Validating Specifications for Model-Based Testing

Submitted to SERP 2008

Pieter Koopman
Software Technology
Radboud University Nijmegen
The Netherlands
Phone/Fax: +31 24 3652(483/525)
Email: pieter@cs.ru.nl

Peter Achten (contact)
Software Technology
Radboud University Nijmegen
The Netherlands
Phone/Fax: +31 24 3652(483/525)
Email: P.Achten@cs.ru.nl

Rinus Plasmeijer
Software Technology
Radboud University Nijmegen
The Netherlands
Phone/Fax: +31 24 3652(644/525)
Email: rinus@cs.ru.nl

Keywords: functional programming, model-based testing, validation tools, quality of specifications

***Abstract*—In model-based testing the behavior of a system under test is compared automatically with the behavior of a model. A significant fraction of issues found in testing appear to be caused by mistakes in the model. In order to ensure that it prescribes the desired behavior, it has to be *validated* by a human. In this work we describe a tool, esmViz, to support this validation. Models are given in a pure, lazy functional programming language. esmViz provides an interactive simulator of the model, as well as diagrams of the observed behavior. The tool is built on the iTask toolkit which results in an extremely concise GUI definition. Experiments show that esmViz helps to gain understanding of a model and to detect and remedy errors.**

I. Introduction

In *model-based testing* the behavior of a system under test, sut, is compared automatically with the behavior of its specification. Examples of model-based test tools are G∀st [5], QuickCheck [2], TorX [9], T-Uppaal [7]. The specification is a possibly non-deterministic state transition system used as model in the tests. The number of states, inputs and outputs can be infinite. The sut is assumed to be a state transition system with a hidden state. One can only apply inputs to the system and observe the corresponding output. Key advantages of model-based test tools are the significant reduction of the amount of manual testing; increase of test speed due to automation; and reuse of specifications for regression testing.

Model-based test systems execute a finite number of traces. For each trace the sut and the specification start in their initial state. An input is selected that is covered by the specification, it is applied to the sut, and the allowed states of the specification are computed. If, during this process, the test system discovers that no states are reachable for the specification, then the sut has shown behavior that is not covered by the specification. In test jargon it is said that an *issue* is found.

Ideally, each issue indicates an error in the sut. However, in practice a significant fraction of issues appear to be caused by problems with the specification: it does not correctly capture the intentions of the users and the sut does something different. Even though the fraction of issues depends on a lot on factors such as the kind of system and the effort spent in creating the model, we estimate that the specification has to be blamed for about 25% of the issues.

Incorrect specifications are a problem for several reasons. First, if an issue is found it is not clear whether we have to blame the specification or the sut. Finding and correcting errors in the specification takes time during the test phase of the project. Second, errors in the specification are only found during model based testing if the behavior of the sut differs from the specified behavior. Third, any change in the specification during the testing phase can cause major implementation changes to the sut. Finally, any change in model or sut invalidates in principle all previous test results. Hence, errors in the specification can be very expensive and it is worthwhile to invest effort to ensure its quality.

In the model-based test system G∀st the pure, lazy functional language Clean serves as specification language. Due to its high abstraction level it is possible to write concise specifications which contributes to their quality. It allows the test engineer to model arbitrarily large state, input, and output domains exactly as desired. The advantages have been presented earlier ([4], [6]). The Clean compiler checks quality aspects like type correctness and consistent definition of used identifiers. Other quality aspects such as the reachability of states, determinism and completeness, and the preservation of constraints can be checked by systematic testing [**?**].

The use of a high level specification language does not rule out the possibility that the specification prescribes the wrong behavior in a consistent way. Hence, these kinds of errors can not be found by the above mentioned techniques. In order to ensure that the specification prescribes the desired behavior, it has to be *validated* by a human. In this work we introduce the tool esmViz to support validation of G∀st models. This *simulator* enables the user to execute the specification. Such an interactive execution appears to be more illustrative than reviewing the specification. Second, it is possible to record the traces of the specification executed in the simulator. The states visited and their transitions can be visualized in an expanded state transition diagram. Since the type of states, inputs and outputs can be infinite and different in each and every specification, doing this conveniently is not straightforward. The key to the solution is to use generic definitions such that operations on these types can be derived instead of defined manually.

The layout of the paper is as follows: in Sect. II we introduce the concepts and notation that will be used throughout this paper. In Sect. III we discuss the issues that arise when testing against a formal specification. In Sect. IV we describe esmViz. Its implementation is discussed in Sect. V. Related work is discussed in Sect. VI. We present user experiences in Sect. VII, and conclude in Sect. VIII.

II. Model-based testing

In model-based testing the test tool compares the observed behavior of the system under test, sut, with the model in order to judge the correctness of the behavior. Any deviation of the observed behavior of the sut from the behavior allowed by the model is called an *issue*. In this section we review the models used by the model-based test tool G∀st.

The models used by G∀st for testing state based systems are extended state systems, ESMs. An ESM consists of some initial state

s_0 and a set of transitions of the form $s \xrightarrow{i/o} t$. In such a transition s is the source state, i is the input triggering this transition, o is the output of the system associated with this state and input, and t is the target state of the system. The sets of possible states S, possible inputs I, and possible outputs O of the ESM can all be infinite. The i/o combination is also called the *label* of the transition from s to t.

A *trace* $s \overset{\sigma}{\Longrightarrow} t$ is a sequence of labels. The empty trace contains no labels. If we have a trace $s \overset{\sigma}{\Longrightarrow} t$ and a transition $t \xrightarrow{i/o} u$ we can construct the trace $s \overset{\sigma;i/o}{\Longrightarrow} u$. If we are not interested in the target state, we will occasionally write $s \xrightarrow{i/o} \equiv \exists t.s \xrightarrow{i/o} t$ and $s \overset{\sigma}{\Longrightarrow} \equiv \exists t.s \overset{\sigma}{\Longrightarrow} t$. All traces from a given state are defined as: $\mathsf{traces}(s) \equiv \{\sigma | s \overset{\sigma}{\Longrightarrow}\}$. The init of a state s is the set of inputs i, such that there is an output o and target state t in the ESM such that there exists a transition $s \xrightarrow{i/o} t$. The after of a state s is the set of possible target states t, reachable after the given trace σ: $s \,\mathsf{after}\, \sigma \equiv \{t | s \overset{\sigma}{\Longrightarrow} t\}$. We overload traces, init, and after for sets of states instead of a single state by taking the union of the set members.

A. Conformance

In model-based testing we try to determine conformance of the sut and the model called spec. The sut is assumed to be a transition system, but treated as a black box: one can observe its traces, but not its internal state. During tests, all observed traces of the sut have to be traces of the specification to say that the sut *conforms* to the specification. Formally, this relation is defined as:

$$\begin{aligned} \mathsf{sut}\ \mathit{conf}\ \mathsf{spec} \quad \equiv \quad & \forall \sigma \in \mathsf{traces}_{\mathsf{spec}}(s_0), \\ & \forall i \in \mathsf{init}(s_0\, \mathsf{after}_{\mathsf{spec}}\, \sigma), \\ & \forall o \in O. \\ & (t_0\, \mathsf{after}_{\mathsf{sut}}\, \sigma) \xrightarrow{i/o} \Rightarrow\ (s_0\, \mathsf{after}_{\mathsf{spec}}\, \sigma) \xrightarrow{i/o} \end{aligned}$$

Here s_0 is the initial state of spec, and t_0 the initial state of sut. Intuitively the conformance relation reads: if the specification allows input i after trace σ, then the observed output of the sut should be allowed by the specification. If spec does not specify a transition for the current state and input, anything is allowed. Because the sut is a black box, its initial state t_0 is generally not know explicitly. We assume that the sut is in this abstract state when we switch it on, or we reset it.

Limiting the applied inputs to the init of the states of the current traces allows for partial specifications spec.

B. Testing Conformance

The conformance relation defined above covers all traces. Most interesting systems contain cycles, so traces can become infinitely long. Due to the possible infinite types for input and output, there can be even infinitely many traces of finite length. It is clear that in general a test system cannot prove conformance by executing tests. The test system G∀st approximates the conformance of the sut to the model by executing a finite number of traces of finite length.

To increase efficiency the test system records the set of allowed states, $s_0 \,\mathsf{after}\, \sigma$, rather than the trace σ. If at some point in the test this set of states becomes empty we have found an *issue*: a trace that shows that there is no conformance between sut and the model. Clearly this way of testing is *sound*, each trace leading to an issue during testing shows that there is no conformance between the sut and the model. This way of model-based testing is also *complete*, if there is no conformance between sut and the model, there are one or more traces indicating this. Such a trace can be found by testing (if the allowed length during tests is sufficiently large).

C. Representation of the transitions

To represent the ESM in the model-based test tool G∀st we need a finite (preferably small) and flexible representation, even if the set of transitions is infinite. Furthermore it should be easy to determine the init of the set of actual states, or to determine if an input is in this set, since this information is needed before we can apply an input during model based testing. The crucial step is to use a *function* to model the transitions rather than a data structure containing individual transitions. Each function alternative with variables in its patterns captures a family of related transitions. As usual lists represent sets. To define init easily we use specifications of type $S \times I \rightarrow [\mathit{Trans}\ O\ S]$.

A basic assumption in G∀st is that a transition always contains a sequence (list) of output symbols. This gives some additional flexibility as well as a suitable notation for no output (the empty list). Usually it is most convenient to specify the sequence of outputs and the target state in a transition. However, the number of allowed output sequences for one input can get huge, which makes it infeasible to state them explicitly. For instance in an authentication procedure a typical step is to ask for a challenge (the input), the response is a 64 bit number. Listing all possible outputs and target states explicitly requires 2^{64} transitions. In such a situation we prefer one function of type $[O] \rightarrow [S]$ rather than all individual transitions. Here the list of states as result has the usual meaning: all states (zero or more) that correspond to the given output sequence. Again, a single function captures a family of related transitions. In Clean these types are:

```
:: Spec s i o :== s i → [Trans o s]
:: Trans o s  =   Pt [o] s | Ft ([o] → [s])
```

Note that we use type parameters to allow any concrete type to be used for state (s), input (i), and output (o).

1) Example: As an example specification we show the model of a beverage vending machine that supplies coffee and tea (see Fig. 1). Initially the machine is in a state called `Off`. After the input `SwitchOn`

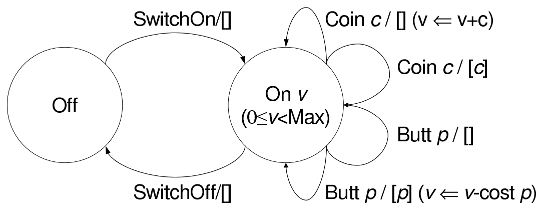


Fig. 1. The intented specification of the beverage vending machine

it enters state `On` 0 without producing any output. The integer in this state is used to record the amount of money inserted. Now the user can either insert a coin with a value given as parameter as long as the counter in the state remains less then `Max`, or press a button to receive a product. If there is enough money the user gets his product and the value of the counter is decreased accordingly. The types used in this model are:

```
:: Money   :== Int
:: State   = Off           | On Money
:: Input   = SwitchOn      | SwitchOff
           | Coin Money    | Butt Product
:: Product = Coffee        | Tea
:: Output  = Cup Product   | Return Money
```

A possible specification is given as the function `vSpec` below. We deliberately introduce some errors and strange transitions in this specification, later we return to it in an attempt to find these problems.

```
vSpec :: !State !Input → [Trans Output State]
vSpec Off SwitchOn  = [Pt [] (On 0)]
vSpec s   SwitchOff = [Pt [] Off]
vSpec (On s) (Coin c)
  // condition should be s+c<Max
  | s<Max     = [Pt [] (On (s+c))]
  // output should be Return c
```

```
               = [Pt [] (On s)]
  // pattern should be (Butt Coffee)
vSpec (On s) (Butt coffee)
  | s≥20     = [Pt [Cup Coffee] (On (s-20)),Pt [] (On s)]
vSpec (On s) (Butt Tea)
  // we get Coffee instead of Tea
  | s≥10     = [Pt [Cup Coffee] (On (s-10))]
  // do nothing for other buttons
vSpec (On s) (Butt p) = [Pt [] (On s)]
  // otherwise: nothing defined
vSpec state input = []
```

This specification is partial (e.g. the effect of pressing a product button when the machine in the state `Off` is not defined), and nondeterministic (if there is enough money in the machine and the user asks for coffee, the machine either produces coffee, or does nothing at all). Non-determinism models limited knowledge of the state of the real machine: e.g. if there are coffee beans it will produce coffee, otherwise it cannot produce coffee and waits for a new command.

III. Issues found in model based testing

Issues are traces that show that there is no conformance between the sut and the specification. Ideally each issue found indicates an error (bug) in the sut, but that is not always the case. Other sources of issues are inaccuracies in the model, problems in the interface between the test system and the sut, and internal faults in the test tool. One wishes to eliminate these other sources of issues before actual testing starts.

In ordinary automatic testing the test tool executes a manually specified or recorded trace. As a rule of thumb test engineers say that 40% of the issues found in this kind of tests indicates a real error in the sut. A tiny fraction of these issues is caused by the test tool itself, or the interface with the sut. Most issues are caused by the fact that the trace used does not correspond to the current version of the specification, or the specification itself is incorrect.

In model-based testing the traces are generated automatically and on-the-fly from the specification. This guarantees that the traces used during the tests always correspond to the current specification. As one expects this implies that a larger fraction of the issues found indicate errors in the sut. In our experience about 75% of the issues found during model-based testing indicate errors in the sut. The fraction of actual errors depends on the amount of effort spent on making a high quality specification, the quality of the informal specification and requirements used as basis, and the size and complexity of the system.

The specification is a Clean function, hence the compiler can readily check relevant properties: **i)** are all used identifiers properly defined, **ii)** is the entire specification type correct, **iii)** are all alternatives (transitions) reachable. Still, well typed specifications can go wrong. The problems with specifications that cannot be detected by the compiler can be divided in the following classes.

1) Relevant behavior of the system is not covered in the specification. Since the test system is carefully designed to handle partial specifications, this cannot be detected. Missing parts of the behavior are not covered in the tests.
2) The specification contains design errors. Typically a family of transitions is too large or too small, or leads to the wrong target state. If the sut does a better (or at least different) job, the test system will notice the difference if an appropriate trace occurs and hence reports an issue. Consider the alternative for `vSpec (On s) (Coin c)` in the example of the previous section. The wrong condition and forgotten return of money if the state becomes too large are probably design errors.
3) The transitions are designed correctly, but the implementation is incorrect. A typical example is the use of lowercase identifiers (variables) where an uppercase identifier (constructor) is intended, or vice versa. Another source of problems is copy-paste programming used to define similar transitions, where not all necessary changes are made. In our example this occurs in the transitions for `vSpec (On s) (Butt coffee)`, and `vSpec (On s) (Butt Tea)`.

All these problems result in well typed models. If the implementation is based on such a model, it is not possible to detect the problems by testing. Nevertheless, they must be found and preferably before model-based testing starts.

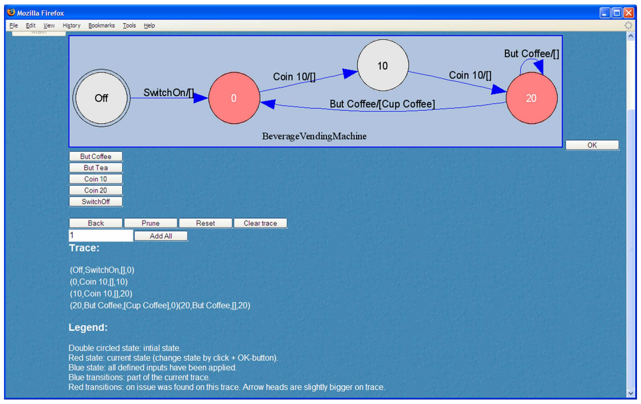
Various approaches to find these kind of problems are: **Inspection or reviews of the specification.** Problems can be found by manual inspection of the specification. As the model tells the whole story, there is nothing that prevents these errors from being detected by reviewing the code. However, due to their subtle nature, they might be missed.
Model checking. If we have the right properties and the specification is available in a form suited for model checking, the problems can be found by model checking. Limiting factors are the availability of the model in a form suited for a particular model checker, and the availability of properties to check. If the problems are known we can often find such a set of properties quite easily, but that is too late. In our example we can require: **p1)** every transition preserves the amount of money, **p2)** the amount of money in the machine is always less then `Max`, and **p3)** if we receive a product, it must be equal to the requested product. Finding a complete set of properties that reveals all problems is in general quite tricky.
Testing properties of the specification. Properties on transitions can be tested by the logical branch of G∀st. The advantage is that everything can be done within the same framework, especially the Clean specification function can be used as subject of tests. The drawback is that testing gives less certainty for large systems (although for small specifications the logical test system provides a proof by exhaustive testing).
Validation by simulation. The specification can be used as basis for an interactive simulation. With some effort the simulator not only displays the current transition, but also depicts the state space that is covered in the current simulation. Such a simulation can reveal that (important) parts of the behavior are missing, as well as problems with individual transitions. This requires a thorough observation of the shown behavior. Since the state space is discovered step-by-step by the user, the chances of finding the problems are quite good.

Each of the above methods can in principle find problems in the specification, but none of them can guaranteed this. Each method either requires human spotting of problems, or human formulation of properties revealing the problems.

In the remainder we describe esmViz. It combines model-checking of properties on transitions with validation by step-wise simulation. Together with G∀st, this covers a broad range of tools to investigate the quality of models.

IV. Validation of Specifications with esmViz

In this section we describe the web browser-based simulation tool, esmViz, that we have created to determine the quality of specifications. The tool also gives an impression of the behavior specified by the model, and checks user defined predicates on the transitions encountered. Simulation is useful to give non-experts a good impression of the specified behavior. The GUI of esmViz is a screen with the following elements (Fig. 2**(a)**): **1.** A list of found issues. The list is empty in Fig. 2**(a)**. **2.** The explored model as an Extended State Diagram (ESD). **3.** Within the ESD the set of possible active states determines the inputs that can be given. These are enumerated as buttons that the user can press to advance the system one step. In Fig. 2**(a)** the active states are $S = \{0, 20\}$, and init $(S) = \{\texttt{ButtCoffee}, \texttt{ButtTea}, \texttt{Coin10}, \texttt{Coin20}, \texttt{SwitchOff}\}$. **4.** Commands for navigation purposes, resetting the exploration, and so on. **5.** The current trace, as explained in Sect. II. Here the trace

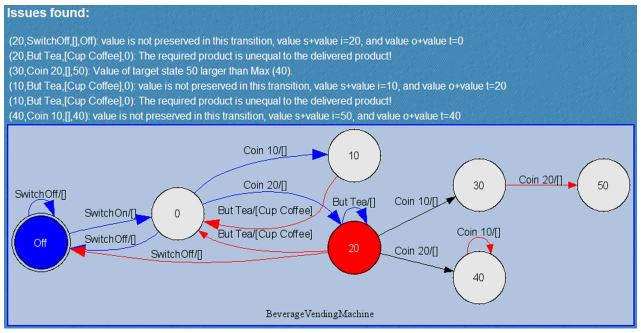


Trace:

(Off,SwitchOn,[],0)
(0,Coin 10,[],10)
(10,Coin 10,[],20)
(20,But Coffee,[Cup Coffee],0)(20,But Coffee,[],20)

Legend:

Double circled state: intial state.
Red state: current state (change state by click + OK-button).
Blue state: all defined inputs have been applied.
Blue transitions: part of the current trace.
Red transitions: on issue was found on this trace. Arrow heads are slightly bigger on trace.



Issues found:

(20,SwitchOff,[],Off): value is not preserved in this transition, value s+value i=20, and value o+value t=0
(20,But Tea,[Cup Coffee],0): The required product is unequal to the delivered product!
(30,Coin 20,[],50): Value of target state 50 larger than Max (40).
(10,But Tea,[Cup Coffee],0): value is not preserved in this transition, value s+value i=10, and value o+value t=20
(10,But Tea,[Cup Coffee],0): The required product is unequal to the delivered product!
(40,Coin 10,[],40): value is not preserved in this transition, value s+value i=50, and value o+value t=40

Fig. 2. **(a)** The validation tool in action with the beverage vending machine case. **(b)** ESD showing some of the issues in the beverage vending machine.

has length 4. **6.** Finally, a legend that tells what the elements of the rendering are.

The tool esmViz creates an ESD of the behavior encountered during simulation which is rendered as a directed graph. In ESM diagrams a parameterized state is drawn as one state, in the ESD a state is created for each value of the parameters encountered during simulation. In the beverage vending machine example the states (On 10) and (On 20) are different in the ESD, but they are one state in the ESM (Fig. 1). A transition $s \xrightarrow{i/o} t$ is rendered as an arrow between state s and state t, and has label i/o at its edge.

A. The ESM description

The ESD is created by esmViz based on an ESM and instances of generic functions used for instance to display and compare values of the data types used for states S, input I and output O. The ESM as described in Sect. II is a Clean value of type (ESM S I O):

```
:: ESM s i o = { s_0  :: s
               , d_F  :: Spec s i o
               , out  :: s i → [[o]]
               , pred :: (SeenTrans s i o) → [[String]]}
:: SeenTrans s i o :== (s,i,[o],s)
```

Field s_0 is s_0, and d_F is δ_F. The function out is needed to generate the output sequences to be used when esmViz encounters a transition of type [o] → [s]. If such transitions can not occur in the used specification, this field can be undefined. Field pred is a predicate over the transitions seen during simulation as discussed in Sect. III. Each problem detected is reported as a nonempty list of strings.

While exploring esm, the tool collects all visited states, transitions and issues. This results in a partially known automaton, and is captured concisely with the following type:

```
:: KnownAutomaton s i o
   = { trans  :: [ SeenTrans s i o]
     , issues :: [(SeenTrans s i o,[String])] }
```

Encountered states can be extracted easily from the seen transitions and are not recorded separately. Transitions that correspond to an issue are drawn in red.

The tool esmViz also indicates the transitions that are part of the current traces. For a nondeterministic specification there can be multiple traces active. We record this as a list of transitions that is possible in each step of the trace.

```
::  Trace s i o :== [[SeenTrans s i o]]
```

Trace transitions are drawn in blue with larger arrowheads.

For implementation reasons it is convenient to record the set of active states. For a nonempty trace these are exactly the states in the after set of the current traces. Let this set after k steps be $\mathbb{S}_k$. Each state in $\mathbb{S}_k$ is rendered with a red interior. States are displayed as circles, where esm.s_0 has a double border. Initially, $\mathbb{S}_0 = \{$esm.s_0$\}$.

The user can choose one input of init($\mathbb{S}_k$), which is the set of all possible inputs. This set of inputs is empty for a final state. The number of possible inputs is limited (by default 50). Given a concrete choice **i** $\in$ init($\mathbb{S}_k$), esmViz adds all transitions from the current states that correspond to this input. For transition specified by Pt o t in spec, the output and target state to be used are immediately clear. For transitions specified by a function Ft f of type [o]↦[s], the function esm.out is used to determine the outputs and target states of transitions. If the target states of these transitions exist the arrows go to the existing states, otherwise the states are added to the ESD. If the transitions are already in the ESD, they just have to be painted red, otherwise they are added. The new transitions are also added to the traces, and the existing part of the trace is pruned to reflect the new extensions. The set of new states $\mathbb{S}_{k+1}$ is computed with $\mathbb{S}_{k+1} = \{t | s \in \mathbb{S}_k \wedge s \xrightarrow{\mathbf{i}/o} t \in \delta_F\}$.

The system determines for each known state whether the user has 'discovered' all outgoing edges, i.e. all edges with i in the init of that state. In that case, the state is rendered with a blue interior instead of a light grey default one. This provides a strong clue which part of esm has been fully explored.

Pressing the button labeled *Back* removes the last transition from each trace. The known automaton is not affected by going back in the trace. The browser's back button acts as undo action. With the *Add all* button all transitions leaving from the current states are added. These transitions are not added to the trace, nor effect $\mathbb{S}_k$. Using an integer edit field, adding transitions can be done recursively n steps deep. Pressing *Prune* removes all transitions and associated issues that do not belong to the current trace. The *Reset* button brings the esmViz tool in its initial state, only the state s_0 is displayed. The trace can be removed by the button *Clear trace*, the states and transitions in the ESD are not effected by this action.

The current state can be changed by clicking on a state in the diagram. If this state is part of the trace or reachable from an active state in one step the trace will be adapted accordingly, otherwise a new trace starts at that node.

B. Example

Here is the beverage vending machine esm specification:

```
vendingESM :: ESM State Input Output
vendingESM
= { s_0 = Off, d_F = vSpec, out = undef, pred = healthy }
```

where healthy checks **p1** – **p3** (Sect. III). An ESD showing all issues discribed by healthy is depicted in Fig. 2(**b**).

```
healthy :: (SeenTrans State Input Output) → [[String]]
healthy (s,i,o,t)
= [ if (vs+vi ≠ vo+vt) // value preservation in transition? (p1)
    ["value is not preserved in this transition, "
    ,"value s+value i=", toString (vs+vi)
    ,", and value o+value t=",toString (vo+vt)]  []
  , if (vt>Max)       // value of target state within bound? (p2)
    ["Value of target state ",toString vt
    ," larger than Max (",toString Max,")."]     []
  ,case (i,o) of   // obtained the ordered product? (p3)
    (Butt p,[Cup q]) | p =!= q
      = ["The required product is unequal"
        ," to the delivered product!"]
    _ = []
  ]
where vs = value s; vi = value i
      vo = value 0; vt = value t
```

V. Implementation

The esmViz tool has been written in Clean, using the iTask toolkit [8]. Despite its conciseness (800loc) it offers a fair amount of functionality (see also other tools in Sect. VI). In this section we present the most interesting parts of the implementation. These are the main structure of the GUI (Sect. V-A) and the integration of the ESD rendering tool Graphviz [3] that we used in the application (Sect. V-B).

A. The Main GUI Structure: Iterating iTasks

The main GUI structure of esmViz is an iteration of the main tool task function DiGraphFlow. As discussed in Sect. IV, it provides the user with a number of elements, expressed as a list of choices (the arguments of orTaskL below which folds the basic iTask -||- choice operator over the list):

```
DiGraphFlow (ka,as,trace,n)                                      1.
= orTaskL                                                        2.
  [issuesToHtml ka.issues !>> state                              3.
  ,chooseTaskV (sortBy (λ(a,_) (b,_).a<b)                        4.
   [(render i,step i) \\ i ← possibleInputs esm as])             5.
  ,chooseTask                                                    6.
   [("Back" ,        back)                                       7.
   ,("Prune",        prune)                                      8.
   ,("Reset",        return_V (newKA,[esm.s_0],[],n))            9.
   ,("Clear trace", return_V (ka,as,[],n)) ]                    10.
  ,stepN <<! traceHtml trace <<! legend ]                       11.
```

Note the correspondence between this definition and the GUI as displayed in Fig. 2(**a**). The list of found issues are displayed before the ESD editor (line 3); the possible inputs init $\mathbb{S}_k$ are defined in lines 4–5; the navigation commands are summarized in lines 6–11; and finally, the trace and legend are displayed in line 11. The state task is given below:

```
state                                                            1.
= editTask "OK"                                                  2.
  (mkDigraph ThisExe                                             3.
     ( ka, esm.s_0, as, allEdgesFound esm ka,                    4.
       map fst ka.issues, flatten trace ))                       5.
  =>> λdig → let                                                 6.
     (as‘,trace‘) = findSelectedStates dig ka as trace           7.
  in return_V (ka,as‘,trace‘,n)                                  8.
```

The iTask editTask l v combinator creates an editor with initial value v with which users can create new values of the same type as v's type. When the button labeled with l has been pressed, then the new value is returned by this editor and the task is done. As discussed in Sect. IV, the user can select a new state. For reasons of space, we do not show the code of the other functions.

B. The Rendering of the Explored Automaton

By far the most intricate component of the GUI is the ESD editor. Creating attractive renderings of directed graphs is known to be a hard problem. Fortunately, we can rely on other tools to solve this problem. Here we have used the Graphviz tool set [3]. Directed graphs are described using the DOT language. Given a DOT text file, the dot tool can be invoked to create a rendering in various formats (we will use the *gif* output). Note that this interface is text-based, whereas editors in the iTask toolkit are type based. We can embed the text based tools of Graphviz in the type based iTask toolkit in a compositional way by defining a suitable collection of data types that describe an ESD as a directed graph. This collection of data types captures the DOT language. The relevant top level type definitions are:

```
:: Digraph      = Digraph String [GraphAttribute]
                                 [NodeDef]
                                 (Maybe SelectedItem)
:: NodeDef      = NodeDef Int [NodeAttribute] [EdgeDef]
:: EdgeDef     :== (Int,[EdgeAttribute])
:: SelectedItem = Node Int
```

A (Digraph name atts nodes item) value represents a directed graph. A directed graph has nodes, each of which is identified by a number, and is connected with other nodes by means of edges. Graphs, nodes, and edges have attributes. Graphviz supports an extensive set of attributes (almost 150) that can be used to alter and tweak the output. In DOT, attributes are specified as *name* = *value* pairs. Some attributes are shared by graphs, nodes, and edges. We have represented attributes separately for graphs, nodes, and edges, each as a list of unary data constructors. For instance, for graph attributes we have GAtt_*name value* pairs. A single generic function prints these values as correct DOT expressions. The result is that we have both a typed representation of DOT expressions (Digraph values) as well as a textual representation (printing such a value with toString). The function mkDigraph yields the Digraph value that represents the currently explored ESD.

The iTask editor for Digraph values performs the following actions for a d :: DiGraph value identified by *name*. First, compute e = toString d and save e in file *name.dot*. Second, invoke *dot* on *name.dot*, which yields a rendering as *name.gif*. Third, invoke *dot* to create a *name.map* file to allow the user to select states. Fourth, alter the lines in *name.map* to invoke a script that sends the label of the selected state to the server application. Finally, generate the proper HTML to be included in the application page. The server application, when receiving the label of a selected state, updates the corresponding Digraph value to reflect the change. Now the application continues with the new Digraph value.

VI. Related Work

The mCRL2 tool set [**?**], [**?**] uses a process algebraic specification language, mCRL2 [**?**], to describe distributed, communicating systems. It has a functional style data language with recursive types, data constructors, functions, lambda-abstraction, and structured data. It comes with an extensive number of tools (15) for analysis purposes. Five are relevant to our work: with xsim a user can explore a linearized mCRL2 specification in a similar way as with our tool, using a GUI (the simulation tool sim has a command line interface): the user can select actions, after which the tool shows the resulting state. Besides interactively exploring the mCRL2 specification, the tool set also allows to render the complete state space: NoodleView (for 2D rendering) and FSMView (for 3D rendering). Before this is possible, the state space needs to be generated with lps2lts.

The TorX tool set [9], [?] is a model based test tool to check conformance of real suts, based on the ioco theory of testing. The specification is a Labeled Transition System (LTS), or one that is derived from a higher level specification language that converts to LTS (e.g. mCRL2 described above). The tool uses the specification to automatically determine inputs, observe outputs from the sut, and make a final verdict. In this sense, it is not useful for exploring a specification. However, once a test run has been created, the user can explore the actual trace which is depicted as a message sequence chart.

The Uppaal tool set [1], [7] can be used for both validation and verification (using model checking) of time-based systems. Validation is done by means of a graphical simulator of a time-based automaton specification. The automaton specification is basically a labeled transition system with timing constraints. Uppaal allows for simple data types, clocks, and constraints on these clocks. The user can create specifications in an intuitive, graphical way. The user can stepwise direct the system's behavior, or generate a random trace.

The esmViz tool differs with the mCRL2 approach in that we use a single modeling formalism. Except for the 3D rendering all of the functionality of the mCRL2 tool set is available in esmViz. The TorX tool set is really a model based testing harness, and is less suited for exploration purposes. Specifications within Uppaal can be created graphically. In esmViz specifications are given as a function, out of which a graphical approximation is 'discovered' by the user or by the system. In our opinion this combines the best of both worlds: the succinctness of functional programming with the intuitive appeal of a graphical rendering.

VII. Experiences

In order to judge the quality of esmViz 10 master students in computer science studied some test cases with and without esmViz. These students are literate Clean programmers, have a basic understanding of model-based testing with G∀st and the specifications needed (but no hands-on experience). After an introduction to esmViz and playing with an example similar to the beverage vending machine in this paper the students were asked to locate problems in two other case studies. The examples were heavily parameterized specifications of a number guessing game and a telephone number database that contains potentially over one million states. Drawing all these states makes finding the problems only harder. The errors in the specification can however all be found by traces of about ten to twenty transitions.

The students found esmViz very handy to get a feeling for the behavior of the specified system. Everybody found it much easier to understand a specified system with the tool than without. Finding errors in the specification by simulation remains hard, but the tool makes it easier. The same holds for finding the source of issues found by G∀st. This is consistent with the general observation in all kinds of testing: finding issues is one thing, but finding their cause is another.

VIII. Conclusions

There are two kinds of conclusions from the work described in this paper. First, the specification simulator esmViz described in this paper really helps a lot to understand the behavior of the extended state machines used as specification in model-based testing. Although the compiler of the statically typed functional programming language used as carrier of these specifications checks the models, the models can still contain errors. Finding these semantical errors is hard. The simulator helps in locating these problems, especially if an appropriate constraint on transitions or states is known. Second, implementing such a tool with iTasks is a real pleasure. Integrating Graphviz with iTasks turned out to be smooth. Implementing a browser interface for esmViz using the iTask system imposes some restrictions on the layout of the GUI, but works well. The different possible user actions are modeled each by an iTask. The iTask system is well suited to compose these tasks in a flexible way and takes care of rendering them.

References

[1] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[2] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 268–279. ACM Press, 2000.

[3] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[4] P. Koopman. Testing with functions as specifications. In E. Brinksma, W. Grieskamp, and J. Tretmans, editors, *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[5] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer, 2003.

[6] P. Koopman and R. Plasmeijer. *Fully Automatic Testing with Functions as Specifications*, volume 4164 of *LNCS*, pages 35–61. Springer, Eotvos Lorand University, Budapest, Hungary, July 4-16 2006.

[7] K. Larsen, M. Mikucionis, and B. Nielsen. Online Testing of Real-Time Systems Using UPPAAL. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004 - Revised Selected Papers*, volume 3395 of *LNCS*, pages 79–94. Springer, September 21 2004.

[8] R. Plasmeijer, P. Achten, and P. Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 141–152, Freiburg, Germany, Oct 1–3 2007. ACM.

[9] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.