexer, and the observation queue. The plug in programs implement the translation functionality and the PCO handling. The plug in programs run concurrently by which they satisfy the requirement of blocking independence. For now the file server also implements the shared data map; in the future we may prefer to implement also that as a separate plug in program.

The translation programs work like filters that read messages that have to be translated from standard input and provide the result on standard output, and provide error messages and diagnostics on standard error. We have separate programs for encoding and for decoding. These programs need to be able to access the shared data map, as we indicated above. The data map may be constructed in a lazy (demand driven) way, i.e. its contents may be updated while a test run takes place. Those updates will only extend the map; they will not change or remove parts of the map. Because of the updates we can not simply provide the map as a file that is read by an encoder or decoder during start up. Instead of the map itself we provide an interface to the map. The interface takes the form of a single file to which data map translations requests can be written and from which results can be read. The single file is synthesized by the adapter file server, and the name of the file is passed as command line argument to translation plug in programs when they are started by the synthetic file server.

The interaction programs also interface with the synthetic adapter file server via their standard input, output and error. We have a single program for each PCO: the program provides stimuli and obtains observations. What configuration information the interaction programs need is connection kind specific.

The synthetic adapter file server provides the connection between translation plug in programs and interaction plug in programs. The synthetic adapter file server provides the synthetic files that are used for the interface with the driver. In addition it provides a file to which configuration commands can be written. The exact form of the configuration interface is still to be decided. The configuration commands start given plug in programs with given command line arguments, and to set up the infrastructure (a pipe) to connect them together. This interface is intended to be used for the initial (pre-start) configuration of the adapter, as well as for dynamic reconfiguration.

Discussion This approach was partially inspired by cmd(3) of Inferno.

We have chosen the encoding and decoding components to be separate, concurrently running components that wait until work is provided to them in the form of a message to process, process the message and produce corresponding output, and wait for the next message. Alternatively, we could have made them sub-routines that are only invoked to do a particular task. However, that would have made it impossible to extend the adapter with new translation functionality without recompiling the PCO handlers.

An important consequence of the choice we made is that the order in which interactions take place at the PCOs (the order in which the adapter receives messages from the system under test) is preserved as the order in which observatons go through the observation queue. Because both PCO handler and mapping component run concurrently, it is imaginable that some PCO handler and decoding mapping component combinations run faster than others. It is then also imaginable that an interaction a that takes place before another interaction b is processed slower than interaction b such that b reaches the observation queue before a. This issue is for the moment left for further study.

Advantages of this approach are the following. Firstly, we use compositionality on the level of programs, not on the level of code modules, which means that we can reconfigure without having to recompile. Secondly, the synthetic adapter file server only needs to set up the pipes over which the plug in programs will interact. It does not have to move data around between those programs itself.

Disadvantages of this approach are the following. Firstly, the essential interaction goes via standard input and output of the plug in programs which means that both data and control (configuration) messages go over the same interface. We can separate these using the approach of acme(1) [3]: let the synthetic adapter file server provide each of the plug in programs with multiple synthetic files, one for each different flow of information. However, in that case the synthetic adapter file server also has to process these files (transport the data between the components).

2.2. Multiple identical synthetic file servers with plugin programs

The single synthetic file server approach that we discussed in the previous section deals with configuration in a top-down manner: we first start the synthetic file server, we then tell the synthetic file server which plugins to start, and how they should be connected. This approach is good for actual deployment, but for experimentation a more bottom up approach would be useful.

With the bottom up approach we initially start the PCO-handler programs. With each PCO-handler program we start a dedicated *muxview* synthetic file server by which we interact with the PCO-handler. Such muxview offers access to the standard input, output and error of its connected PCO-handler via the synthetic files that it serves.

A muxview synthetic file server combines two functionalities. Firstly it "translates" between access via standard input, output and error of a running program and access via files visible in the namespace. Secondly, it multiplexes access to the programs standard output resp. standard error to multiple readers, and the programs standard input to multiple writers. Each reader sees the same data. The data that is written to the programs standard input can be monitored by multiple readers via a separate synthetic file.

When a muxview file server is started it is given a program with command line arguments to start. It starts the program with the given arguments, with the standard input, output and error connected to pipes of which muxview holds the other ends. Because of the constraints regarding concurrency and blocking the muxview file servers will consist of multiple processes: two reader processes that read data from the programs stdout resp. stderr pipes and passes it to a middle man that enqueues it until the user is ready for it, a file server proces that reads 9p requests from the user, and passes them on to the middle man which either processes them immediately and passes back responses, or enqueues the requests until the data needed to service the requests comes in. This approach was not invented by us, but adapted from the sshnet implementation.

For the combination of muxview and encoder or decoder we need a uni-directional version of muxview. For the encoder muxview only needs to provide access to its standard input and standard error, and muxview should connect the standard output of the encoder to a given file. This given file will typically be a writeable file of a muxview that is connected to a PCO-handler. For the decoder muxview only needs to provide access to its standard output and standard error, and muxview should connect the standard input of the decoder to a given file. This given file will typically be a readable file of a muxview that is connected to a PCO-handler.

We can extend muxview to make data available in multiple formats, like for example a 'raw' format containing the data 'as is' as well as a hexadecimal representation.

Discussion The ability to have multiple readers, and the ability to add new readers while the program is already running allows us to step by step construct the components structure – i.e. on top of the encoders and decoders we have the mux component and the demux+observation queue – with the ability to eavesdrop at every connection between the components. This eases debugging of the components, and may even help to reverse engineer the precise format of the interaction messages of the system under test.

Initially we designed the muxview file server functionality as part of individual PCO handler components. The advantage of that approach is that we get a uniform interface for the PCO handler components which provides better decoupling of translation and interaction components. This may lead to reduced interfacing dependencies between PCO handlers and encoders and decoders, and thus potentially to easier component reuse. However, it turned out that most of the complexity of the resulting components was in the multiplexing file server support, and the actual PCO handling code was only a fraction of the code of the entire component. Because of that we changed the design and extracted the multiplexing file server functionality as a separate reusable building block: muxview.

For the unidirectional version that we need for the encoder and decoder the "given file" that we connect to standard input or standard output of the program started could be the standard input resp. standard output of muxview itself.

Instead of a single synthetic file server that gives bidirectional access to standard input and output of a program we could also consider a simpler file server in the spirit of "tee" that only provides unidirectional access. We have not thought this through in detail.

2.3. Single monolithic synthetic file server

A less compositional approach is to build the entire adapter component as a single monolothic synthetic file server. As we have seen, several of the functional adapter components need to be able to run concurrently. For this we use the Plan 9 threading library. So, the functional components will be mapped onto threads and procs, and their interfaces are formed by messages sent over channels.

Discussion Advantages of this approach are the following. We can implement those sub components that receive data from the environment of the adapter (i.e. from the driver and from the SUT) as separate processes such that they can block when waiting for data. The components of the adapter can pass data over channels. By having a single multi-threaded implementation we can reduce the number of times that data is copied, by passing pointers around over the channels.

A disadvantage of this approach is that the reusable components consist of pieces of C code, and thus our intended "plug and play" involves piecing together pieces of C code – we prefer reusable components that can be reused at the shell level as described with the previous approaches.

An alternative could be to use Inferno and provide the components as separate Inferno modules. We have not looked into this yet.

3. Related work

We are aware of the following extension mechanisms offered by existing Plan 9 programs. We have not studied whether Inferno has additional features to support this.

synthetic file system Acme(1) provides a synthetic file system to client (helper) applications. The helper applications are started indepedently, outside the control of acme(1).

pipe to standard input and output of programs Httpd(1) communicates with clients over their standard input and output. The clients are started by httpd(1). Also the juke(7) player component playlistfs(7) uses external programs, in its case to decode the audio file formats.

configuration file and synthetic file systems The plumber(4) [4] reads from a configuration file which helpers potentially are available, and how they can be started when not yet available. The helper programs may be started outside the plumber(4), but in addition the plumber(4) will start them when needed. Each helper application implements a synthetic file server over which it is accessed.

4. Discussion

For the PCO handler programs the implementation as separate synthetic file servers, either using muxview together with plugin programs, or using natively implemented file server programs, is probably the most versatile, also because it allows exploratory manual interaction with a system under test before the mapping (encoding and decoding) component is implemented. This resembles a small part of the plumber(4) approach. However, the file server interface overhead of our approach – even when extracted into separate component muxview – is rather large. The use of filter-style plugin programs leads to a more balanced design. Moreover, if we really need it, we can regain some of the file server interface style flexibility by using muxview together with the plugin programs.

For the encoder and decoder components that essentially only transform data the simpler pipe-style filter approach is probably sufficicient, and therefore more suited, because implementing pipe-style filters is less involved than implementing a synthetic file server. For the coordination between multiple encoder and decoder components, in particular when using on-the-fly created mappings, the most natural implementation for a coordination component consists of a synthetic file server. This could be the same synthetic file server that provides the driver access, or it could be a specific one only for mapping (coding) coordination.

5. Conclusion

We have discussed a number of scenarios to construct a multi-component synthetic file server in a compositional way.

The simplest solution – using the thread library – offers the least isolation between the components, but also has the least implementation overhead for the inter-component interfaces.

The more involved solutions where the individual components are implemented as separate programs offer greater separation between the components, and allow greater implementation freedom for the individual components. However, this may come at the cost of increased interfacing code size. Extracting the core interfacing functionality in a separate reusable component partially eleviates the problem.

Due to time constraints these conclusions are mostly based on thought experiments and only a rather small implementation experiment: two small implementations of synthetic file servers that give access to udp, and to the standard input and output of a forked-off program. Both offer access to their “thing” using the same file interface. In these programs the code that gives access to udp, resp. the forked-off program is small compared to the synthetic file server implementation code. This is in part caused by the decoupling between the data producer (respectively udp, and the standard output of the forked-off program) and the file server access to it, as well as the ability present in these synthetic file servers to make the same data available to multiple readers in either “as is” (raw) or hexadecimal format.

Acknowledgements Francisco Ballesteros provided valuable suggestions for improving the initial draft, and pointed me in the direction of cmd(3) of Inferno.

References

[1] A. F. E. Belinfante. Experiments towards model-based testing using plan 9: Labelled transition file systems, stacking file systems, on-the-fly coverage measuring. In G. Guardiola, E. Soriano, and F. J. Ballesteros, editors, *Proceedings of the First International Workshop on Plan 9, Madrid, Spain*, pages 53–64, Madrid, December 2006. Universidad Rey Juan Carlos.

[2] M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer Verlag, 2005.

[3] Rob Pike. Acme: A user interface for programmers. In *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pages 223–234. USENIX, 1994.

[4] Rob Pike. Plumbing and other utilities. In *USENIX Annual Technical Conference, General Track*, pages 159–170, 2000.