Automatic Conformance Testing of Internet Applications

H.M.A. van Beek and S. Mauw

Department of Mathematics and Computer Science
Technische Unversiteit Eindhoven
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands
{harm,sjouke}@win.tue.nl

Abstract. We adapt and extend the theories used in the general framework of automated software testing in such a way that they become suitable for black-box conformance testing of thin client Internet applications. That is, we automatically test whether a running Internet application conforms to its formal specification. The actual implementation of the application is not taken into account, only its externally observable behaviour. In this paper, we show how to formally model this behaviour and how such formal specifications can serve as a basis for the automatic conformance testing of Internet applications.

1 Introduction

Since activity on the Internet is growing very fast, systems that are based on communication via the Internet appear more and more. To give an example, in the United States only, 45 billion dollar of products has been sold via the Internet in 2002 [1]. This is an increase of 38% compared to the on-line sales in 2001. Apart from the number of so-called Internet applications, the complexity of these applications increases too. This increasing complexity leads to a growing amount of errors in Internet applications, of which examples can be found at The Risks Digest [2], amongst others. This increasing number of errors asks for better testing of the applications and, preferably, this testing should be automated.

Research has been done in the field of automated testing of applications that are not based on Internet communication [3]. In this paper, we adapt and extend the theories used in the general framework of automated software testing in such a way that they become suitable for the testing of Internet applications.

We focus on black-box conformance testing of thin client Internet applications. That is, given a running application and a (formal) specification, our goal is to automatically test whether the implementation of the application conforms to the specification. Black box testing means that the actual implementation of the application is not taken into account but only its externally observable behaviour: We test *what* the application does, *not how* it is done. Interaction with the application takes place using the interface that is available to normal users of the application. In this case, the interface is based on communication via the Internet using the HTTP protocol [4].

As a start, in Section 2 the distinction between Web based and Window based applications is drawn. Next, in Section 3 we introduce how we plan to automatically test Internet applications. In Section 4, we describe the formalism we make use of for this automatically testing. To show the usefulness of the framework, we give a practical example in Section 5. We discuss related work in Section 6 and draw some final conclusions in Section 7.

2 Web based versus Window based Applications

In general, Web based applications, or Internet applications, behave like window based applications. They both communicate via a user interface with one or more clients. However, there are some major differences.

The Internet applications we focus on, are based on client-server communication via the Internet. The application runs on a server which is connected to the Internet. Via this connection, clients who are also connected to the Internet can interact with the application using prescribed protocols. Clients send requests over the Internet to the server on which the application runs. The server receives the requests and returns calculated responses.

In Figure 1 a schematic overview of the communication with Internet applications and window based applications is given. Clients interacting with window based applications are using a (graphical) user interface which is directly connected to the application. When interacting with Internet applications, the client sends an HTTP request [4] via the Internet, i.e. via some third parties, to the server. The server receives the request which subsequently is sent to the application. After receiving the request, the application calculates a response which is sent back to the requesting client. As can be seen in Figure 1, when testing an Internet application we have to take into account five entities, viz. clients, communication protocols, third parties, web servers and the application itself.

Clients The clients we focus on are so-called *thin clients*. This means that they have reduced or no possibility to do calculations. They make use of a centralised resource to operate. In the context of Internet applications, thin clients are usually web browsers. In general, more than one client can simultaneously access an Internet application. Unlike stand-alone applications, clients can fail, i.e. they can "disappear": a browser can simply be closed without notifying the application.

Dependency on third parties Since interaction takes place via the Internet, communication depends on third parties. First of all, packages transmitted go via routers which control the Internet traffic. It is not known which route on the world wide web is taken to get from the client to the server and back. Apart from transmitting the requests and responses, there are more dependencies, like DNS servers for translating domain names into IP addresses, trusted third parties for verifying certificates and e-mail servers for both the sending and receiving of e-mail messages.
Stand-alone applications usually do not depend on any of these parties.

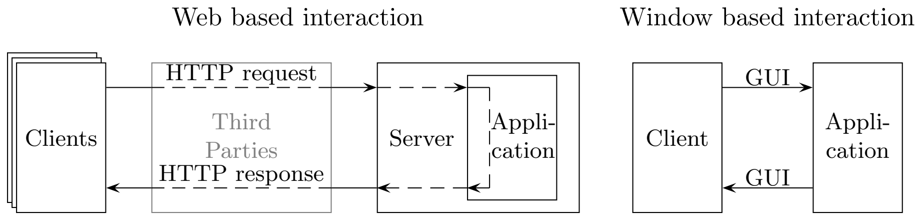

Fig. 1. Internet interaction versus stand-alone interaction.

Communication via the Internet Most of the communication with Internet applications we focus on is based on the HyperText Transfer Protocol (HTTP) [4]. This protocol is request-response based. A web server is waiting for requests from clients. As soon as a request comes in, the request is processed by an application running on the server. It produces a response which is sent back. Since the communication takes place via the Internet, delay times are unknown and communication can fail. Therefore, messages can overtake other messages.

Web servers A web server is a piece of hardware connected to the Internet. In contrast to stand-alone machines running a stand-alone application, a client might try to access a web server which is down or overtaxed, causing the interaction to fail.

Internet applications The Internet application itself is running on a web server. The applications we focus on, are based on request-response interaction with multiple clients. Since more than one client can interact with the application simultaneously, there might be a notion of who is communicating with the application. By keeping track of the interacting parties, requests and corresponding responses can be grouped into so-called *sessions*.

Main differences between Internet based and window based applications are the failing of clients and web servers, the failing of communication and overtaking of messages between clients and the application and the dependency on third parties. Furthermore, Internet applications are request-response based where window based applications interact with the clients using a (graphical) user interface. Finally, most Internet applications focus on parallel communication with more than one client. Since multiple clients can share a common state space, testing Internet applications is basically different from testing window based applications. Window based applications are mostly based on single user interaction. More differences between Web based and Window based systems can be found in e.g. [5].

3 Testing Internet applications

Now that we have a notion of what Internet applications look like, we informally show how implementations of these applications can be tested.

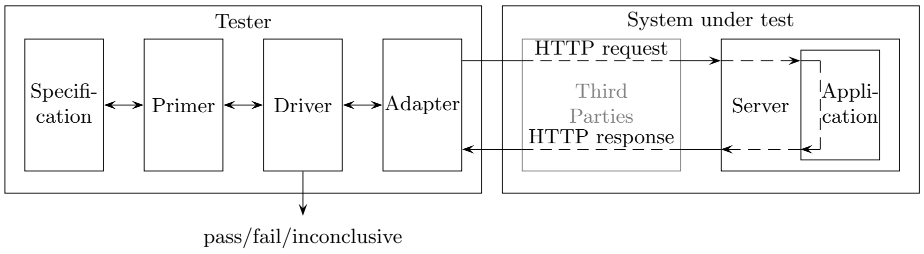

Fig. 2. Automatic testing of Internet applications.

We focus on black-box testing, restricting ourselves to *dynamic testing*. This means that the testing consists of really executing the implemented system. We do this by simulating real-life interaction with the applications, i.e. by simulating the clients that interact with the application. The simulated clients interact in a similar way as real-life clients would do. In this way, the application cannot distinguish between a real-life client and a simulated one. See Figure 2 for a schematic overview of the test environment.

We make use of a tester which generates requests and receives responses. This is called *test execution*. By observing the responses, the tester can determine whether they are expected responses in the specification. If so, the implementation passes the test, if not, it fails.

The tester itself consists of four components, based on [6]:

Specification The specification is the formal description of how the application under test is expected to behave.

Primer The primer determines the requests to send by inspecting the specification and the current state the test is in. So the primer interacts with the specification and keeps track of the test's state. Furthermore, the primer checks whether responses received by the tester are expected responses in the specification at the state the test is in.

Driver The driver is the central unit, controlling the execution of the tests. This component determines what actions to execute. Furthermore, the verdict whether the application passes the test is also computed by the driver.

Adapter The adapter is used for encoding abstract representations of requests into HTTP requests and for decoding HTTP responses into abstract representations of these responses.

While executing a test, the driver determines if a request is sent or a response is checked. If the choice is made to send a request, the driver asks the primer for a correct request, based on the specification. The request is encoded using the adapter and sent to the application under test. If the driver determines to check a response, a response is decoded by the adapter. Next, the primer is asked whether the response is expected in the specification. Depending on the results, a verdict can be given on the conformance of the implementation to its specification.

As mentioned in Section 2, clients, web servers, their mutual communication and third parties can fail. In such a case, no verdict can be given on the correctness of the implementation of the Internet application. However, depending on the failure, it might be possible to determine the failing entity.

4 Conformance Testing of Internet Applications

As a basis for conformance testing of Internet applications, we take the formal framework as introduced in [7–9]. Given a specification, the goal is to check, by means of testing, whether an implemented system satisfies its specification. To be able to formally test applications, there is a need for implementations and formal specifications. Then, *conformance* can be expressed as a relation on these two sets.

Implementations under test are real objects which are treated as black boxes exhibiting behaviour and interacting with their environment. They are not amenable to formal reasoning, which makes it harder to formally specify the conformance relation. Therefore, we make the assumption that any implementation can be modelled by a formal object. This assumption is referred to as the *test hypothesis* [10] and allows us to handle implementations as formal objects. We can express conformance by a formal relation between a model of an implementation and a specification, a so-called *implementation relation*.

An implementation is tested by performing experiments on it and observing its reactions to these experiments. The specification of such an experiment is called a *test case*, a set of test cases a *test suite*. Applying a test to an implementation is called *test execution* and results in a verdict. If the implementation passes or fails the test case, the verdict will be *pass* or *fail*, respectively. If no verdict can be given, the verdict will be *inconclusive*.

In the remainder of this section, we will instantiate the ingredients of the framework as sketched above. We give a formalism for both modelling implementations of Internet applications and for giving formal specifications used for test generation. Furthermore, we give an implementation relation. By doing this, we are able to test whether a (model of an) implementation conforms to its specification. Apart from that, we give an algorithm for generating test suites from specifications of Internet applications.

4.1 Modelling Internet Applications

To be able to formally test Internet applications, we need to formally model their behaviour. Since we focus on conformance testing, we are mainly interested in the communication between the application and its users. We do not focus on the representation of data. Furthermore, we focus on black-box testing, which means that the internal state of applications is not known in the model. Finally, we focus on thin client Internet applications that communicate using the HyperText Transfer Protocol (HTTP) [4]. As a result, the applications show a request/response behaviour.

These observations lead to modelling Internet applications using labelled transition systems. Each transition in the model represents a communication action between the application and a client. The precise model is dictated by the interacting behaviour of the HTTP protocol.

In general, an HTTP interaction is initiated by a client, sending a request for some information to an application. A request can be extended with parameters. These parameters can be used by the application. After calculating a response, it is sent back to the requesting client. Normally, successive requests are not grouped. However, the grouping can be done by adding parameters to the requests and responses. In such a way, alternating sequences of requests and responses are turned into sessions.

Note that we test the interaction behaviour of Internet applications communicating via HTTP. We do *not* model the client-side tools to interact with Internet applications, i.e., we do not model the behaviour of the application when using browser buttons like *stop*, *back*, *forward* and *refresh*. Main reason for not including this behaviour is that different client implementations cause distinct interaction behaviour.

Furthermore, we do not add (failure of) components in the system under test other than the application to the specification. This means that failure of any of these components leads to tests in which the result will be *inconclusive*. If all components in the system under test operate without failure, verdicts will be pass or fail.

The tester should behave like a set of thin clients. The only requests sent to the application are the initial request which models the typing in of a URL in the browser's address bar and requests that result from clicking on links or submitting forms which are contained in preceding responses.

Since we focus on HTTP based Internet applications, and thus on sessions of alternating request-response communication with applications, we make use of so-called *multi request-response transition systems* (MRRTSs) for both modelling implementations of Internet applications and giving formal specifications used for test generation. An MRRTS is a labelled transition system having extra structure. In the remainder of this section we explain MRRTSs in more detail and show how they relate to labelled transition systems and *request-response transition systems* (RRTSs).

Labelled Transition Systems The formalism of *labelled transition systems* is widely used for describing the behaviour of processes. We will provide the relevant definitions.

Definition 1. *A* labelled transition system *is a 4-tuple* $\langle S, L, \rightarrow, s_0 \rangle$ *where*

- S *is a countable, non-empty set of* states*;*
- L *is a countable set of* labels*;*
- $\rightarrow \subseteq S \times L \times S$ *is the* transition relation*;*
- $s_0 \in S$ *is the* initial state*.*

Definition 2. *Let s_i $(i \in \mathbb{N})$ be states and a_i $(i \in \mathbb{N})$ be labels. A (finite) composition of transitions*

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots s_n \xrightarrow{a_n} s_{n+1}$$

is then called a computation. *The sequence of actions of a computation, $a_1 \cdot a_2 \cdot \ldots \cdot a_n$, is called a* trace. *The empty trace is denoted by ε. If L is a set of labels, the set of all finite traces over L is denoted by L^*.*

Definition 3. *Let $p = \langle S, L, \rightarrow, s_0 \rangle$, $s, s' \in S$, $S' \subseteq S$, $a_i \in L$ and $\sigma \in L^*$. Then,*

$$\begin{aligned}
s \xrightarrow{a_1 \cdot \ldots \cdot a_n} s' &=_{\text{def}} \exists s_1, \ldots, s_{n-1}\ s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots s_{n-1} \xrightarrow{a_n} s' \\
s \xrightarrow{a_1 \cdot \ldots \cdot a_n} &=_{\text{def}} \exists s'\ s \xrightarrow{a_1 \cdot \ldots a_n} s' \\
\text{init}(s) &=_{\text{def}} \{a \in L \mid s \xrightarrow{a}\} \\
\text{traces}(s) &=_{\text{def}} \{\sigma \in L^* \mid s \xrightarrow{\sigma}\} \\
\text{traces}(S') &=_{\text{def}} \bigcup s' \in S'\ \text{traces}(s') \\
s\ \textbf{after}\ \sigma &=_{\text{def}} \{s' \in S \mid s \xrightarrow{\sigma} s'\} \\
S'\ \textbf{after}\ \sigma &=_{\text{def}} \bigcup s' \in S'\quad s'\ \textbf{after}\ \sigma
\end{aligned}$$

A labelled transition system $p = \langle S, L, \rightarrow, s_0 \rangle$ will be identified by its initial state s_0. So, e.g., we can write traces(p) instead of traces(s_0) and p **after** σ instead of s_0 **after** σ.

We aim at modelling the behaviour of the HTTP protocol using labelled transition systems. Therefore, we need to add restrictions on the traces in the labelled transition system used for modelling this behaviour. One of these restrictions is that traces in the LTSs should answer the alternating request/response behaviour.

Definition 4. *Let A, B be sets of labels. Then* alt(A, B) *is the (infinite) set of traces having alternating structure with respect to elements in A and B, starting with an element in A. Formally,* alt(A, B) *is the smallest set such that*

$$\varepsilon \in \text{alt}(A, B) \ \wedge\ \forall \sigma \in \text{alt}(B, A) \forall a \in A\ a\sigma \in \text{alt}(A, B)\ .$$

As mentioned before, interactions with an Internet application can be grouped into sessions. To be able to specify the behaviour within each session, we make use of a projection function. This function will be used for determining all interactions contained within one session.

Definition 5. *Let σ be a trace and A be a set of labels. Then $\sigma\,|_A$, the projection of σ to A, is defined by*

$$\begin{aligned}
\varepsilon\,|_A &=_{\text{def}} \varepsilon \\
(a \cdot \sigma)\,|_A &=_{\text{def}} \begin{cases} a \cdot (\sigma\,|_A) & \text{if } a \in A \\ \sigma\,|_A & \text{if } a \notin A\ . \end{cases}
\end{aligned}$$

Definition 6. *A* partitioning *S of a set A is a collection of mutually disjoint subsets of A such that their union exactly equals A:*

$$\bigcup S = A \quad \wedge \quad \forall B, C \in S\ B \neq C \Rightarrow B \cap C = \emptyset$$

Request-Response Transition Systems We give a formal definition of a request-response transition system, denoted by RRTS. RRTSs can be compared to input-output transitions systems (IOTSs) [11]. As in IOTSs, we differentiate between two sets of labels, called *request labels* and *response labels*, respectively. RRTSs are based on pure request/response alternation.

Definition 7. *Let L be a countable set of labels and $\{L_?, L_!\}$ be a partitioning of L. Then, a* request-response transition system $\langle S, L_?, L_!, \rightarrow, s_0\rangle$ *is a labelled transition system $\langle S, L, \rightarrow, s_0\rangle$ such that*

$$\forall \sigma \in \mathrm{traces}(s_0) \quad \sigma \in \mathrm{alt}(L_?, L_!) \ .$$

Elements in $L_?$ are called request labels*, elements in $L_!$* response labels*.*

RRTSs resemble the notion of Mealy machines, however, it turns out to be technically adhered to start from the notion of RRTSs.

Multi Request-Response Transition Systems IOTSs can be used as a basis for multi input-output transition systems (MIOTSs) [12]. Similarly, in a multi request-response transition system (MRRTS), multiple request-response transition systems are combined into one. All subsystems behave like an RRTS, however interleaving between the subsystems is possible.

Definition 8. *Let L be a countable set of labels. Let $\mathbb{L} \subseteq \mathcal{P}(L) \times \mathcal{P}(L)$ be a countable set of tuples such that $\{A, B \mid (A, B) \in \mathbb{L}\}$ is a partitioning of L. Then, a* multi request-response transition system $\langle S, \mathbb{L}, \rightarrow, s_0\rangle$ *is a labelled transition system $\langle S, L, \rightarrow, s_0\rangle$ such that*

$$\forall (A, B) \in \mathbb{L}\ \forall \sigma \in \mathrm{traces}(s_0) \quad \sigma\,|_{A \cup B} \in \mathrm{alt}(A, B) \ .$$

The set of all possible request labels, $\mathbb{L}_?$, is defined by

$$\mathbb{L}_? \quad =_{\mathrm{def}} \quad \bigcup_{(A,B)\in\mathbb{L}} A \ .$$

The set of all possible response labels, $\mathbb{L}_!$, is defined by

$$\mathbb{L}_! \quad =_{\mathrm{def}} \quad \bigcup_{(A,B)\in\mathbb{L}} B \ .$$

Note that an RRTS $\langle S, L_?, L_!, \rightarrow, s_0\rangle$ can be interpreted as MRRTS $\langle S, \{(L_?, L_!)\}, \rightarrow, s_0\rangle$, i.e., each MRRTS having singleton $\mathbb{L}$ is an RRTS.

We introduce some extra functions on the sets of tuples as introduced in Definition 8.

Definition 9. *Let $\mathbb{L} \subseteq \mathcal{P}(L) \times \mathcal{P}(L)$ be a countable set of tuples, where each tuple contains a set of request labels and a set of response labels. We define functions for determining corresponding requests or responses given either a request label or response label. For $x \in L$, we define functions* $\mathrm{req}, \mathrm{resp} : L \rightarrow \mathcal{P}(L)$*, such that*

$$(\mathrm{req}(x), \mathrm{resp}(x)) \in \mathbb{L} \quad \textit{and} \quad x \in \mathrm{req}(x) \cup \mathrm{resp}(x) \ .$$

4.2 Relating Multi Request-Response Transition Systems

An implementation conforms to a specification if an implementation relation exists between the model of the implementation and its specification. We model both the implementation and the specification as multi request-response transition systems, so conformance can be defined by a relation on MRRTSs.

While testing Internet applications, we examine the responses sent by the application and check whether they are expected responses by looking at the specification. So we focus on testing whether the implementation does what it is expected to do, not what it is not allowed to do.

Given a specification, we make use of function exp to determine the set of expected responses in a state in the specification.

Definition 10. *Let p be a multi request-response transition system $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. For each state $s \in S$ and for each set of states $S' \subseteq S$, the set of expected responses in s and S' is defined as*

$$\begin{aligned} \exp(s) &=_{\text{def}} \text{init}(s) \cap \mathbb{L}_! \\ \exp(S') &=_{\text{def}} \bigcup s' \in S' \quad \exp(s') \ . \end{aligned}$$

If a model of an implementation i conforms to a specification s, the possible responses in all reachable states in i should be contained in the set of possible responses in the corresponding states in s. Corresponding states are determined by executing corresponding traces in both i and s.

Definition 11. *Let* MRRTS *i be the model of an implementation and* MRRTS *s be a specification. Then i conforms to s with respect to request-response behaviour, i* **rrconf** *s, if and only if all responses of i are expected responses in s:*

$$i \ \textbf{rrconf} \ s \quad =_{\text{def}} \quad \forall \sigma \in \text{traces}(s) \quad \exp(i \ \textbf{after} \ \sigma) \subseteq \exp(s \ \textbf{after} \ \sigma) \ .$$

Relation **rrconf** on MRRTSs is analogous to relation **conf** on LTSs as formalised in [13].

4.3 Test Derivation

An implementation is tested by performing experiments on it and observing its reactions to these experiments. The specification of such an experiment is called a *test case*. Applying a test to an implementation is called *test execution*. By now we have all elements for deriving such test cases.

Since the specification is modelled by an MRRTS, a test case consists of request and response actions as well. However, we have some more restrictions on test cases. First of all, test cases should have finite behaviour to guarantee that tests terminate. Apart from that, unnecessary nondeterminism should be avoided, i.e., within one test case the choice between multiple requests or between requests and responses should be left out.

In this way, a test case is a labelled transitions system where each state is either a terminating state, a state in which a request is sent to the implementation under test, or a state in which a response is received from the implementation. The terminating states are labelled with a verdict which is a **pass** or **fail**.

Definition 12. *A* test case t *is an LTS* $\langle S, \mathbb{L}_? \cup \mathbb{L}_!, \rightarrow, s_0\rangle$ *such that*

- *t is deterministic and has finite behaviour;*
- *S contains terminal states* **pass** *and* **fail** *with* $\mathrm{init}(\mathbf{pass}) = \mathrm{init}(\mathbf{fail}) = \emptyset$*;*
- *for all* $s \in S \setminus \{\mathbf{pass}, \mathbf{fail}\}$, $\mathrm{init}(s) = \{a\}$ *for* $a \in \mathbb{L}_?$ *or* $\mathrm{init}(s) = \mathbb{L}_!$*.*

We denote this subset of LTSs by TESTS*. A set of test cases* $T \subseteq$ TESTS *is called a* test suite*.*

We do not include the possibility for reaching *inconclusive* states in test cases. Such verdicts are given if a component in the system under test, other than the application, fails. The tester (as described in Section 3) is able to identify errors caused by the application and lead to a **fail** state. Other errors result in an inconclusive verdict.

As mentioned, we call a set of test cases a *test suite*. Such a test suite is used for determining whether an implementation conforms to a specification. A test suite T is said to be *sound* if and only if all implementations that conform to the specification pass all test cases in T. If all implementations that do not conform to the specification fail a test case in T, T is called *exhaustive*. Test suite that are both sound and exhaustive are said to be *complete* [9].

Definition 13. *Let* MRRTS *i be an implementation and T be a test suite. Then, implementation i passes test suite T if no traces in i lead to a fail state:*

$$i \ \mathbf{passes}\ T \quad =_{\mathrm{def}} \quad \neg\exists t \in T\ \exists \sigma \in \mathrm{traces}(i) \quad \sigma \cdot \mathbf{fail} \in \mathrm{traces}(t)$$

We use the notation $\sigma \cdot \mathbf{fail}$ to represent trace σ leading to a fail state, i.e., $\sigma \cdot \mathbf{fail} \in \mathrm{traces}(t) =_{\mathrm{def}} t \xrightarrow{\sigma} \mathbf{fail}$.

Definition 14. *Let s be a specification and T be a test suite. Then for relation* **rrconf***:*

$$\begin{array}{lll} T \textit{ is sound} & =_{\mathrm{def}} & \forall i \quad i \ \mathbf{rrconf}\ s \Longrightarrow i \ \mathbf{passes}\ T \\ T \textit{ is exhaustive} & =_{\mathrm{def}} & \forall i \quad i \ \mathbf{rrconf}\ s \Longleftarrow i \ \mathbf{passes}\ T \\ T \textit{ is complete} & =_{\mathrm{def}} & \forall i \quad i \ \mathbf{rrconf}\ s \Longleftrightarrow i \ \mathbf{passes}\ T \end{array}$$

In practice, however, such a complete test suite will often be infinitely large, and therefore not suitable. So, we have to restrict ourselves to test suites for detecting non-conformance instead of test suites for giving a verdict on the conformance of the implementation. Such test suites are called *sound*.

To test conformance with respect to request-response behaviour, we have to check for all possible traces in the specification that the responses generated by the implementation are expected responses in the specification. This can be done by having the implementation execute traces from the specification. The responses of the implementation are observed and compared with the responses expected in the specification. Expected responses pass the test, unexpected responses fail the test. The algorithm given is based on the algorithm for generating test suites as defined in [14].

Algorithm 1. Let s be MRRTS $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. Let C be a non-empty set containing all possible states of the specification in which the implementation can be at the current stage of the test. Initially $C = \{s_0\}$. We then define the collection of nondeterministic recursive algorithms gentest^n ($n \in \mathbb{N}$) for deriving test cases as follows:

$$\text{gentest}^n \quad : \quad \mathcal{P}(S) \rightarrow \text{TESTS}$$

```
$\text{gentest}^n(C) =_{\text{def}}$ [ return $\mathbf{pass}$
                   [] $n > 0 \wedge a \in \mathbb{L}_? \ \wedge\ C\ \mathbf{after}\ a \neq \emptyset \quad \rightarrow$
                        return $a \cdot \text{gentest}^{n-1}(C\ \mathbf{after}\ a)$
                   [] $n > 0 \quad \rightarrow$
                        return $\sum\{b \cdot \mathbf{fail} \ \mid\ b \in \mathbb{L}_! \setminus \exp(C)\}$
                         $+\ \sum\{b \cdot \text{gentest}^{n-1}(C\ \mathbf{after}\ b) \ \mid\ b \in \exp(C)\}$
                   ]
```

The $\cdot$ infix notation is used for sequential composition. So, e.g., $a \cdot b$ relates to transitions $s \xrightarrow{a} s' \xrightarrow{b} s''$. As mentioned, notation $a \cdot \mathbf{pass}$ and $a \cdot \mathbf{fail}$ is used for representing transitions $s \xrightarrow{a} \mathbf{pass}$ and $s \xrightarrow{a} \mathbf{fail}$, respectively. We use Σ-notation to indicate that it is not known which of the responses is returned by the implementation. So, e.g. $a + b$ relates to transitions $s \xrightarrow{a} s'$ and $s \xrightarrow{b} s''$. Depending on whether the response is expected, the algorithm might either continue or terminate in a **fail** state.

Although a choice for the first option can be made in each step, we added a parameter to the algorithm, $n \in \mathbb{N}$, to force termination. As mentioned, we want all test cases to be finite, since otherwise no verdict might take place.

The set of derivable test cases from gentest$^n(C)$ is denoted by $\overline{\text{gentest}^n(C)}$. So $\overline{\text{gentest}^n(C)}$ is the set of all possible test cases of at most n transitions starting in states C of the specification. Although our goal is to generate sound test suites, we will prove that in the limit, as n reaches infinity, test suite $\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})}$ is complete for specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. To prove this, we make use of some lemmas.

Lemma 1. *Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$ and $\sigma_0, \sigma_1 \in L^*$, $\sigma_1 \neq \varepsilon$. Then*

$$\sigma_0\sigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \quad \Longleftrightarrow \quad \sigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\sigma_0|}(s_0\ \mathbf{after}\ \sigma_0)})$$

where $|\sigma|$ is the length of trace σ.

Sketch of proof. This lemma can be proved by using induction on the structure of σ_0.

Lemma 2. *Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$, $\sigma_0 \in L^*$ and $n > 0$. Then*

$$\sigma \cdot \mathbf{fail} \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \quad \Longrightarrow \quad \exists \sigma' \in L^* \exists b \in \mathbb{L}_! \ \sigma = \sigma' b \ .$$

Proof. This can be easily seen by looking at the definition of the gentest algorithm: State **fail** can only be reached after execution of a $b \in \mathbb{L}_!$.

□

Theorem 1. *Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. Then test suite $\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})}$ is complete.*

Proof. Let s be $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$ and T be $\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})}$. Then,

$$\begin{array}{ll}
 & T \text{ is complete} \\
\equiv & \{ \text{ definition of complete test suites } \} \\
 & \forall i \quad i \ \mathbf{rrconf} \ s \ \Leftrightarrow \ i \ \mathbf{passes} \ T \\
\equiv & \{ \text{ definition of } \mathbf{rrconf} \text{ and } \mathbf{passes} \ \} \\
 & \forall i \quad \forall \sigma \in \text{traces}(s) \ \text{exp}(i \ \mathbf{after} \ \sigma) \subseteq \text{exp}(s \ \mathbf{after} \ \sigma) \\
 & \qquad \Longleftrightarrow \\
 & \qquad \neg \exists t \in T \ \exists \sigma \in \text{traces}(i) \ \sigma \cdot \mathbf{fail} \in \text{traces}(t)
\end{array}$$

We prove this by proving exhaustiveness ($\Leftarrow$) and soundness ($\Rightarrow$) separately.

– Exhaustiveness.

$$\begin{array}{l}
\forall i \ \forall \sigma \in \text{traces}(s) \ \text{exp}(i \ \mathbf{after} \ \sigma) \subseteq \text{exp}(s \ \mathbf{after} \ \sigma) \\
\qquad \Longleftarrow \\
\qquad \neg \exists t \in T \ \exists \sigma \in \text{traces}(i) \ \sigma \cdot \mathbf{fail} \in \text{traces}(t)
\end{array}$$

We prove exhaustiveness by contradiction:
Let $\sigma \in \text{traces}(s)$ and $b \in \text{exp}(i \ \mathbf{after} \ \sigma)$ such that $b \notin \text{exp}(s \ \mathbf{after} \ \sigma)$. Then, we prove that $\exists t \in T \ \exists \sigma' \in \text{traces}(i) \ \sigma' \cdot \mathbf{fail} \in \text{traces}(t)$.

$$\begin{array}{ll}
 & \exists t \in T \ \exists \sigma' \in \text{traces}(i) \ \sigma' \cdot \mathbf{fail} \in \text{traces}(t) \\
\Leftarrow & \{ \ b \in \text{exp}(i \ \mathbf{after} \ \sigma) \Rightarrow \sigma b \in \text{traces}(i), \ \text{Let } \sigma' = \sigma \cdot b \ \} \\
 & \exists t \in T \ \sigma \cdot b \cdot \mathbf{fail} \in \text{traces}(t) \\
\Leftarrow & \{ \text{ Definition of T } \} \\
 & \exists n > 0 \ \sigma \cdot b \cdot \mathbf{fail} \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\
\equiv & \{ \text{ Lemma 1 } \} \\
 & \exists n > 0 \ b \cdot \mathbf{fail} \in \text{traces}(\overline{\text{gentest}^{n-|\sigma|}(\{s_0 \ \mathbf{after} \ \sigma\})}) \\
\Leftarrow & \left\{ \begin{array}{l} \text{gentest (third option), let } n > |\sigma|, \\ \quad b \notin \text{exp}(s_0 \ \mathbf{after} \ \sigma) \Rightarrow b \in \mathbb{L}_! \setminus \text{exp}(s_0 \ \mathbf{after} \ \sigma) \end{array} \right\} \\
 & \text{true}
\end{array}$$

– Soundness.

$$\begin{array}{ll}
\forall i & \forall \sigma \in \text{traces}(s) \ \text{exp}(i \ \mathbf{after} \ \sigma) \subseteq \text{exp}(s \ \mathbf{after} \ \sigma) \\
 & \Longrightarrow \\
 & \neg \exists t \in T \ \exists \sigma \in \text{traces}(i) \ \sigma \cdot \mathbf{fail} \in \text{traces}(t)
\end{array}$$

Soundness is also proved by contradiction:
Let $t \in T$ and $\sigma \in \text{traces}(i)$ such that $\sigma \cdot \textbf{fail} \in \text{traces}(t)$. Then, by definition of T, $\exists n > 0\ \sigma \cdot \textbf{fail} \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})})$. Let $m > 0$ such that $\sigma \cdot \textbf{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})})$. We prove that $\exists \sigma' \in \text{traces}(s) \exists b \in \exp(i \textbf{ after } \sigma')\ b \notin \exp(s \textbf{ after } \sigma')$.
Let $\sigma'' \in \text{traces}(s)$ and $b'' \in \exp(i \textbf{ after } \sigma'')$. Then, we prove that $b'' \notin \exp(s \textbf{ after } \sigma'')$. Since $\sigma \cdot \textbf{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})})$, using Lemma 2, $\exists \sigma' \in \text{traces}(s) \exists b \in \mathbb{L}_! \ \sigma = \sigma' \cdot b$. Let $\sigma = \sigma'' \cdot b''$. Then,

$$\begin{aligned}
& \sigma \cdot \textbf{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})}) \\
\equiv\ & \{\ \sigma = \sigma'' \cdot b''\ \} \\
& \sigma'' \cdot b'' \cdot \textbf{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})}) \\
\equiv\ & \{\ \text{Lemma 1}\ \} \\
& b'' \cdot \textbf{fail} \in \text{traces}(\overline{\text{gentest}^{m-|\sigma''|}(s_0 \textbf{ after } \sigma'')}) \\
\Rightarrow\ & \{\ \text{Definition of algorithm gentest (third option)}\ \} \\
& b'' \in \mathbb{L}_! \setminus \exp(s_0 \textbf{ after } \sigma'') \\
\Rightarrow\ & \{\ \text{Set theory}\ \} \\
& b'' \notin \exp(s_0 \textbf{ after } \sigma'')
\end{aligned}$$

□

Algorithm 1 can easily be optimised. As can be seen by looking at the algorithm, each choice for inspecting a response of the implementation leads to $|\mathbb{L}_!|$ new branches in the generated test case. However, many of these branches will never take place as a result of the alternating request-response behaviour of the implementation: the implementation can only send responses on requests sent by the tester. It can be proved that by adding only this restricted set of responses to the test cases, such optimised generated test suites are still complete.

5 Example: Internet Vote

We show how the theory introduced in former sections can be used for testing real-life Internet applications. As an example, we take a voting protocol. All members of a group of voters are asked whether they are for or against a proposition. They are able to visit a web site where they can either vote or have a look at the current score. They can vote at most once and they can check the score as often as they want to.

We start by giving an MRRTS that formally specifies the application. Let V be the set of voters and $\mathbb{P} = \{\text{for}, \text{against}\}$. Then,

– $\mathbb{L}$, the set of tuples of transition labels is defined as

$$\begin{aligned}
\mathbb{L} = \ & \{\ (\ \{\text{vote}(v,p)_{?s} \mid p \in \mathbb{P}\}, \{\text{ok}_{!s}, \neg\text{ok}_{!s}\}\ ) \mid v \in V, s \in \mathbb{N}\ \} \\
\cup\ & \{\ (\ \{\text{score}_{?s}\}, \{\text{score}(f,a)_{!s}\}\ ) \mid f, a, s \in \mathbb{N}\ \}\ .
\end{aligned}$$

The first part specifies the interactions where voter v sends a request to vote for or against the proposition (p). The response is a confirmation (ok) or a denial ($\neg$ok), depending on whether the voter had voted before. The second part specifies the requests for the score which are responded by the number of votes for (f) and against (a) the proposition. All labels are extended with an identifier (s) for uniquely identifying the sessions.

- The set of states S is defined as $S = \mathcal{P}(\mathbb{L}_?) \times \mathcal{P}(V) \times \mathcal{P}(V) \times \mathcal{P}(\mathbb{N} \times \mathbb{N} \times \mathbb{N})$. For $\langle R, F, A, C \rangle \in S$,
  - $R \subseteq \mathbb{L}_?$ is the set of requests on which no response has been sent yet;
  - $F \subseteq V$ is the set of voters who voted for the proposition;
  - $A \subseteq V$ is the set of voters who voted against the proposition;
  - $C \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ is the score at the moment that a request for the score is sent. We need to keep track of this score for determining the possible results that can be responded: The scores returned should be at least the scores at the time of the sending of the request.
    For $\langle s, f, a, \rangle \in C$,
    * $s \in \mathbb{N}$ is the session identifier;
    * $f \in \mathbb{N}$ is the number of voters who voted for the proposition;
    * $a \in \mathbb{N}$ is the number of voters who voted against the proposition.
- Let $s \in \mathbb{N}$, $v \in V$ and $p \in \mathbb{P}$. Then, transition relation $\rightarrow$ is defined by the following derivation rules.
  If no session exists with identifier s, a session can be started by sending a request to vote for or against the proposition or by sending a request for the current score:

$$\frac{\text{score}_{?s} \notin R, \quad \neg\exists_{w \in V}\exists_{q \in \mathbb{P}}\, \text{vote}(w,q)_{?s} \in R}{\langle R, F, A, C \rangle \xrightarrow{\text{vote}(v,p)_{?s}} \langle R \cup \{\text{vote}(v,p)_{?s}\}, F, A, C \rangle}$$

$$\frac{\text{score}_{?s} \notin R, \ \neg\exists_{w \in V}\exists_{q \in \mathbb{P}}\, \text{vote}(w,q)_{?s} \in R}{\langle R, F, A, C \rangle \xrightarrow{\text{score}_{?s}} \langle R \cup \{\text{score}_{?s}\}, F, A, C \cup \{\langle s, |F|, |A| \rangle\} \rangle}$$

If a request to vote for or against the proposition has been sent and the voter has not voted before, the vote can be processed and confirmed:

$$\frac{\text{vote}(v,\text{for})_{?s} \in R, \ v \notin F \cup A}{\langle R, F, A, C \rangle \xrightarrow{\text{ok}_{!s}} \langle R \setminus \{\text{vote}(v,\text{for})_{?s}\}, F \cup \{v\}, A, C \rangle}$$

$$\frac{\text{vote}(v,\text{against})_{?s} \in R, \ v \notin F \cup A}{\langle R, F, A, C \rangle \xrightarrow{\text{ok}_{!s}} \langle R \setminus \{\text{vote}(v,\text{against})_{?s}\}, F, A \cup \{v\}, C \rangle}$$

If a request to vote has been sent and the voter has already voted before or the voter is concurrently sending a request to vote in another session, the vote can be rejected:

$$\frac{\begin{array}{c}\text{vote}(v,p)_{?s} \in R, \\ \neg\exists_{t \in \mathbb{N} \setminus \{s\}}\exists_{q \in \mathbb{P}}\, \text{vote}(v,q)_{?t} \in R \vee v \in F \cup A\end{array}}{\langle R, F, A, C \rangle \xrightarrow{\neg\text{ok}_{!s}} \langle R \setminus \{\text{vote}(v,p)_{?s}\}, F, A, C \rangle}$$

If a request for the score has been sent, the scores can be sent to the requesting client. Since interactions can overtake each other, the result can be any of the scores between the sending of the request and the receiving of the response. So, the score must be at least the score at the moment of requesting the score and at most the number of processed votes plus the number of correct votes, sent in between requesting for the score and receiving the score:

$$\frac{\begin{array}{c}\text{score}_{?s} \in R,\ \langle s, f, a\rangle \in C,\\ f \leq f' \leq |F| + (\#_{v\in V} \exists_{t\in\mathbb{N}}\, \text{vote}(v, \text{for})_{?t} \in R \wedge v \notin F \cup A),\\ a \leq a' \leq |A| + (\#_{v\in V} \exists_{t\in\mathbb{N}}\, \text{vote}(v, \text{against})_{?t} \in R \wedge v \notin F \cup A)\end{array}}{\langle R, F, A, C\rangle \xrightarrow{\text{score}(f',a')_{!s}} \langle R \setminus \{\text{score}_{?s}\}, F, A, C \setminus \{\langle s, f, a\rangle\}\rangle}$$

– Initial state $s_0 = \langle \emptyset, \emptyset, \emptyset, \emptyset\rangle$: no requests have been sent yet, no one has voted for and no one has voted against the proposition.

Labelled transition systems suit nicely for giving a theoretical framework for automatic conformance testing. However, as expected, using LTSs for giving specifications of Internet applications is not convenient. To make this framework useful in practice, we need a formalism for easier specifying these applications. Therefore, we are currently developing *DiCons* [15, 16], which is a formal specification language dedicated to the domain of Internet applications. We will not give the actual *DiCons* specification here since this goes beyond the scope of this paper. However, to give an example, the Internet vote described above can be specified in *DiCons* in five lines of code.

As a proof of concept, we implemented an on-the-fly version of Algorithm 1. We used this algorithm to test eleven implementations of the Internet vote application: one correct and ten incorrect implementations. We tested by executing 26.000 test cases per implementation. This took approximately half a day per implementation. We tested using different lengths of test traces and different numbers of voters. The test results are summarised in Table 1. The left column describes the error in the implementation. In the second column, the percentage of test cases that ended in a fail state is given.

As can be seen, in nine out of ten incorrect implementations, errors are detected. In all test cases, only requests are sent that are part of the specification, i.e., only requests for votes by known voters are sent. Because we did not specify that unknown voters are forbidden to vote, errors in the implementation that allow other persons to vote are not detected: the implementation conforms to the specification.

The percentages in Table 1 strongly depend on the numbers of voters and lengths of the test traces. Some errors can easily be detected by examining the scores, e.g. incorrect initialisation. This error can be detected by traces of length 2: request for the score and inspect the corresponding response. Other errors, however, depend on the number of voters. If the last vote is counted twice, all voters have to vote first, after which the scores have to be inspected. This error can only be detected by executing test traces with at least a length of two times the number of voters plus two.

Table 1. Test results

implementation	% failures	verdict
1. correct implementation	0.00	**pass**
2. no synchr.: first calculate results, then remove voter	33.30	**fail**
3. no synchr.: first remove voter, then calculate results	32.12	**fail**
4. votes are incorrectly initialised	91.09	**fail**
5. votes for and against are mixed up	87.45	**fail**
6. votes by voter 0 are not counted	32.94	**fail**
7. voter 0 cannot vote	91.81	**fail**
8. unknown voter can vote	0.00	**pass**
9. voters can vote more than once	68.75	**fail**
10. voter 0 is allowed to vote twice	16.07	**fail**
11. last vote is counted twice	8.82	**fail**

6 Related Work

Automatic test derivation and execution based on a formal model has been an active topic of research for more than a decade. This research led to the development of a number of general purpose black box test engines. However, the domain of Internet applications implies some extra structure on the interacting behaviour of the implementation which enforces the adaptation of some of the key definitions involved. Therefore, our work can be seen as an extension to and adaptation of the formal testing framework as introduced in [7–9]. The major difference stems from our choice to model an Internet application as a multi request-response transition system. We expect that existing tools (such as TorX [6]) can be easily adapted to this new setting. The reader may want to consult [3] for an overview of other formal approaches and testing techniques.

Approaching the problem of testing Internet applications from another angle, one encounters methodologies and tools based on capture/replay (see e.g. [17, 18]). In the case of capture/replay testing, test cases are produced manually and recorded once, after which they can be applied to (various) implementations. These tools prove very beneficial for instance for regression testing. However, automatic generation of test cases has several advantages. In general it proves to be a more flexible approach, yielding test suites that are better maintainable and more complete and test suites can be generated quicker (and thus cheaper). The main disadvantage of automatic black box testing is that it requires a formal model of the implementation under test.

A methodology that comes very close to ours is developed by Ricca and Tonella [19]. The starting point of their semi-automatic test strategy is a UML specification of a web application. This specification is manually crafted, possibly supported by re-engineering tools that help in modelling existing applications. Phrased in our terms, Ricca and Tonella consider RRTSs as their input format (which they call *path expressions*). We perform black-box testing, whereas they consider white-box testing. This implies that their approach considers implementation details (such as cookies), while we only look at the observable behaviour. White-box testing implies a focus on test criteria, instead of a complete testing algorithm. Finally, we mention the difference in user involvement. In our approach the user has two tasks, viz. building an abstract specification and instantiating the test adapter which relates abstract test events to concrete HTTP-events. In their approach the user makes a UML model, produces tests and interprets the output of the implementation. For all of this, appropriate tool support is developed, but the process is not automatic. In this way derivation and execution of a test suite consisting of a few dozens of tests takes a full day, whereas our on-the-fly approach supports many thousands of test cases being generated, executed and interpreted in less time.

Jia and Liu [20] propose a testing methodology which resembles Ricca and Tonella's in many respects, so the differences with our work are roughly the same. Their focus is on the specification of test cases (by hand), while our approach consists of the generation of test cases from a specification of the intended application's behaviour. Their approach does not support on-the-fly test generation and execution. Like Ricca and Tonella, their model is equivalent to RRTSs which makes it impossible to test parallel sessions (or users) that share data.

Wu and Offutt [21] introduce a model for describing the behaviour of Web applications, which can be compared to the *DiCons* language. In contrast to the model presented in this paper, their model supports the usage of special buttons that are available in most Web browsers. The main difference with our model is that they focus on stateless applications, i.e., responses only depend on the preceding request. We model stateful applications which are based on parallelly executed sessions.

Another functional testing methodology is presented by Niese, Margaria and Steffen in [22]. Where we focus on modelling Internet applications only, they model other subsystems in the system under test as well. In their approach, test cases are not generated automatically, but designed by hand using dedicated tools. Test execution takes place automatically via a set of cooperating subsystem-specific test tools, controlled by a so-called test coordinator.

Our research focuses on conformance testing only. Many other properties are important for the correct functioning of web applications, such as performance, user interaction and link correctness [23]. Testing such properties is essentially different from conformance testing. They focus on *how well* applications behave instead of *what* they do. Plenty of tools are available for performance testing, e.g., [24, 25].

7 Conclusion

The research reported on in this paper is conducted in the context of the *DiCons* project (see [15, 16]). The goal of this project is the application of formal methods (especially process algebra) to the application of dependable Internet applications. One of the results of this project is the development of the *DiCons* language, which is targeted to the specification of the interaction behaviour of

Internet applications. The *DiCons* compiler allows for the generation of stand-alone Internet applications.

Due to the focus of *DiCons* on interaction, rather than on presentation, it is likely that developers will prefer to use a less formal approach that supports the need for a nice user interface. However, our current research shows that development of a formal interaction model, like in *DiCons*, still has benefits. Our research shows that there is a point in making a formal model, even if it is not used to generate Internet applications, since a formal model can be used for (automated) conformance testing of the application.

The input of the testing process described in this paper is a multi request-response transition system which is a theoretically simple model, but which is very hard to use in practice for the specification of real applications. Since *DiCons* is targeted to specify Internet applications and since its operational semantics is an MRRTS, we plan to connect the *DiCons* execution engine to our prototype testing tool.

As the development of a formal model of an Internet application is quite an investment, we expect that only in cases where it is vital that the application shows the correct interaction behaviour automated formal testing will be applied. However, there will be a huge gain in reliability and maintainability of the application (e.g. because of automated regression testing), compared with e.g. capture and replay techniques.

Although we have only built a simple prototype, we can already conclude that the proposed testing approach works in practice, since it quickly revealed (planted) errors in erroneous implementations. Interestingly enough, playing with the prototype made it clear that the response times in the HTTP-protocol are much slower than in traditional window based applications, resulting in less test runs per time unit. We cannot foresee if the unreliability of an Internet connection will prevent us from executing lengthy test runs over the Internet.

An interesting point is that the actual HTTP-response of an Internet application has to be matched against the expected abstract event from the specification. In our current prototype tool we simply scan for the occurrence of certain strings, but this does not seem to be a safe and generic approach. Future research should answer the question of how to match actual HTTP-replies against abstract events.

References

1. Regan, K.: U.S.: E-Commerce Topped $45B in 2002. E-Commerce Times (2003)
2. ACM Committee on Computers and Public Policy, P.G. Neumann, moderator: The Risk Digest, Forum On Risks To The Public In Computers And Related Systems. `http://catless.ncl.ac.uk/Risks/` (1985–2003)
3. Brinksma, E., Tretmans, J.: Testing Transition Systems: An Annotated Bibliography. In Cassez, F., Jard, C., Rozoy, B., Ryan, M., eds.: Summer School MOVEP'2k – Modelling and Verification of Parallel Processes, Nantes (2000) 44–50
4. Fielding, R., Gettys, J., Mogul, J.C., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol – HTTP/1.1. RFC 2616, The Internet Society, Network Working Group (1999)

5. Rice, J., Farquhar, A., Piernot, P., Gruber, T.: Using the web instead of a window system. In: Human Factors in Computing Systems, CHI'96 Conference Proceedings, Vancouver, B.C, Canada (1996) 103–110
6. Belinfante, A., Feenstra, J., de Vries, R., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal test automation: A simple experiment. In: 12^{th} Int. Workshop on Testing of Communicating Systems, Kluwer Academic Publishers (1999)
7. Brinksma, E., Alderden, R., Langerak, J., van de Lagemaat, R., Tretmans, J.: A formal approach to conformance testing. In: Second International Workshop on Protocol Test Systems, North-Holland (1990) 349–363
8. Tretmans, J.: A formal approach to conformance testing. In Rafiq, O., ed.: International Workshop on Protocol Test Systems VI. Volume C-19 of IFIP Transactions., North-Holland (1994) 257–276
9. ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8: Proposed ITU-T Z.500 and committee draft on "formal methods in conformance testing". CD 13245-1, ISO – ITU-T, Geneva (1996)
10. Bernot, G.: Testing against formal specifications: A theoretical view. In Abramsky, S., Maibaum, T.S.E., eds.: TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development. Volume 494 of Lecture Notes in Computer Science., Springer-Verlag (1991) 99–119
11. Tretmans, J.: Testing labelled transition systems with inputs and outputs. In Cavalli, A., Budkowski, S., eds.: Participants Proceedings of the Int. Workshop on Protocol Test Systems VIII – COST 247 Session, Evry, France (1995) 461–476
12. Heerink, L.: Ins and outs in refusal testing. PhD thesis, University of Twente, The Netherlands (1998)
13. Brinksma, E., Scollo, G., Steenbergen, C.: LOTOS specifications, their implementations and their tests. Protocol Specification, Testing and Verification VI, IFIP 1987 (1987) 349–360
14. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software—Concepts and Tools **17** (1996) 103–120
15. van Beek, H.: Internet protocols for distributed consensus – the DiCons language. Master's thesis, Technische Universiteit Eindhoven (2000)
16. Baeten, J., van Beek, H., Mauw, S.: Specifying internet applications with DiCons. In: Proc. 16th ACM Symposium on Applied Computing, Las Vegas, USA (2001)
17. CollabNet, Inc.: MaxQ. `http://maxq.tigris.org/` (1999–2003)
18. The Original Software Group Ltd.: TestWEB. `http://www.testweb.com/` (2003)
19. Ricca, F., Tonella, P.: Analysis and testing of web applications. In: Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01), Toronto, Ontario, Canada, IEEE Computer Society (2001) 25–34
20. Jia, X., Liu, H.: Rigorous and automatic testing of web applications. In: Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002), Cambridge, MA, USA (2002) 280–285
21. Wu, Y., Offutt, J.: Modeling and testing web-based applications. ISE Technical ISE-TR-02-08, GMU (2002)
22. Niese, O., Margaria, T., Steffen, B.: Automated functional testing of web-based applications. In: Proceedings of the 5th Int. Conference On Software and Internet Quality Week Europe (QWE2002), Brussels, Belgium (2002)
23. Benedikt, M., Freire, J., Godefroid, P.: VeriWeb: Automatically testing dynamic web sites. In: Proceedings of the 11th international world wide web conference (WWW2002), Honolulu, Hawaii, USA (2002)
24. Dieselpoint, Inc.: dieseltest. `http://www.dieseltest.com/` (2001)
25. Fulmer, J.: Siege. `http://www.joedog.org/siege/` (2002)