The following full text is a preprint version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/66905

Please be advised that this information was generated on 2017-09-24 and may be subject to change.
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 easy use. 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 GVst 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 ([7]).

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 GVst 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, 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 deterministic state transition system used as model in the tests. The model is a pair of states, inputs and outputs can be infinite. The sut is an extended state system, ESM. An ESM consists of some initial state

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 GVst [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.

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 GVst [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.

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 GVst [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.

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 GVst [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.

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 GVst [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.

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 GVst [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.

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 GVst [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.

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 GVst [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.

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 GVst [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.
so and a set of transitions of the form \( s \overset{i/o}{\rightarrow} t \). 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 \( s \overset{a}{\Rightarrow} t \) is a sequence of labels. The empty trace contains no labels. If we have a trace \( s \overset{a}{\Rightarrow} t \) and a transition \( i/o \) \( u \) we can construct the trace \( s \overset{\sigma}{\Rightarrow} u \). If we are not interested in the target state, we will occasionally write \( s \overset{a}{\Rightarrow} t \) and \( s \overset{\sigma}{\Rightarrow} \). All traces from a given state are defined as: \( \text{traces}(s) \equiv \{s \overset{a}{\Rightarrow} t \} \). 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 \( s \overset{i/o}{\rightarrow} t \). The after of a state \( s \) is the set of possible target states \( t \), reachable after the given trace \( \sigma: s \overset{\sigma}{\Rightarrow} \).

### 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:

\[
\text{sut conf spec } \equiv \forall \sigma \in \text{traces}_{\text{spec}}(s_0), \\
\forall i \in \text{init}(s_0) \text{ after spec } \sigma, \\
\forall o \in O, \\
(t_0 \text{ after sut } \sigma) \overset{i/o}{\rightarrow} (s_0 \text{ after spec } \sigma) \overset{i/o}{\rightarrow}
\]

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 \( \sigma \), 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 \( \text{spec} \).

### 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 \( \text{Vgst} \) 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, \( s_0 \) after \( \sigma \), rather than the trace \( \sigma \). 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 testing is sufficiently large).

### C. Representation of the transitions

To represent the ESM in the model-based test tool \( \text{Vgst} \) 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 \( S \times I \rightarrow [\text{Trans} O S] \).

A basic assumption in \( \text{Vgst} \) 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 \text{Clean} these types are:

\[
:: \text{Spec } s i o := s i \rightarrow [\text{Trans } s i o] \\
:: \text{Trans } s i o = \text{Pt } s i o \mid \text{Pt } (s o i) \rightarrow [s]
\]

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 it enters state On v (0 ≤ v < Max) 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:

\[
:: \text{Money} := \text{Int} \\
:: \text{State} = \text{Off} | \text{On Money} \\
:: \text{Input} = \text{SwitchOn} | \text{SwitchOff} \\
:: \text{Coin Money} | \text{Butt Product} \\
:: \text{Product} = \text{Coffee} | \text{Tea} \\
:: \text{Output} = \text{Cap Product} | \text{Return Money}
\]

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

\[
\text{vSpec} :: !\text{State} !\text{Input} \rightarrow !\text{Trans Output State} \\
\text{vSpec Off SwitchOn} = \text{Pt } [[\text{On } 0]] \\
\text{vSpec s SwitchOff} = \text{Pt } [[\text{Off}]] \\
\text{vSpec (On s) (Coin c)} \\
// condition should be s < c \leq \text{Max} \\
\text{c} \leq \text{Max} = \text{Pt } [[\text{On } (s + c)]] \\
// output should be Return c
\]
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\*φ. 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*φ, 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) = {ButtCoffee, ButtTea, Coin10, Coin20, SwitchOff}. 4. Commands for navigation purposes, resetting the exploration, and so on. 5. The current trace, as explained in Sect. II. Here the trace
A. The ESM description

Each problem detected is reported as a nonempty list of strings.

Encountered states can be extracted easily from the seen transitions 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 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.

\[ \text{KnownAutomaton } s i o := \{ \text{trans } : : \text{SeenTrans } s i o, \text{issues } : : \{ \text{SeenTrans } s i o, \text{String} \} \} \]

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.

\[ \text{Trace } s i o := \{ \text{SeenTrans } s i o \} \]

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 $S_k$. Each state in $S_k$ is rendered with a red interior. States are displayed as circles, where esm.s_0 has a double border. Initially, $S_0 = \{\text{esm.s_0}\}$. The user can choose one input of init($S_0$), 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 $i \in \text{init}(S_0)$, esmViz adds all transitions from the current states that correspond to this input. For transition specified by $\text{pt} o t$ in spec, the output and target state to be used are immediately clear. For transitions specified by a function $\text{pt} f t o t$ of type $[o]b{s}$, 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 $S_{k+1}$ is computed with $S_{k+1} = \{s | s \in S_k \land s \xrightarrow{i/o} t \land t \in S\}$.

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 $S_k$. 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:

\[
\text{vendingESM } ::= \text{ESM State Input Output}
\]

\[
\text{vendingESM} = \{ s_0 = \text{Off}, d_F = \text{vSpec}, \text{out} = \text{undef}, \text{pred} = \text{healthy} \}
\]

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.
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 follows the basic iTask - | choice operator over the list):

\[
\text{DiGraphFlow} (ka, as, trace.n) = \text{orTaskL} \\
\begin{cases}
\text{issuesToHtml} ka, issues !>> \text{state} \\
\text{.chooseTaskV} \left( \text{sortBy} (\lambda(a,_) (b,_) a < b) \right) \left( \text{render} 1, \text{step} 1 \right) | i \mapsto \text{possibleInputs} \text{ esm as} \\
\text{.chooseTask} \\
\{ "\text{Back}" | \text{back} \\
\{ "\text{Prune}" | \text{prune} \\
\{ "\text{Reset}" | \text{returnV} (newKA, [esm.s_0], [])) \\
\{ "\text{Clear trace}" | \text{returnV} (ka, as, []) \\
\{ "\text{Linearize trace}" !>>> \text{traceHtml} \text{trace} !>>! \text{legend} \\
\end{cases}
\]

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 list \( l_v \) 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:

\[
\text{state} = \text{editTask} "OK" \text{ mkDiGraph ThisExe} \\
\quad (ka, esm.s_0, as, allEdgesFound esm ka, \\
\quad \map \text{fstat ka.issues, \text{flatten trace}}) \\
\quad \rightarrow \text{Dig} \rightarrow \text{let} \\
\quad (\text{as}, \text{trace}) = \text{findSelectedStates} \text{ dig ka as trace} \\
\quad \text{in returnV} (ka, as, trace'.n)
\]

The \text{iTask} \text{editTask} \( l_v \) \( v \) \text{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.

### V. IMPLEMENTATION

The esmViz tool has been written in Clean, using the iTask toolkit [8]. Despite its conciseness (800 loc) 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 follows the basic iTask - | choice operator over the list):

\[
\text{DiGraphFlow} (ka, as, trace.n) = \text{orTaskL} \\
\begin{cases}
\text{issuesToHtml} ka, issues !>> \text{state} \\
\text{.chooseTaskV} \left( \text{sortBy} (\lambda(a,_) (b,_) a < b) \right) \left( \text{render} 1, \text{step} 1 \right) | i \mapsto \text{possibleInputs} \text{ esm as} \\
\text{.chooseTask} \\
\{ "\text{Back}" | \text{back} \\
\{ "\text{Prune}" | \text{prune} \\
\{ "\text{Reset}" | \text{returnV} (newKA, [esm.s_0], [])) \\
\{ "\text{Clear trace}" | \text{returnV} (ka, as, []) \\
\{ "\text{Linearize trace}" !>>> \text{traceHtml} \text{trace} !>>! \text{legend} \\
\end{cases}
\]

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 list \( l_v \) 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:

\[
\text{state} = \text{editTask} "OK" \text{ mkDiGraph ThisExe} \\
\quad (ka, esm.s_0, as, allEdgesFound esm ka, \\
\quad \map \text{fstat ka.issues, \text{flatten trace}}) \\
\quad \rightarrow \text{Dig} \rightarrow \text{let} \\
\quad (\text{as}, \text{trace}) = \text{findSelectedStates} \text{ dig ka as trace} \\
\quad \text{in returnV} (ka, as, trace'.n)
\]

The \text{iTask} \text{editTask} \( l_v \) \( v \) \text{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
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], [?] 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 GVst 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 GVst. 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


