MBT'04 Preliminary Version

Heuristics for Faster Error Detection with Automated Black Box Testing

Antti Kervinen[1] and Pablo Virolainen[2]

Tampere University of Technology
Institute of Software Systems
PO Box 553, FIN-33101 Tampere, FINLAND

Abstract

Three building blocks for test guidance algorithms, the *step evaluation*, the *state evaluation* and the *evaluation order*, are proposed in this paper. We show how a simple family of coverage criteria can be used to evaluate individual testing steps, and how the nondeterministic behaviour of the tested system can be handled and longer term test step plans created with the state evaluation. We use the evaluation order to define which and when states are evaluated. Six heuristic algorithms based on these ideas are implemented. Four of them use a game-like approach to black box testing. In addition, three other test guidance algorithms are implemented for comparison. The algorithms are compared by measuring the number of testing steps required for detecting errors that are infiltrated to the conference protocol systems of two different sizes.

Key words: Conformance testing,testing automation,test selection heuristics

1 Introduction

To find errors by testing under given constraints on time and cost, the key issue is: "What subset of all possible test cases has the highest probability of detecting the most errors" [12].

Fault detection is more likely when the system under test (SUT) is guided to states that have not been visited before. To help guide the system into these unexplored areas various coverage metrics, such as statement, decision, and condition coverage, can be used in white box testing [12].

In black box testing we do not have information on the internal structure of the SUT. In this paper we show that several simple coverage metrics can

[1] ask@cs.tut.fi

[2] pablo@cs.tut.fi

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

still be helpful to reveal errors. Since the coverage of the states of the system cannot be measured, the coverage of the specified behaviour (that is, the interaction with the environment) of the system is measured.

In this paper we will use a *labelled transition system* (LTS) to model a SUT's behaviour. Although our model is a single "flat" LTS, our algorithms and results apply to several other formalisms also. For example, parallel composition of several LTSs, LOTOS, SDL, Petri nets or any other formalism that can be unwinded into a flat state space can be considered. Yet this leads easily to the state explosion problem, there are methods that can be used to cope with it. The state space can be constructed compositionally [17], or if it can not be constructed at once, it can be generated on-the-fly [7] by calculating the next states to a bounded depth [14]. Not all the algorithms of this paper are applicable with the last method, but the best performers are.

Exploration testing [9], as we call our testing method, is automated. When the expected behaviour of the system (in the form of a deterministic LTS) is given to a test engine, it explores the LTS beginning from its initial state. In each state the engine decides whether to send an input, listen to an output, or reset the system and restart the testing. Only those inputs and outputs that are possible in the current state of the LTS may be sent to or accepted from the SUT.

The problem is how to make decisions which reveal errors, that is, make the system give an unexpected output or remain silent when it should give an output, without exploring the LTS for too long.

The main objective and contribution of this paper is to introduce and compare heuristic algorithms which aim revealing errors quickly; and show how the problem of designing a specification coverage aided test selection algorithm can be split in parts. The introduced algorithms are compared by running tests against conference protocol systems consisting of two and three conference protocol entities. (The conference protocol basically provides a chat service with multiple chat rooms.) The number of test steps needed for finding an error is measured and compared. Errors are infiltrated to the systems by replacing one correct protocol peer by a mutated one.

Correct and mutated protocol entities were implemented at the University of Twente [4]. The implementations have been tested before in [1,14], but with a different test setup. While in the mentioned publications a single conference protocol entity is tested, we test the service provided by systems of two and three clients. With this setup we aim to capture the nature of a truly concurrent and reactive system: the system is often able to read inputs and give several correct responses at the same time.

Numerous papers have been published on automated test generation. In [2] Chow presented an automata theoretic testing strategy. A generalised version of his W-method [3], was further developed by Luo and v. Bochmann [11] to suit better for testing concurrent systems. These methods, among the methods based on distinguishing sequences and unique input/output sequences [15], aim to find "transfer faults". They check, transition by transition, that after the transition the SUT and the specification enter identical states. Often these methods provide very strong (or complete) fault coverage, but also the price is high. Required tests are extremely long and, to ensure the fault coverage, assumptions on the specification and the size of the state space of the SUT have to be made.

The transition tour method [13] tries to execute every transition in the specification at least once. This approach is more practical (for example, incomplete specifications are allowed) than the methods mentioned earlier, although it tests less. Our approach resembles this method, but in addition to covering transitions we can use simultaneously coarser coverage criteria to guide test runs. In [10] Lee et.al. introduced transition tours for communicating finite state machines. They presented a guidance method that tries to execute every transition in each state machine separately, instead of the full state space of the specification. This is an interesting and justified coverage criteria which, unfortunately, can not be handled by the coverage classes presented in this paper.

Our ideas on test guidance are somewhat similar to Pyhälä and Heljanko [14]. They proposed a heuristic which may precalculate some future steps in the specification to make the choice of the next step. We have implemented also their algorithm for comparison.

In Section 2 the LTSs and coverage criteria are formalised. The test engine architecture is presented in Section 3 to show the context in which the test guidance algorithms operate. Section 4 contains the main issue of the paper. Firstly, the concepts of step evaluation, state evaluation and evaluation order, which are the basic building blocks of our heuristic algorithms, are presented. The rest of the section is dedicated to the description of the heuristic algorithms to be compared. The test setup for the comparison and the results are presented in Section 5.

2 Background

We will use labelled transition systems (LTS) to represent the behaviour of the system under test (SUT).

Definition 2.1 [LTS] A labelled transition system, abbreviated LTS, is a quadruple $(S, \Sigma, \Delta, \hat{s})$ where S is a set of *states*, Σ is a set of *visible actions* (alphabet), $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ is a set of *transitions* where $\tau \notin \Sigma$ is an *invisible action*, and $\hat{s} \in S$ is an *initial state*.

Every visible action is either input or output (response). A transition is called an *input* (*output*) *transition* if it is labelled by an input (output) action.

Definition 2.2 Let $L = (S, \Sigma, \Delta, \hat{s})$, be an LTS $s \in S$, $\Sigma_I \subseteq \Sigma$ the set of input actions and $\Sigma_O = \Sigma \setminus \Sigma_I$ the set of output actions. $T_I(s) = \{\, (s, a, s') \in \Delta \mid a \in \Sigma_I \,\}$ is the set of input transitions leaving state s. Correspondingly

$T_O(s) = \{ (s, a, s') \in \Delta \mid a \in \Sigma_O \}$ is the set of output transitions leaving state s.

The test engine, which is the entity testing the SUT, is given a single, deterministic (that is, there are no transitions labelled by invisible actions and two transitions leaving the same state never share the same label) LTS as an input for a test run. At each point of the test run the engine keeps track of the *current state* of the LTS, starting from the initial state. The transitions leaving the current state specify what actions are allowed in the next testing step.

Execution of a transition means that the current state is updated to the destination state of the transition. Because only the transitions leaving the current state can be executed, and because the LTS is deterministic, we may as well talk about executing an action without confusion. The determinism is required because we want to measure the coverage of the transitions.

The transitions of an LTS $(S, \Sigma, \Delta, \hat{s})$ that are executed during a test run are stored in a multiset $\Delta_{\text{exec}} \in \mathbb{N}^{\Delta}$. As well, Δ_{exec} can be thought as a map from transitions to natural numbers. If a transition $t \in \Delta$ has not been executed, then $\Delta_{\text{exec}}(t) = 0$ and we denote $t \notin \Delta_{\text{exec}}$. If t has been executed $n > 0$ times, $\Delta_{\text{exec}}(t) = n$ and we also say that $t \in \Delta_{\text{exec}}$. Adding elements $T \subseteq \Delta$ to the multiset is defined naturally: $\Delta'_{\text{exec}} = \Delta_{\text{exec}} + T$, where for every t in T it holds $\Delta'_{\text{exec}}(t) = \Delta_{\text{exec}}(t) + 1$ and for the other $t' \in \Delta, t' \notin T$ it holds $\Delta'_{\text{exec}}(t') = \Delta_{\text{exec}}(t')$.

Our test engine is able to measure eight coverage classes which are identified by triplets of zeroes and ones.

Definition 2.3 Transition $t = (s, a, s')$ is $b_1b_2b_3$-*covered* by transition $t_e = (s_e, a_e, s'_e)$ iff $((b_1 = 1) \Rightarrow (s = s_e)) \wedge ((b_2 = 1) \Rightarrow (a = a_e)) \wedge ((b_3 = 1) \Rightarrow (s' = s'_e))$.

Definition 2.4 Let $L = (S, \Sigma, \Delta, \hat{s})$ and Δ_{exec} be a multiset of executed transitions. $\Delta_{b_1b_2b_3}(\Delta_{\text{exec}}) = \{ t \in \Delta \mid \neg\exists t' \in \Delta_{\text{exec}} : t'\ b_1b_2b_3\text{-covers } t \}$ is the set of *uncovered transitions* in the sense of coverage class $b_1b_2b_3$.

In this paper we use three coverage classes: 111, 001 and 010. Intuitively, to cover an LTS according to the coverage 111 every transition of the LTS must be executed. The coverage 001 demands that every state be arrived in, and 010 that every action appearing in the transitions of the LTS appears also in the executed transitions.

3 Test engine

The explorer is the heart of the test engine (see Figure 1). It runs in a loop, asking the heuristics module for the next testing step. If the heuristics suggests sending an input to the SUT, the explorer forwards it to the adapter module. The adapter converts input actions to commands and sends them to

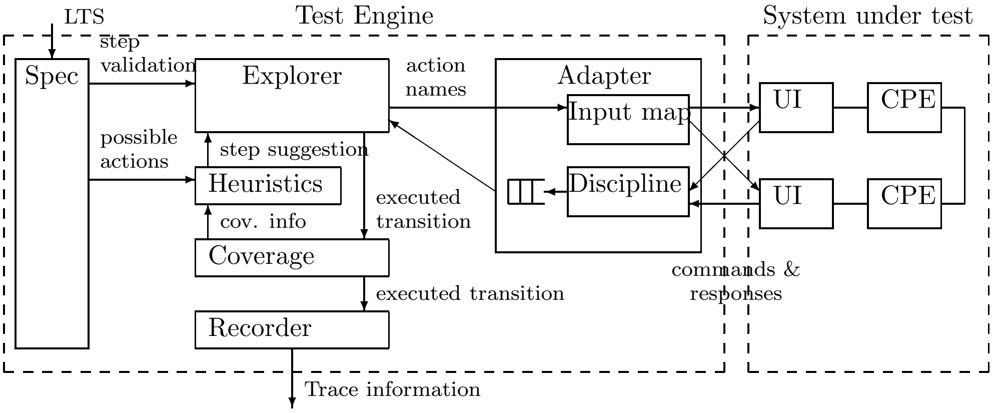

Fig. 1. Testing architecture

the appropriate receiver in the SUT. In our case the receivers are the user interface processes (UI) of the actual implementations of the conference protocol entities (CPE).

If the input action is accepted, which in our setup means that sending the command to the `stdin` pipe of the UI process was successful, the explorer executes the corresponding transition in the specification. Executions change the current state of the specification, coverage values, and get recorded in the executed trace. If the input action is refused, the explorer executes the refused input action in the specification, if possible. The ability to detect the refusal of inputs makes the tested relation between specification and the implementation more extensive than the *ioco* [16] relation. However, we actually are testing the ioco relation in this paper because in our test runs the input refusals neither occur nor are taken into account in the specification.

If the heuristics suggests listening to the output, the explorer tries to read and remove the first element in the response queue of the adapter module. The queue is filled by the discipline module which reads responses from the `stdout` pipes of the UIs and converts them into actions. The discipline module decides the order in which the responses of the processes are queued, in case the responses become readable simultaneously. We use "random discipline": when more than one response is readable, they can be queued in any order.

If the queue is empty, the explorer waits until an action becomes readable from the queue or a timeout occurs. The timeout causes the explorer to receive a special δ (quiescence) action (δ belongs to the alphabet of the specification LTS and is considered an output action).

When an output action is received from the adapter module, the explorer verifies from the specification module that the received action can be executed in the current state. If not, it announces an illegal response error and stops testing. Otherwise, the action is executed similarly to an input transition execution.

If the heuristics module decides that testing should not be continued from the current state, it returns "deadlock" to the test engine. Our current implementation of the test engine stops testing after the deadlock.

4 Heuristics

We divide the problem of building a test guidance algorithm in three parts. The first part, *step evaluation*, gives values to single input and output transitions. The value describes the desirability of their execution. In addition to the transition data, the multiset of executed transitions may affect the values.

Definition 4.1 [Step evaluation function] For an LTS $L = (S, \Sigma, \Delta, \hat{s})$ a step evaluation function is a function $eval : \Delta \times \mathbb{N}^{\Delta} \to \mathbb{R}$.

Test guidance would be more straightforward if there were at most one possible output transition in every state. Then we could, given an appropriate step evaluation function, precalculate the shortest possible sequence of transitions after which the step evaluation function is non-positive for every transition. Of course, this would not be very practical because finding such a sequence is easily an NP-hard problem.

However, the guidance becomes even more complicated when there are several alternative correct outputs at the same time. With *state evaluation*, which is the second part of the problem, we try to estimate the best step sequences by making assumptions on the behaviour of the SUT when it is allowed to respond in several different ways.

State evaluation associates a value with a state, based on the values of the transitions leaving the state (given by step evaluation), the values of their destination states (by previous state evaluations), and according to its expectations on next outputs.

We have implemented two state evaluation heuristics. In the first one, used in the "pessimistic player" algorithms, the SUT is expected to output actions so that it minimises the desirability of the possible futures.

In the other state evaluation method, the probabilities of responses in each state are estimated during the test run, based on the responses that are actually received in the state from the SUT. The average of the desirabilities of the responses weighted with their probabilities is used for the evaluation of the state. This is applied in the "adaptive player" and "state space evaluation" algorithms.

The third and the last part of the process is the *evaluation order*: which states and transitions should be evaluated and when. The algorithms implement two evaluation orders. In the player algorithms a limited number of steps beginning from the current state is calculated. The names of the algorithms are inspired by the game-like approach to a test run. In every turn the test engine decides if it makes a move (sends an input to the SUT) or lets the SUT make a move (listens to the response of the SUT). The move is the execution of a transition: each move increases the score by the value of the transition (given by the step evaluation function). The player algorithms are greedy:

they try to gather as big a score as possible during the steps they calculate.

In the evaluation order of the "state space evaluation" algorithm values for every state of the LTS are calculated at the beginning of a test run. The values are recalculated when necessary during the test. Thereby, all parts of the state space, even far away from the current state, affect the decision of the next testing step.

We have implemented the following nine test guidance algorithms and evaluated their error detection performance in test runs. The first two are simple random algorithms which are implemented just for sparring with the other seven.

4.1 Random

The random heuristic selects randomly one transition leaving the current state. If the selected transition is labelled by an output action, the test engine waits for an output. Otherwise the input action of the transition is sent to the SUT.

4.2 Greedy random

This is a slightly enhanced random heuristic. In each state a decision about the next step is made according to the following rules. Output is listened to if there are no inputs or if a correct output certainly causes a new transition to be covered. Otherwise, if a new transition is covered by executing an input action, then the input is sent. If not, then if it is possible that the SUT gives output that covers a new transition, then output is listened to in a state s with the probability $|\{ t \in T_O(s) \mid t \notin \Delta_{\text{exec}} \}| \,/\, |T_O(s)|$. If the output were not listened to due to the previous rule, and if both inputs and outputs are possible in the state, then output is listened to with probability 0.5. Otherwise, a random input is chosen.

4.3 Pessimistic player

There are two versions of the pessimistic player algorithm that have different step evaluation functions. Both use the same prediction heuristic in the state evaluation: the SUT is assumed to choose its responses so that it minimises the sum of the evaluation function values of executed transitions.

The skeleton of the state evaluation parts of the player algorithms is presented as Algorithm 1. It returns a value that describes how desirable is the state that is given as the first parameter.

The algorithm is recursive. As the bottom of the recursion, it returns the value 0 to the states for which it is called with the (search) depth 0. If *depth* > 0, the best input value and the value for listening to outputs are calculated. The desirability of an input transition $t \in T_I(s)$ is the sum of $\mathit{eval}(t, \Delta_{\text{exec}})$ (a step evaluation function, we will get back to this soon) and the value of its destination state. The latter is calculated by calling recursively

```
function $player(s : \mathsf{state}, depth : \mathsf{integer}, \Delta_{\text{exec}} : \mathsf{multiset\ of\ transitions})$
    if $depth = 0$ or $(T_O(s) = \emptyset$ and $T_I(s) = \emptyset)$ then return 0 end if

    $desirability$ and $P$ are local associative maps, initially empty.
    $action$ is a global associative map.
    $best\_input$ is a structure consisting of a transition and a value

    ## Evaluate inputs, store a transition–value pair with the greatest value
    $best\_input := \underset{t \in T_I(s)}{\text{MAX}} [eval(t, \Delta_{\text{exec}}) + player(dest\_state(t), depth-1, \Delta_{\text{exec}} + \{t\})]$

    ## Evaluate outputs, calculate expected output value
    for each $t \in T_O(s)$ do
        $desirability[t] := eval(t, \Delta_{\text{exec}}) + player(dest\_state(t), depth-1, \Delta_{\text{exec}} + \{t\})$
    end for
    $P := estimate\_probabilities\_of\_outputs(T_O(s), \Delta_{\text{exec}}, desirability)$
    $ovalue := \sum_{t \in T_O(S)} P[t] \cdot desirability[t]$

    ## Return the best choice, prefer output when input and output are equally good
    if $T_O(s) = \emptyset$ or $(T_I(s) \neq \emptyset$ and $best\_input.value > ovalue)$ then
        $action[s] := best\_input.transition$
        return $best\_input.value$
    else
        $action[s] := listen\_to\_output$
        return $ovalue$
    end if
```

Algorithm 1. The state evaluation skeleton of the player algorithms

the algorithm for the destination state with depth decreased by one and with t added to the multiset of the executed transitions. The input transition that has the greatest desirability value is stored with the value to the *best_input* structure. Our implementation of the MAX function is deterministic in the player algorithms: if there are many equally desirable input transitions, it always returns the same. The desirabilities of outputs $T_O(s)$ are calculated similarly to the inputs.

In the pessimistic player algorithm the probability estimation returns a map P where $P[t] = 1$ for the output transition t which has the minimal desirability. For the other output transitions t', $P[t'] = 0$. Thus the minimal desirability of the available output transitions is stored in *ovalue*.

Finally, the greater of the values of *best_input.value* and *ovalue* is associated with the state s by returning it as the value of the function. The algorithm stores the best action in the state (either *listen_to_output* or the best input) in the global *action* table.

The evaluation order in the player algorithms is the same. The *player* function is called before every testing step with the following parameters:

- $s =$ the *current state* of the specification
- $depth = 5$
- Δ_{exec} the multiset of the executed transitions during the test run so far.

When the algorithm stops, the action of the next testing step is stored in $action[current\_state]$.

In the simple version of the algorithm only the transition coverage (111) is taken into account in the step evaluation function, see equation (2) below. Thus all transitions that are not executed are considered equally desirable. To avoid trying to execute the same sequence of transitions unsuccessfully over and over again the execution count of a transition decreases the value of the step evaluation function. We use the decrement of 1/10 of the value of an unexecuted transition per execution.

$$(1) \qquad mem(t,T) = \begin{cases} 0 \text{ if } t \notin T \\ 1 \text{ if } t \in T \end{cases}$$

$$(2) \quad eval\text{-}simple(t, \Delta_{\text{exec}}) = 10 \cdot mem(t, \Delta_{111}(\Delta_{\text{exec}})) - \Delta_{\text{exec}}(t)$$

$$(3) \quad eval\text{-}comp(t, \Delta_{\text{exec}}) = 10 \cdot mem(t, \Delta_{111}(\Delta_{\text{exec}})) + 50 \cdot mem(t, \Delta_{001}(\Delta_{\text{exec}})) \\ +250 \cdot mem(t, \Delta_{010}(\Delta_{\text{exec}})) - \Delta_{\text{exec}}(t)$$

The heuristic idea behind the complex evaluation function (3) is that sending an input that has not been sent before as well as requesting an unseen output are the easiest ways to find errors. Therefore, new input and output *actions* should be tested with high priority compared to new input and output *transitions*. Likewise, visiting unvisited states is appreciated.

The factors 10, 50 and 250 in the complex evaluation function are fixed with the assumption that the algorithm is called with search depth 5. In this case the existence of an unexecuted action in one search branch makes it superior to branches which include only actions that have been executed. If the best branches contain equal number of unexecuted actions, then the branches containing unvisited states are superior to those with none.

4.4 Adaptive player

Similarly to the pessimistic player, there are two versions of the adaptive player algorithm. One uses the step evaluation function (2) and the other the function (3). The only difference between the adaptive and the pessimistic player is the way they estimate probabilities of outputs that are available at the same time.

Whereas the pessimistic player gives the output listening the minimal value of the branches beginning with output actions, the adaptive player uses a weighted average of the output branches. The idea is to try to estimate the probabilities of the output actions based on the responses that have been received previously in the same state.

```
function $sseval((S, \Sigma, \Delta, \hat{s}) : \mathsf{LTS}, \Delta_{\text{exec}} : \mathsf{multiset})$
    for each $s \in S$: $value[s] := 0$ end for
    $Calc := S$
    while $Calc \neq \emptyset$
        choose $s \in Calc$, $Calc := Calc \setminus \{s\}$
        $best\_input := \underset{t \in T_I(s)}{\text{MAX}} [dec(value[dest\_state(t)]) + eval(t, \Delta_{\text{exec}})]$
        $oval := \sum_{t \in T_O(s)} P[t] \cdot [dec(value[dest\_state(t)]) + eval(t, \Delta_{\text{exec}})]$
        $new\_value := max(best\_input.value, oval)$
        if $new\_value \neq value[s]$ then
          $Calc := Calc \cup previous\_states(s)$
          $value[s] := new\_value$
        end if
        if $best\_input.value > oval$ or $(best\_input.value = oval$ and $rand() < 0.5)$
        then
          $choose[s] :=$ choose randomly among the best input transitions
          $action[s] := best\_input.transition$
        else
          $action[s] := listen\_to\_output$
        end if
    end while
```

Algorithm 2. State space evaluation

The function *estimate_probabilities_of_outputs*$(T_O(s))$ in Algorithm 1 returns the following mapping from output transitions to probabilities:

$$P[t] = \frac{1 + \Delta_{\text{exec}}(t)}{\sum_{t' \in T_O(s)} (1 + \Delta_{\text{exec}}(t'))} \tag{4}$$

If none of the output transitions has been executed, their probabilities are equal. The probability of a transition grows when it becomes executed. Thus the player algorithm tries to adapt to the behaviour of the SUT.

4.5 State space evaluation

The state space evaluation algorithm associates desirability values with both states and transitions. Unlike players, it evaluates the values for every state in the state space. Thus the algorithm is able to make very long plans for execution sequences. The price to pay is that calculating the values may be expensive in big state spaces. The state space evaluation algorithm is presented as Algorithm 2.

The algorithm stores values evaluated for each state in the *value* table, which initially contains zeroes. In the beginning, every state belongs to the *Calc* set, which is the set of the states to be (re)evaluated. The algorithm runs in a loop until the *Calc* set is empty.

In the loop, one state at a time is removed from the *Calc* set and evaluated as follows; let the chosen state be s. The value for sending an input is stored into the *best_input* structure, with a transition that gives the value. The transition is chosen randomly among the equally good ones. The value is the greatest of the sums of $\mathit{eval}(t, \Delta_{\mathrm{exec}})$ and the decreased value of the destination state of t, where $t \in T_I(s)$.

In our test runs we chose the decrement to be roughly 10 % (with a small constant decrement) and used 8-bit fixed point arithmetic. Thus $\mathit{dec}(a) = max(0, \lfloor \frac{230a-1}{256} \rfloor)$.

The step evaluation function in the implementation is

$$\mathit{eval\text{-}sspace}(t, \Delta_{\mathrm{exec}}) = 256 \cdot \mathit{mem}(t, \Delta_{111}(\Delta_{\mathrm{exec}})) \tag{5}$$

where *mem* is as defined in equation (1). The effect of the decrement is that the shorter the sequence required for reaching the state from the state s, the more desirable the state can be. With our decrement function and the step evaluation function (5), the desirability of the state s is affected by the desirabilities of the states at most 56 steps away from s. This is because the value of a state can be at most 2511 (consider a state s having an input transition $t = (s, a, s)$ to itself, where $\mathit{eval\text{-}sspace}(t, \Delta_{\mathrm{exec}}) = 256$).

In evaluation of listening to the output the probability of outputs is estimated similarly to the adaptive player algorithm: $P[t]$ is defined in the equation (4). The value for output listening, stored into variable *oval*, is a weighted average of sums of $\mathit{eval}(t, \Delta_{\mathrm{exec}})$ and decreased values of the destination states of transitions t.

The value of the chosen state s is the greater of the values *best_input.value* and *oval*. If the value is different from $\mathit{value}[s]$, the new value is stored to the value table. All states from which there is a transition to the state s are added to the *Calc* set because their values may be affected by change of $\mathit{value}[s]$.

To avoid looping eternally, the *dec* function should be chosen in a way that the values of states do not grow without limit. To make the algorithm stop faster, we use coarse fixed point arithmetic: the value of the *mem* function is multiplied by 256 in the evaluation function and *dec* uses only integer part of the fraction.

In the test runs with the step evaluation function (5) the state space evaluation algorithm performed badly. The problem is that the algorithm rates a looping unexecuted transition equally good to a very long sequence of unexecuted transitions that leads to new states. We wrote the following step evaluation function to enhance the performance, although it does not solve the problem.

$$\mathit{eval\text{-}ss2}(t, \Delta_{\mathrm{exec}}) = 256 \cdot \mathit{mem}(t, \Delta_{111}(\Delta_{\mathrm{exec}})) + 512 \cdot \mathit{mem}(t, \Delta_{010}(\Delta_{\mathrm{exec}})) \tag{6}$$

4.6 Pyhälä-Heljanko

This algorithm has been implemented like presented in [14]. The algorithm aims to cover all transitions of the LTS.

Before every testing step the algorithm calls a *GreedyTestMove* subroutine with probability 0.75. If the subroutine is not called or it could not make the decision of the next step, the decision is made randomly. If there are input actions available, the random selection decides to send a random input (from $T_I(current\_state)$) with probability 0.5. Otherwise, the test engine listens to the output.

The *GreedyTestMove* checks first the transitions leaving the current state. If there are uncovered input and output transitions, then a random action of the uncovered input transitions is chosen with probability 0.5 and output is listened to with probability 0.5.

If there are uncovered input transitions but not uncovered output actions then a random uncovered input is selected. If there are uncovered output transitions but not uncovered input transitions then output is listened to.

If there are neither uncovered input transitions nor uncovered output transitions, then a bounded search in the LTS is made. The algorithm constructs as short sequence of actions $\sigma \in \Sigma^*$ as possible so that executing it takes LTS into a state s, from which uncovered transitions can be executed. The search is bounded so that the length of σ is at most ten actions. If a desired state is not found within the bound, *GreedyTestMove* could not make a decision and a random choice is made as described in the first place.

This algorithm resembles the player algorithms: both are greedy and use a bounded lookahead to the LTS to make decisions. There are still remarkable differences. This algorithm looks ahead only when there are no uncovered transitions in the current state and the look ahead is no deeper than the number of steps to the nearest desired state. The players look ahead before every step and always to the same depth. Furthermore, this algorithm is optimistic: when there is an uncovered output in the current state, the output is always listened to if there are no uncovered input transitions. The players are optimistic only when input actions are considered. During the test run, the adaptive player becomes more and more pessimistic about the output actions which have not been executed. The pessimistic player always expects the least desirable output to be received. Lastly, this algorithm is nondeterministic, whereas the players are deterministic.

5 Experiments

5.1 Implementation

The tested conference protocol implementation was downloaded from [5]. We tested the protocol in the service level through the UI included in the package [5]. The protocol provides a chatting service. A user can join a conference,

Command/ "Response"	Meaning
"Conference Protocol"	Response when a UI process is started
j *me cf*	Join conference *cf* with nickname *me*
s *m*	Send message *m* to current conference
l	Leave conference
">"	Prompt; the user is not in any conference
"*me*>"	Prompt; the user is in a conference with nickname *me*
"*se*> *m*"	The user received a message *m* from user with nickname *se*.

Table 1
Commands and responses

send messages to the conference it participates in (which causes all other users in the same conference to receive the messages), leave the conference, and start from the beginning. The commands and responses are listed in Table 1.

The implementation allows each user to participate in at most one conference at a time. The communication is not promised to be reliable (messages may be lost) but the messages should not be delivered to clients in other conferences.

We had five setups for SUTs. Two of them contained two client processes and the rest three client processes. Every setup contained one mutated client process, the other clients were correct implementations. The setups containing equal number of clients differed by the place in which the mutated client was running: sometimes client 1, sometimes client 2 or 3 was the broken one. With these setups we could get more variation into the results of the deterministic algorithms and thus make them more comparable with the nondeterministic ones.

All client processes and the test engine ran on the same machine (UltraSPARC IIe, 500 MHz, 512 MB of memory running Solaris 8 operating system). Client processes communicated through sockets (of datagram type), that provide connectionless and unreliable service.

5.2 Specification

We modelled the behaviour of the systems where two or three users communicate via the conference protocol. In the systems there are two conferences for the users to participate in. The users are called *CL1*, *CL2* and *CL3*, which they use as their nicknames when joining conferences. Users can send only one message, *M1*, to their conferences.

For simplicity it is specified that messages sent to conferences are never lost. That is, everyone present in the conference, except for the sender itself,

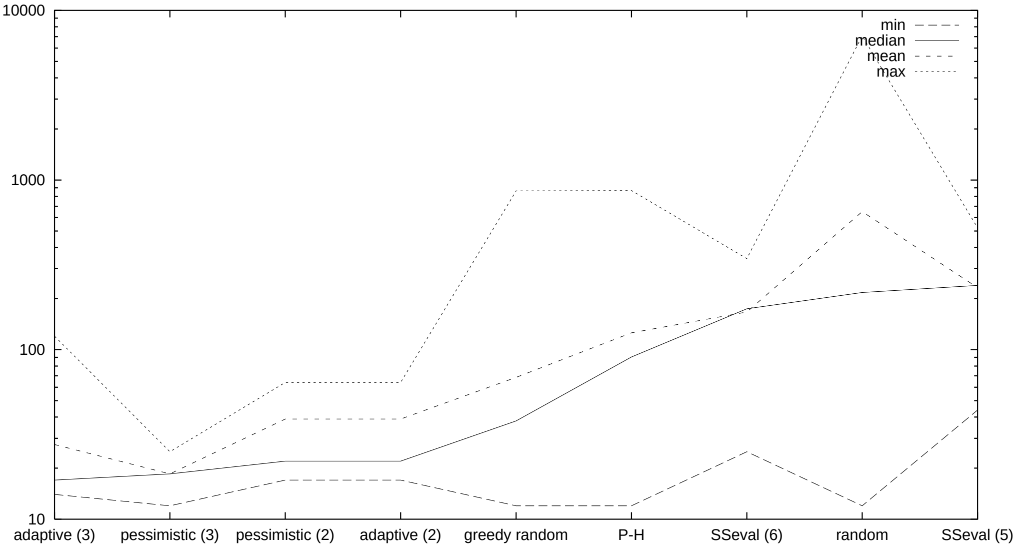

Fig. 2. The number of test steps before error was detected in the two client system

receives the sent messages. Therefore, sending messages is limited so that we can be sure whether or not each user should receive them. For example, when a user starts joining a conference, nobody can send messages to the conference before the user has received his prompt. Otherwise, we would not know if the user should receive the messages sent during the joining or not.

The systems of two and three clients were modelled with TVT (Tampere Verification Tool) [8]. The specifications are deterministic LTSs without τ transitions. The model with two clients[3] has 671 states, 24 visible actions and 1784 transitions. The other model[4] with three clients has 191995 states, 45 visible actions and 853218 transitions.

5.3 Results

We used nine different mutants of the client processes in the test setups. Every mutant was tested 32 times with each nondeterministic and once with each deterministic heuristic algorithm. The mutants are identified by the numbers explained in [5]. The mutants used in the results are 14, 17, 19, 21, 22, 24, 27, 28 and 60. Without going into the details of the mutated behaviours, their errors ranged from inability to receive certain PDUs to errors in updating internal data structures and errors in sending. The mutants were chosen so that they produce erroneous service. Some other mutants satisfied the specification.

The heuristic algorithms were compared by measuring the number of steps required to detect an error in the mutated systems. The results are presented in Figures 2 and 3. The algorithms are sorted by the median number of steps. Also the minimum, the maximum and the average number of steps are shown.

[3] *http://www.cs.tut.fi/~ask/MBT04/CL2_CF2_MSG1.lsts.gz*

[4] *http://www.cs.tut.fi/~ask/MBT04/CL3_CF2_MSG1.lsts.gz*

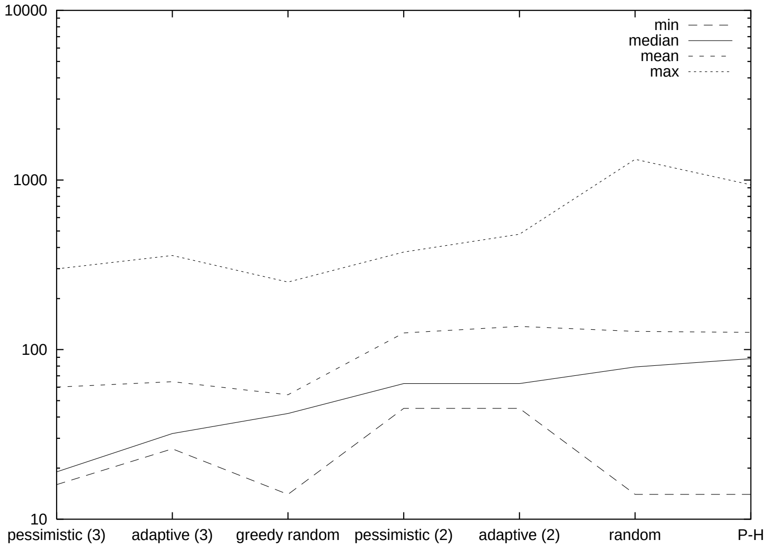

Fig. 3. The number of test steps before error was detected in the three client system

The test runs do not give means to estimate the differences of the state evaluation functions: the adaptive and the pessimistic players perform roughly equally well with the same step evaluation functions. We believe the reason was that the errors were detected too quickly. The adaptation to the SUT's behaviour in the states did not really take place.

Algorithms with step evaluation functions that included the coverage of action names (010) performed better than algorithms with simpler functions. Player algorithms with the function (3) were always better than with the function (2). The performance of the state space evaluation algorithm was also enhanced when its step evaluation function was switched from (5) to (6). But even with the enhancement, the state space evaluation algorithm performed poorly.

With the LTS of 671 states the state space evaluation algorithm runs faster than the player algorithms. With the LTS of 191995 states the implementation of the state space evaluation algorithm was too slow to be used. The algorithm was optimised so that only the states that are directly affected by the execution of a test step are recalculated, that is, put into the *Calc* set in the beginning of Algorithm 2. With step evaluation function (5) the only state that is added to *Calc* is the source state of the executed transition. With evaluation function (6) all the source states of transitions labelled by the executed action are inserted into *Calc*.

The effects of the determinism of the player algorithms can be seen in the figures. The gap between the minimal and the maximal values is relatively small and the median is very close to the other end (which happened to be the minimum in the test runs). The results would probably change if the players chose randomly the input action to be sent among equally good ones.

The greedy random algorithm worked surprisingly well. Even if it monitors only the transition coverage and does not calculate future steps, it worked better than the players with equally simple step evaluation function in the three client system. Probably nondeterminism would have helped the players in this case.

The greedy random outperformed the Pyhälä-Heljanko algorithm in both test runs. This may be due to the greedy random's eagerness to listen to the output whenever it is guaranteed to increase the coverage. Furthermore, when there is no such guarantee, the greedy randomly prefers sending inputs which are ensured to increase the coverage to listening to the outputs which does not necessary affect the coverage. Again, there was a clear gap between Pyhälä-Heljanko and pure random walk in the test setups with two clients, as it was also in [14].

6 Conclusions

We have presented how automatic guidance of testing can be based on covering the specification of the SUT.

The main test guidance algorithms presented in this paper include separate step and state evaluation parts. Heuristics like "prefer the untested input/output to the tested ones", "visit the unvisited states of the specification", "execute the unexecuted transitions" and "avoid executing the same transition too many times" can be implemented in the step evaluation part. The heuristics which predict the responses of the SUT belong to the state evaluation part.

In some cases, a finer step evaluation function could be useful in the heuristics. In [6] a distance between actions is proposed. The problem is that, given that action *datareq!CL1!CF1!M1* (client *CL1* sends message *M1* to conference *CF1*) is executed, unseen actions *datareq!CL1!CF1!M2* and *leave!CL1!CF1* (client 1 leaves conference 1) are considered equally interesting. Yet in this case it could be more interesting to leave the conference for the first time than send another message. Distances between actions would help estimating "How new an action is this?" If very similar actions have been already executed, then some other action (with greater distance) would be preferred.

In our test runs the distances between actions would not probably have much effect. There were only 24 or 45 actions. For example, clients can send only one kind of messages to conferences. Therefore executing new actions was equally interesting no matter what actions had been executed before. This may be often the case when the specification is written specifically for testing purposes. The situation is different if more data is bound in actions: a user could send, say, three messages instead of one and join five conferences instead of the two.

The test runs showed that when the errors are simple, the evaluation function that measures action name coverage in addition to the transition coverage detects errors faster.

Acknowledgements
Thanks for Jaco Geldenhuys and Antti Valmari for their valuable comments. This work is part of the SASOKE project funded by TEKES, Conformiq Software Oy Ltd and Nokia Research Center.

References

[1] Belinfante, A., Feenstra, J., de Vries, R. G., Tretmans, J., Goga, N., Feijs, L., Mauw, S. & Heerink, L.: "Formal Test Automation: A Simple Experiment". *International Workshop on Testing of Communication Systems*, Kluwer Academic, 1999, pp. 179–196.

[2] Chow, T. S.: "Testing Software Design Modeled by Finite-state Machines". *IEEE Transactions on Software Engineering*, SE-4(3), 1978, pp. 178–187.

[3] Fujivara, S.,v. Bochmann, G., Khendek, F., Amalou, M. & Ghedamsi, A.: "Test Selection Based on Finite State Models". *IEEE Transaction on Software Engineering*, Vol. 17(6), 1991, pp. 591–603.

[4] Conference Protocol Case Study, *http://fmt.cs.utwente.nl/ConfCase/.* Last updated by Jan Feenstra on 1999-11-19.

[5] Conference protocol implementation source code, *http://fmt.cs.utwente.nl/ ConfCase/v1.00/implementations/confprotv3c2.tgz.* Files in the package are dated 25–28 Feb 2002.

[6] Feijs, L. M. G., Goga, N., Mauw, S. & Tretmans, J.: "Test Selection, Trace Distance and Heuristics". *Proc. TestCom 2002, 14th International Conference on Testing of Communicating Systems*, Kluwer Academic Publishers 2002, pp. 267–282.

[7] de Vries, R.G. & Tretmans, J.: "On-the-fly conformance testing using SPIN". *International Journal on Software Tools for Technology Transfer*, 2(4): pp. 382–393, 2000.

[8] Hansen, H., Virtanen, H. & Valmari, A.: "Merging State-Based and Action-Based Verification". *Proc. ACSD 2003, 3rd International Conference on Application of Concurrency to System Design*, IEEE Computer Society 2003, pp. 150–156.

[9] Helovuo, J. & Leppänen, S.: "Exploration Testing". *Proc. ICACSD 2001, 2nd IEEE International Conference on Application of Concurrency to System Design*, IEEE Computer Society 2001, pp. 201–210.

[10] Lee, D., Sabnani, K. K., Kristol, D. M. & Paul, S.: "Conformance Testing of Protocols Specified as Communicating Finite State Machines—A Guided Random Walk Based Approach". *IEEE Transactions on Communications*, Vol. 44(5), 1996, pp. 631–640.

[11] Luo, G., v. Bochmann, G.: "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method". *IEEE Transactions on Software Engineering*, Vol. 20(2), 1994, pp. 149–161.

[12] Myers, G. J.: *The Art of Software Testing.* John Wiley & Sons, Inc, 1979.

[13] Naito, S. & Tsunoyama, M.: "Fault Detection for Sequential Machines by Transition Tour". *Proc. FTCS, Fault Tolerant Computer Systems*, 1981, pp. 238–243.

[14] Pyhälä, T. & Heljanko, K.: "Specification Coverage Aided Test Selection". *Proc. ACSD'2003, Third International Conference on Application of Concurrency to System Design*, IEEE 2003, pp. 187–195.

[15] Sabnani, K. K. & Dahbura, A.: "A Protocol Test Generation Procedure". *Computer Networks and ISDN Systems*, Vol. 15, 1988, pp. 285–297.

[16] Tretmans, J.: "Test Generation with Inputs, Outputs and Repetitive Quiescence". *Software—Concepts and Tools*, Vol. 17(3), Springer-Verlag 1996, pp. 103–120.

[17] Valmari, A.: "Composition and Abstraction". *Modelling and Verification of Parallel Processes*, Lecture Notes in Computer Science 2067, Springer-Verlag 2001, pp. 58–99.