Experiments Towards Model-Based Testing Using Plan 9: Labelled Transition File Systems, Stacking File Systems, On-the-fly Coverage Measuring

(work in progress)

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

ABSTRACT

We report on experiments that we did on Plan 9/Inferno to gain more experience with the file-system-as-tool-interface approach. We reimplemented functionality that we earlier worked on in Unix, trying to use Plan 9 file system interfaces. The application domain for those experiments was model-based testing.

The idea we wanted to experiment with consists of building small, reusable pieces of functionality which are then composed to achieve the intended functionality. In particular we want to experiment with the idea of 'stacking' file servers (fs) on top of each other, where the upper fs acts as a 'filter' on the data and structure provided by the lower fs.

For this experiment we designed a file system interface (ltsfs) that gives fine-grained access to a labelled transition system, and made two implementations of it. We developed a small fs that, when 'stacked' on top of the ltsfs, extends it with additional files, and an application that uses the resulting file system.

The hope was that an interface like the one offered by ltsfs could be used as a general interface between (specification language specific) programs that give access to state spaces and (specification language independent) programs that use (walk) those state spaces like simulators, model checkers, or test derivation programs.

Initial results (obtained on a less-than-modern machine) suggest that, although the approach by itself is definitely feasible in principle, in practice the fine-grained access offered by ltsfs may involve many file (9p) transactions which may seriously affect performance. In Unix we used a more conservative approach where the access was less fine-grained which likely explains why there we did not suffer from this problem.

In addition we report on experiments to use acid to obtain coverage information that is updated on-the-fly while the program is running This worked quite well. The main observation from those experiments is that the basic block notion of this approach, which has a more 'semantical' nature, differs from the more 'syntactical' nature of the basic block notion in Unix coverage measurement tools like tcov or gcov.

1. Introduction and Motivation

In the world of formal methods and tools one tries to study and analyse systems using models of these systems. For example, by simulating the models to study the behaviour of a system. By looking for the presence of wanted properties or checking for the absence of unwanted ones (model checking). By deriving test cases that allow checking (testing) whether the implementation of a system conforms to its model (specification). This last activity, testing, is the focus of our attention [3, 11].

Those models, or specifications, are written in various modeling or specification languages[1] Examples are LOTOS [8], Promela [7] or SDL [5].

Even though these languages differ from each other, their meaning (semantics) is commonly expressed using labelled transisition systems (LTSs). The fact that those kind of languages share this common formalism to represent their behaviour is something that can be exploited

[1]We will from now on use the word specification language, for simplicty.

when building software tools for them, like the simulators, model checkers etc. The idea is to isolate specification language specific details in specification language specific tools, and implement generic, specification language independent algorithms only once in other tools. Instead of building a simulator for each specification language one can instead build a single simulator that can simulate a labelled transition system, together with translaters that provide labelled transition based access to the specification languages one is interested in.

The CADP toolkit [1] uses this approach. It defines a C interface (API) that is used between providers of labelled transition systems and their users. This interface is called the Open/Caesar interface [6]. CADP comes with a number of (specification language specific) tool components that provide this interface for several specification languages (among which is LOTOS), and with generic tools like a simulator, a state space generator, etc. that use the Open/Caesar interface. A user can extend the toolkit both by providing the Open/Caesar interface for additional specification formalisms, and by using the interface to implement additional (specification language independent) tools.

In Unix we have used this same approach of separating specification language specific functionality from generic specification language independent functionality in our prototype toolset for model-based testing [11]. In that toolset we did not restrict ourselves to a single interface between specification language specific LTS generating functionality and our specification language independent test derivation implementation. We used the Open/Caesar API to connect to CADP, to be able to access specifications in LOTOS. For a few other specification languages we made our own tools to provide the labelled transition system. These tools do not provide the Open/Caesar C API. Instead, they are self-contained programs that read a specification from file after which a simple textual 'protocol' on their standard input and output can be used to access the labelled transition system. The (specification language independent) test derivation tool would then simply fork off a labelled transition system provider and engage in the textual protocol to access the LTS.

This approach worked well.

The aim of our work in Plan 9 (and Inferno) is to experiment with this same approach of separating specification language specific functionality from generic specification language independent functionality, but not via a C API or a textual protocol over a pipe but via a file system interface.

We see various advantages of this approach.

One specific advantage is that we hope it to be more extensible: the C interface only has opaque concepts 'state' and 'label', and it is not possible to 'refine' that. Also in the textual protocol we describe each transition with a fixed set of properties. In the file system approach we hope to be able to have an extendable interface, in which we can add more detail corresponding to a particular 'view' on the LTS, by 'stacking' file systems on top of each other where the 'higher' file systems extend the lower one with additional files. The user just sees the end result, and need not be aware of the stacking. In Section 4. we give an example of this approach.

The other advantages are not specific to our application area, but are the general advantages of using a filesystem as interface. We are no longer programming language dependent. In particular, being able to access the interface from the shell using echo and cat makes it easier to just 'play' with an lts. In addition, it makes it easy to gradually built more complex tools, by first prototyping functionality from the shell, and then when needed reimplement it, for example in C. We can easily run different pieces of funtionality on different machines.

We also see potential disadvantages to this approach. We will never be as fast as when using a C API. Because we want to offer fine-grained access, we intend to split the information of many different files (not unlike upas/fs does) but a consequence is that to access this information the user will have to walk, open, read, close many different files, many of which will only be visited once.

2. Preliminaries

2.1. Labelled Transition Systems

A labelled transition system consists of states and transitions. It can easily be depicted in a graph where the nodes are states and the edges between nodes are transitions. The transitions have labels attached to them (hence the name: *labelled* transition systems). Formally they

consist of a 4-tuple $\langle S, L, s_0, R\rangle$ with S a set of states, L a set of labels, s_0 the initial state and $R \subset S \times L \times S$ the transition relation.

The following coffee machine is simple example. It allows the user to enter a coin into the machine, after which the user should press a button to get coffee. If the user waits too long, the machine performs an internal (unobservable, invisible) transition (timeout) and refunds the coin. Usualy we use the special label τ (tau) to denote internal (invisible) transitions.

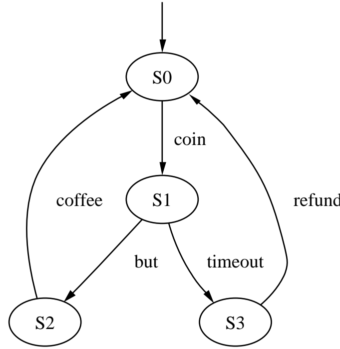

Figure 1: The labelled transition system of a simple coffee machine.

So, in this example $S = \{S0, S1, S2, S3\}$, $s_0 = S0$, $L = \{coin, but, timeout, refund\}$ and $R = \{(S0, coin, S1), (S1, but, S2), (S2, coffee, S0), (S1, timeout, S3), (S3, refund, S0)\}$.

2.2. Implicit and Explicit Transition Relation

In the example above we explictly gave the labelled transition system, including the transition relation. Many specification languages do not describe the transition relation so explicitly (some do). In many cases we can only gradually *unfold* the transition system by following the outgoing transitions of those states that we already have seen (initially the start state).

This is reflected in the interfaces. The Open/Caesar interface has functions to access the start state, and to iterate over the outgoing transitions of state.

In our interface we will have something similar.

3. The Labelled Transition File System Interface

3.1. Requirements

We recapitulate the requirements on the interface (some we did not mention before). The interface must give the user a the start state. It must allow the user to gradually unfold the labelled transition system. It must allow the user to access the labels of the transition. It must be able to tell the user whether a transition is visible or invisible (internal to the system, like the timeout in Figure 1) (this is new). The user should be able to tell it that certain states or transitions are no longer of interest (which should allow freeing resources associated with those) (this is new).

One requirement is related to performance. Since we use this only as an interface on top of the real state and transition representation, its resource usage must be within reason. In particular, the memory overhead should be small.

3.2. Design Considerations

As should be obvious, we have chosen to design the interface as a file system.

We might have chosen to use a directory structure that resembles the tree that we get when we unfold a labelled transition system from the root, but we have not. We did not do that for two reasons. Firstly, that would mean that we can not remove internal nodes from the tree, because without them we can not reach their subtrees or leaves. Secondly, it would be difficult to represent those transitions that close a cycle, like the transitions $coffee$ and $refund$ in Figure 1.

3.3. Directory Structure

We have chosen to use directory structure a bit like the numbered connection directories present in the protocol directories of ip(3).

At the toplevel we have files `ctl` and `stats`, and for each transition a numbered directory.

For specification languages for which the transition relation is implicit we will initially have only one such numbered directory: for the initial transition. While we explore the LTS we create more of those directories. If, however, a specification language has an explicit transition relation we may choose to immediately create numbered directories for all transitions.

A numbered directory represents not only a transition but also the state that we reach by that transition. It contains files that allow the user to access properties of the transition (`label`, `visible`) and files that allow access to the outgoing transitions of the state (`succ` and `heir`, and indirectly `bro`).

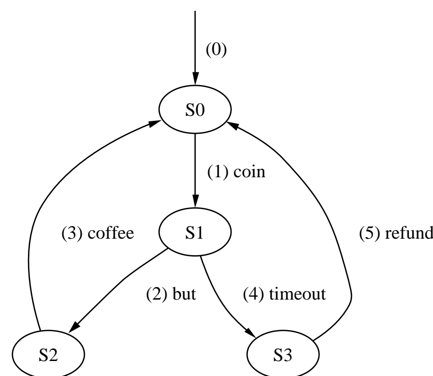

Figure 2: LTS of a simple coffee machine with transition numbers.

Files nr/`label` and nr/`visible` return, when read, respectively a string representation of the label and of the visibility (0 or 1) of transition nr.

File nr/`succ` lists, when read, the outgoing transitions of the state reached by transition nr (in a space-separated list).

Files `heir` and `bro` give an alternative way of unfolding the transition system. File nr/`heir` gives the firstborn son of the state reached by transition nr i.e. the first element of the result of nr/`succ` (empty when nr has no children). File nr/`bro` gives the brother of transition nr i.e. a next element of the list returned by `succ` (empty when we exhausted the list).

We use the numbering of Figure 2 to illustrate `succ`, `heir`, and `bro`. File 1/`succ` gives "2 4". File 1/`heir` gives "2". File 2/`bro` gives "4". File 4/`bro` gives "" (the empty string).

Where `succ` is mostly useful for breadth-first exploration of the labelled transition system, `heir` and `bro` are slightly more general and do not impose a particular exploration strategy.

3.4. Initial Transition

A 'special' case is the initial transition: the transition at the top of Figure 1, without 'source' and without label, that points to the initial state ($S0$).

The toplevel `ctl` file, when read, returns the number of the initial transition.

3.5. Multiple Incoming Transitions

Above we wrote that we have chosen not to use a directory structure that resembles the LTS (tree) unfolding structure because it would be difficult to represent cycles, but we did not tell how we do represent cycles. We will tell now.

Actually, we have to deal with a slightly more general case than only with cycles. We have to deal with states that have multiple incoming transitions, which may also be present in a

non-circulair LTS, as illustrated by state $S2$ in Figure 3.

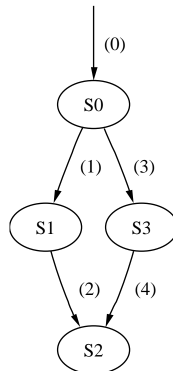

Figure 3: LTS with state with multiple incoming edges (labels omitted).

The solution we take is to treat the first transition that reaches a state as the *canonical* transition that points to that state (and use the number of that transition as the canonical number of the state). The user can access this information via one additional file in each numbered directory: `elsewhere`.

When read, file nr/`elsewhere` returns the number of the first transition other than nr that reached the state that is reached by nr (rephrased: the number of the canonical transition that reached it).

So, for the example of Figure 3 all nr/`elsewhere` files return "" (the empty string), except for 4/`elsewhere` which returns "2".

For the example of Figure 2 both file 3/`elsewhere` and file 5/`elsewhere` give "0". Files 0/`succ`, 3/`succ` and 5/`succ` all return "1". However, files 0/`label`, 3/`label` and 5/`label` return respectively "" (the empty string), "coffee" and "refund".

Our choice has one clear disadvantage, which becomes apparent when the user is allowed to remove states and transitions from an LTS: in that case the user may choose to loose the canonical reference (transition) to a state (which would make it definitely much less *canonical*).

3.6. Removal of States and Transitions

One of the requirements we gave above was the ability for the user to indicate that certain states or transitions would no longer be referenced, which should allow a labelled transition filesystem to free resources associated with those states or transitions.

A natural way to indicate loss of interest in states or transitions is the removal of the numbered directory that gives access to it.

3.7. Implementations

We have two implementations of this interface for Plan9 in C.

The first implementation gives access to a LOTOS specification, using the Open/Caesar interface. We have linked it together with specification-specific C code that was generated from a LOTOS specification by CADP (this C code needed minor tweaking to be accepted by the Plan 9 C compiler).

For the implementation we used lib9p. Initially we used the file trees of lib9p, but this used too much memory (for some 4200 transitions it would take approx. 9 Mb). Nevertheless, the use of file trees made it easy to quickstart an initial implementation. We reimplemented it without file trees, which made more adequate use of resources (a little less than 2 Mb for those 4200 transitions. This can be reduced further by sharing the label strings (right now we allocate a new string for each transition label) and using our own pool allocator for the state and transition data structures to avoid malloc overhead).

The second implementation gives access to specifications in the Aldebaran language of CADP. This is a textual format that closely resembles the structure of an LTS. It was rather easy to derive the implementation for Aldebaran from the one without file trees for LOTOS.

We implemented removal of states and transitions only in our initial file trees based implementation for LOTOS. We did not implement this (yet) in our reimplementation of the LOTOS ltsfs without file trees, nor in the implementation for the Aldebaran language.

Earlier we had already made an alternative implementation for Aldebaran in limbo.

4. Extending The Labelled Transition Interface

4.1. Introduction

So far we have discussed the labelled transition file system interface. It is based on the formal LTS definition we presented above, where we have a single set L containing all the labels.

In the testing theory that we use ([10]) we do not use a single set of labels, but we split the labels in two groups: the input labels (L_I) and the output labels (L_U) (internal action τ is special and does belong to neither input nor output, formally $L = L_I \cup L_U \cup \{\tau\}$). The input labels represent stimuli that the tester can send to the system under test. The output labels represent observations that the tester can get from the system under test.

This means that there is a small gap between the information provided by the labelled transition file interface, and the information needed for our test derivation tool component. We can fill this gap in various ways.

One solution is to provide the test derivation tool component with a parameter: a map (or mapping function) that, when given a label, tells whether it is an input or an output label. This is the approach that we took in our Unix toolset.

Another solution is to extend the file system interface, such that the test derivation tool component can read an additional file in each numbered (state/transition) directory to see whether the label belongs to input or output.

We want to use this second approach. We want the file system interface to provide a separate file named `iokind` in the numbered state/transition directories. File `iokind` contains either the string `input`, the string `output`, or the empty string, for respectively input labels, output labels, and internal actions.

However, we do not want to change the labelled transition file server we already made – it does not care about input or output labels, and we don't want to 'pollute' it with this distinction. (Actually, there are specification languages that already make the distinction between input and output on their language level. A labelled transition file server for such a language could (should) of course directly offer `iokind` files.)

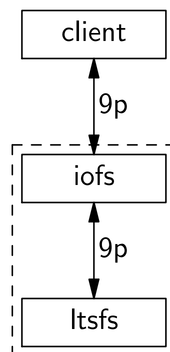

Figure 4: Stacking iofs on top of ltsfs.

What we want to develop is a file server (lets call it iofs) that sits between the user and the real labelled transition file server and extends the file system offered by the labelled transition file server with those `iokind` files. The intermediate file server (iofs) should work as a filter on the 9p 'message stream' between client and server, as illustrated by Figure 4. It should keep as little state as possible. It should *not* keep a local copy (modified by local changes like

inserting `iokind` files) of the file tree (with or without the actual files) in server. With the very many small files that we have to deal with for the labelled transition filesystem keeping a local copy of the file tree would be too expensive (memory wise).

This approach is very similar to the approach taken by trfs [2] and substfs [4]. Also trfs and substfs sit between a user and a real file server and extend (change) the view of the user on the real file server. The main difference is that trfs and substfs change file names but do not change the number of files that the user sees, i.e. with trfs and substfs there is a one to one mapping between files that the user sees and files on the real file server. With iofs this is different: the `iokind` files do not exist on the real file server, but have to be 'invented' by iofs.

This approach differs from the approach taken by libolfs [9], because libolfs seems to keep its own copy of the structure of the file tree on the server (which we don't want to do). Another difference is that libolfs access the (files on the) server via normal operations (read, write, open, close) on the mounted file system instead of talking 9p directly.

4.2. Requirements

File server iofs gets on its command line a (connection to the) file server it has to extend and regular expressions that tell it whether a particular label string belongs to 'input' or to 'output'.

In general, it should forward its requests to the underlying file server, and forward the results of the underlying file server to the client. Some preprocessing and some postprocessing may be necessary.

When the client reads a numbered directory iofs should get the directory entries from the server and add a directory entry for the `iokind` file.

When the client stats a $nr/$`iokind` file iofs should return the same directory entry that it would return for `iokind` in a directory read of nr.

When the client reads a $nr/$`iokind` file iofs should read the label from the $nr/$`label` file of the underlying file server, apply the regular expressions to compute the correct result and return it to the client.

The iofs file server should present the client a consistent set of qids. The `iokind` files should all have distinct qids, that are different from the qids of the other files.

4.3. QID Structure

Compared to trfs and substfs, the difficulty here is to provide the client with a consisten qid space, even when we add files. We need a way to 'invent' new qids for the files that we will add, such that we do not cause conflicts with the files already there.

The solution we took is the following. We assume that the highest bits of the `path` field of the qids of the underlying server contain zeros, so it is safe to bit-shift these a few positions 'upwards', to 'create' some free bits for ourselves to play with. (we can instrument our code with assert statements to check/enforce this assumption). How many bits should we shift? In our case, one could say that for every $nr/$`label` file we want to add a $nr/$`iokind` file, so shifting one bit would be sufficient. Assuming function qid gives us the qid of a file, then $(qid(nr/\texttt{label}).\texttt{path} << 1)$ could be used to refer to the original $nr/$`label` file, and $(qid(nr/\texttt{label}).\texttt{path} << 1)|1$ can then be used to refer to the new $nr/$`iokind` file.

An important advantage of this approach is that the resulting qid space should allows further stacking, using the same approach, because we only add bits to the low end, where we already assumed the bits to be.

How do we make sure that we have the qid of the $nr/$`label` file of the underlying server when we need it? We need it in two cases.

Firstly, when we read (the directory entries of) a numbered directory. In that case we make sure that we know that we are reading a numbered directory (i.e. we are aware that we walked to it, see below for details). We forward the read requests to the underlying fileserver, and as usual, after postprocessing the replies, forward these to the client. In this case postprocessing involves two additional activities. It involves recognizing the directory entry for the `label` file and remembering its qid. It involves also the following. When the underlying file server indicates that all directory entries have been read (by returning a 0 read result) iofs synthesizes a directory entry for the `iokind` file and returns it. This will cause the client to issue one

more read request, at which point iofs does return a 0 read result. The directory entry for the `iokind` file is synthesized using the qid of the `label` file that was remembered earlier.

Secondly, when the client issues a walk request on a nr/`iokind` file. In that case iofs preprocesses the request into a walk to nr/`label` before forwarding it to the underlying file server (and remembers that the original request involved nr/`iokind`). The underlying server returns its qid of nr/`label` in the walk reply, and iofs can associate it with the client fid, to use it to generate the correct qid for a subsequent stat or open request.

4.4. Implementation

We implemented this using lib9p. We mainly use its fid support. For each fid we keep the following information in an Aux struct:

- a string path (together with its size)
- a boolean flag isnrdir
- a boolean flag isiokind
- a boolean flag iokindadded
- a Qid ioqid

We use these fields as follows.

String path represents the current path for this fid in the underlying filesystem. We update it during after each succesful walk in the underlying file system and canonicalize it using cleanname(2).

Flags isnrdir and isiokind are updated when we process the walk message that we get from the client, by trying to match two regular expressions on a copy of the path, updated with the path elements in the walk message and canonicalized using cleanname(2). These flags are used when handling subsequent requests on the fid, like open, read, and stat.

Flag iokindadded and Qid ioqid are used when we process read requests on a numbered directory. Qid ioqid is used to store the qid of the nr/`label` file of the underlying server, which we extract from the directory entries returned in the read replies of the underlying server. We use ioqid when we generate the directory entry for the `iokind` file. Flag iokindadded is used to remember whether or not we already returned such `iokind` directory entry to the client.

We structured the main service loop of the implementation like the one of trfs [2]. Essentially it consists of the following steps:

1. get a client 9p request
2. do some work based on the request message type
   (allocate or lookup a fid, update its data, tweak the 9p request)
3. send the (possibly tweaked) 9p request to the server
4. get the 9p reply from the server
5. do some work based on the reply message type
   (update fid data, tweak the 9p reply (rewrite its qids, tweak data))
6. send the (tweaked) 9p reply to the client

We keep the 9p request that we get from the client and (after possible tweaking) forwarded to the underlying server until we have processed the 9p reply from the underlying server, because we may need both request and reply when processing the reply (at least for the walk message, where we need the names of the 'items' walked).

In our case each message that we get from the client results in a message that we send to the underlying server, even in the case of a read of a nr/`iokind` file because then we read the corresponding `label` file from the underlying server. In general that need not be the case. It is very well possible that a client request can be answered without contacting the underlying server (in our case the last read of a numbered directory need not go via the underlying server). It is just as well possible that to handle a client request our server needs to interchange multiple messages with the underlying server.

This makes it not so easy to see exactly what library support could be offered to ease construction of these kind of file servers, since the obvious candidate library function (the service loop function) may need to be customized for the particular application.

For our special case where each client request results in a corresponding request to the underlying server this would be easier. The library could offer a service function that is structured like ours and that uses two sets of user-supplied processing functions: one set to process client requests, and one to process underlying server replies.

5. A Client for Test Derivation Primitives

Now that we have access to the labelled transition system and know whether the label of a transition represents input or output we can use this to compute what we need for testing. This is the tool component that we usually call the *primer*.

For each stage during the testing we need to compute the set of input labels that represents stimuli that the tester can send to the system under test, and the set of output labels that represent the observations that we expect. We call these sets the input menu respectively output menu. With each label in a menu we associate the set of states that we can reach from the states of the stage by following transitions with that label.

Each testing stage corresponds with one or more states of the labelled transition system (if there is more than one this is due to non-determinism).

To compute the menu for a stage we obtain the outgoing transitions of each state in the stage, either using the `succ` file for the state, or using the `heir` and `bro` files. For each of these transitions we read the `elsewhere` file to check whether there is a canonical number for the state reached by the transition, and the `visible` file. If a transition is not visible (internal) we also have to take into account the outgoing transitions of the state reached by the internal transition. If a transition is visible we read the `label` and `iokind` files to compute the menus.

We go from one testing stage to the next by giving a stimulus or doing an observation. The label corresponding to the stimulus or observation must be in the input resp. output menu of the stage. The states for the next stage are those that were associated with the stimulus resp. observation label. If these states correspond to an earlier stage our next stage will actually be that earlier one (we have detected a cycle).

If we do not take for the next stage all states associated with the stimulus resp. observation label, but only one of those states (randomly chosen), we get a simulator program that simulates the specification. We can test a simulated specification just like a real implementation.

The primer is a separate program that takes as argument the directory on which the iofs is mounted. It offers its functionality also via a (small) file system.

Files `inputs` and `outputs` present, when read, the corresponding menus. Files `input` and `output` present, when read, a randomly chosen element from the corresponding menu. To files `input` and `output` a label string can be written. If the label is valid (in its menu) the write will succeed and a transition to the next stage is done and new menus are computed. Otherwise the write fails and no transition is done. File `states`, when read, present the stages and their corresponding states. File `state`, when read, presents this information for the current stage. To switch to one of these stages without doing a transition the stage number can be written to file `state`.

5.1. Implementation

Also this file server was implemented using lib9p. The implementation is rather straightforward.

6. Completing the Tester

The work we described so far provides the basics for model based testing. We need two more components to complete the test tool: the (generic) *driver* which implements the 'main loop' of the tester, and the (implementation under test (IUT) dependent) *adapter* which provides the driver with a uniform interface to the IUT. The function of the adapter is twofold: to provide the actual connection to the IUT using those interfaces that the IUT offers, and to translate between the abstract labels with which we represent stimuli and observations in the specification and the concrete messages used on the actual interfaces of the IUT. Figure 5 depicts the components of the complete test tool.

The adapter provides a file system as interface to the driver. This interface consists of two files: write-only `input` and read-only `output`. When a stimulus (as obtained from the primer) is written to `input` the adapter will issue the corresponding stimulus to the IUT. When `output` is read the adapter returns an observation in abstract form that can be checked by the primer.

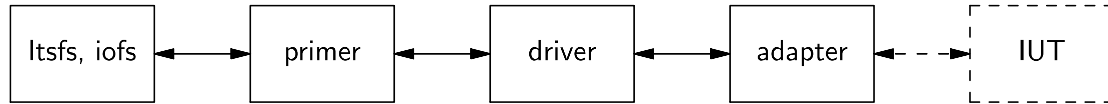

Figure 5: Complete tester with implementation under test.

An adapter that is simple to implement is one that provides the adapter interface to the simulation variant of the primer (giving access to a *simulated* implementation under test).

We implemented a simple driver and a simulation adapter to be able to do at least some testing. The driver 'forever' performs test steps until an error is found or the user stops the testing. For each test step either a stimulus is given or an observation is checked. The choice between stimulating and observing is made at random. When stimulating the driver obtains a (randomly chosen) stimulus from the primer, passes it on to the adapter and passes it back to the primer to make it proceed to the next testing stage. When observing an observation is obtained from the adapter and passed to the primer for checking – if the observation is expected the primer proceeds to the next testing stage.

7. Experimental Results

We experimented in two ways. Initially we did not yet have the driver program, and thus only experimented with the combination of ltsfs, iofs and primer. Once the driver and simulation adapter were ready we compared several 'toy' spefications (in Aldebaran) of simple coffee machines.

7.1. Without Driver and Adapter

We have used the 'stack' of file servers (ltsfs, iofs and primer) to just by hand (using cat and echo) look at the input and output menus and make transitions.

Functionality-wise all worked fine.

Regarding performance we were less succesful than we hoped for (we did perform the experiments on a relatively old 333 Mhz AMD K6-2 based machine with 256 Mb, and realized too late that it could have been nice to do the same experiments on a more modern machine).

All was well with the Aldebaran examples, which were rather small toy examples.

With the LOTOS example computation of the menus took noticible more time, already when the number of states corresponding to a testing stage was small. However, in the LOTOS example the number of outgoing transitions for each state was much larger (approx. 60) than in the Aldebaran examples (at most 5). When, due to internal transitions, the number of states of a stage grew, already quickly the menu computation took longer than we found acceptable.

The reason seems to be the large number of files that is accessed in the labelled transition file system during menu computation. This suggests that we used a too fine-grained access, and might have been better off by letting the `succ` file not only give transition numbers, but also directly the properties label, visible and elsewhere. This would be a direct translation of the textual interface that we use in Unix.

Generally speaking this means that instead of using a separate file for each state or transition property, we then use for each state a single file that holds the transition properties of all its outgoing transitions and the state properties of their destination states (except the outgoing transitions of these destination states). However, adding properties (e.g. the 'iokind' information) in a compositional way might be more involved than with the fine-grained access that we experimented with.

Time limitations unabled us to further explore this, at the time of this writing.

7.2. With Complete Tester

We have used the complete tester to compare several toy specifications of simple coffee machines in Aldebaran. The tester was able to detect unexpected outputs (observations) from the (simulated) implementation under test.

8. Measuring Implementation Coverage on-the-fly

We have experimented with on-the-fly coverage data generation using acid, based on the acid coverage library. Like acid libraries truss and trump it outputs diagnostic information, in this case about basic blocks reached, while the program is running. This worked quite well.

In our current version of this, the diagnostic is only printed when a block is 'hit' that was not covered before. This allows on-the-fly updating of a plot that shows the percentage of basic blocks covered. An alternative would be to print a 'basic block identifier string' (e.g. the 'address' of the corresponding source code) each time the block is 'hit'. In Unix we tried this (not using acid) and it allowed producing even more insightful plots. We will have to do the experiment to see how it will affect the speed of the running program.

The main observation from our experiments is that the basic block notion of this approach, which has a more 'semantical' nature, differs from the more 'syntactical' nature of the basic block notion in Unix coverage measurement tools like tcov or gcov. In particular, with the acid-based approach the start and end of a function do not automatically imply the start and end of a basic block.

8.1. Implementation

We implemented this by modifying a copy of the acid coverage library.

As distributed acid does contain a 'print' builtin command, but no 'flush' builtin. We added one to be able to force timely diagnostic output. Much later we found, thanks to acid libraries trump and truss, that we can 'misuse' the acid builtin 'stop' to force a flush.

Apart from the changes implied by the description above, we also experimented with reducing the list of all basic blocks found by the acid coverage library to only those that correspond to source files of interest. Otherwise we will also get diagnostics about the basic blocks in all library code that is part of a program. The approach we tried was to only take those blocks of which the source address of the start of the block corresponded to a source file of interest.

9. Conclusions

We designed a file system interface, that offers fine-grained access to a labelled transition system and made two different implementations of it.

We designed and implemented a file server that extends the labelled transition file system with new files. This extending file server does have knowledge of the file system hierarchy of the labelled transition file system. The scheme that we used to assign Qids for the extending file server allows further 'stacking' in the same way, because it only uses 'low' (little end) bits of the Qids. So, the assumption that the all the information is in the 'low Qid bits', and that the 'high Qid bits' are zero remains valid.

We implemented a client (the primer) for the extended labelled transition system file system. It derives test primitives and can also be used as simulator. This client allowed us to evaluate this approach of dividing the functionality over various programs that interface via file systemes.

In principle, the labelled transition system approach works well and offers access at the right level of abstraction. The fine-grained access obviates the need for parsing data read from the server in the client.

For practical purposes the access may be too fine-grained. (Or our machine for doing the experiments may be too old and too slow.) The fine-grained nature scatters the information over many files of the server. The large number of file operations that the client needs to access the information seem to have a negative impact on performance. A slightly less fine-grained alternative, with which we have good experiences in Unix, is suggested to alleviate the problem.

We implemented the remaining components (driver, adapter) that we need to do (simple) model-based testing. The resulting tester allowed us to compare several (very simple) specifications with each other using testing: it did detect those (simulated) implementations that did not conform to the specification.

Regarding the printing of coverage information acid proved to be a very flexible tool.

10. Future Work

It may be worthwhile to add the more coarse-grained labelled transitions interface to implementations we described here, to be able to make the comparsion.

Since we already have the implementation to test, and the ltfs for the LOTOS specification, it would be interesting to implement an adapter for that implementation, which would allows us to repeat the LOTOS part of the experiment described in [3]. An interesting question will be whether the uniform Plan 9 interface to resources, and interfaces like dial(2) and announce(2), may ease the construction of such adapter.

Extension of the interfaces may be necessary. If the adapter has to return information to the driver about the application of a stimulus a natural extension could be to make its file `input` also readable, let the driver open it in read-write mode, write the stimulus to it, and read back the information. For animation purposes it may be helpful if we can associate identifiers of visual objects (nodes, edges) with states and transitions of the ltsfs, and these identifiers are somehow also made available through the primer to the driver. This would allow us to animate a visualization of the specification (as for example Figure 1 or Figure 2) by highlighting (changing the color of) nodes and edges as testing proceeds.

References

[1] A toolbox for "construction and analysis of distributed processes". http://www.inrialpes.fr/vasy/cadp/. 1.

[2] F. J. Ballesteros. trfs.c. /n/sourcesdump/2006/0523/contrib/nemo/sys/src/cmd/trfs.c. 4.1., 4.4.

[3] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12th *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer, 1999. 1., 10.

[4] Axel Belinfante. substfs. /n/sources/contrib/axel/substfs. 4.1.

[5] CCITT. Specification and Description Language (SDL). Recommendation Z.100, ITU-T General Secretariat, Geneva, 1992. 1.

[6] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In B. Steffen, editor, *Fourth Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, pages 68–84. Lecture Notes in Computer Science 1384, Springer-Verlag, 1998. 1.

[7] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall Inc., 1991. 1.

[8] ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard IS-8807. ISO, Geneva, 1989. 1.

[9] KIDA Satoshi. libolfs. http://www.tip9ug.jp/who/kida/, /n/sources/kida/libolfs.tgz. 4.1.

[10] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996. 4.1.

[11] J. Tretmans and H. Brinksma. Torx: Automated model based testing. In A. Hartman and K. Dussa-Ziegler, editors, *Proc. 1st European Conf. on Model-Driven Software Engineering*, 2003. 1.