Model-based Testing of the Conference Protocol with Spec Explorer

Matko Botinčan*, Vedran Novaković*
*University of Zagreb, Department of Mathematics, Zagreb, Croatia
{mabotinc,venovako}@math.hr

***Abstract*— This paper presents a case study of model-based testing of the Conference Protocol, a simple multicast chat box protocol reported in [2], [4], [8], [11], [13], with the model-based testing tool Spec Explorer [14]. Our approach differs from previous case studies of the Conference Protocol primary in the choice of the employed modeling language and the tool for model-based testing. The set of results provided by this paper aims to be comparable with existing results on testing the Conference Protocol with other model-based testing tools. Additionally, the presented data by itself also gives a novel performance measurement of the Spec Explorer tool.**

I. Introduction

Testing refers to activity of executing an implementation of a software product or its components in order to expose defects. It is often one of the costliest aspects of the whole software development process. Despite its importance, it is still not as developed as other engineering disciplines, and lacks adequate tool support.

The model-based testing is a kind of automated black-box testing in which the software specification is used as an oracle for validating correctness of the implementation. It is one of the most prominent approaches for dealing with shortcomings of the usual testing methodology.

In model-based testing, tests are generated systematically from a model of the implementation. Here, a model is a (not necessarily comprehensive) abstraction of the actual system from a particular perspective. Models are typically described by a modeling language — these high-level programs describe the operational behavior of the implementation at a chosen level of abstraction.

Spec Explorer [14] is a model-based state exploration and testing tool for .NET platform developed by the Foundations of Software Engineering group in Microsoft Research. It uses Spec# [1] as its primary language for describing models of the system. Several product groups in Microsoft use it for testing operating system components and .NET framework components on a daily basis.

The Conference Protocol [6] is a simple multicast chat box protocol which we found interesting and challenging enough to use as a case study for experimental evaluation of the Spec Explorer model-based testing tool. Our primary goal is to provide a set of results on testing the Conference Protocol with Spec Explorer that would be similar to those on testing the Conference Protocol with other model-based testing tools.

A. Related work

The Conference Protocol has been used as a case study for experimental evaluation of TorX [15] — one of the most influential model-based testing tools developed in academic community so far. The test scenarios in [2] are derived from multiple specification languages (namely, LOTOS, Promela and SDL) and then on-the-fly and batch tested with TorX. [4] investigates combinations of TGV for abstract test generation and TorX for test execution. [11] brings the case study of the Conference protocol with Phact (Philips Automated Conformance Tester).

B. Our contributions

The main contributions of this paper are as follows: a rewrite of the informal specification of the Conference Protocol in more concise and formal terms than its original specification [8]; a Spec# model of one aspect of the Conference Protocol (namely, the one seen by the Conference protocol entities); and experimental results on model-based testing of our implementation of the Conference Protocol against the given Spec# model by using the Spec Explorer model-based testing tool.

The rest of the paper is organized as follows. In Section II, we give an overview of concepts and underlying theoretical foundations of model-based testing with Spec Explorer. Informal description and a partial Spec# model of the Conference Protocol is presented in Section III. Section IV reports experimental results of performed model-based testing activities. Finally, in Section V we give concluding remarks.

II. Model-based testing with Spec Explorer

This section provides an overview of the Spec Explorer model-based testing tool and its underlying theoretical foundations. We start by introducing interface automata and their syntactic representation in Spec Explorer in form of model programs. The notion of interface automata turns out to be important for precise formulation of the conformance relation between the model and the implementation by means of the alternating simulation. Moreover, it also serves to describe the test generation in a unified manner, including both the offline and the online testing.

A. Interface automata

One way to describe the behavior of reactive systems (i.e., systems reacting with their environment) is by using the notion of interface automata [7]. In this setting, both participants of the model-based testing, the model and the implementation under test (IUT), are viewed as interface automata, and the relationship between them is described by the conformance relation.

The Spec Explorer tool adopted the formalism based on interface automata [16], and slightly extended it [5] by giving model programs a more rigorous and concise semantics in terms of abstract state machines [9]. However, due to page limitation of this paper, we mostly rely on the exposition [16] based on interface automata, as it requires less preparatory material, while still being sufficient for the scope of this paper.

Interface automaton can be defined as an ordered tuple $\mathcal{M} = (S, S^{init}, A^c, A^o, \Gamma^c, \Gamma^o, \delta)$ where:

- S is a set of states;
- $S^{init} \subseteq S$ is a set of initial states;
- A^c and A^o are mutually disjoint sets of controllable actions and observable actions, respectively; the set $A^c \cup A^o$ of all actions is denoted by A;
- $\Gamma^c \colon S \to \mathscr{P}(A^c)$ and $\Gamma^o \colon S \to \mathscr{P}(A^o)$ are enabling functions for controllable and observable actions; in addition, $\Gamma(s)$ denotes the set $\Gamma^c(s) \cup \Gamma^o(s)$ of all enabled actions in a state s;
- $\delta \colon S \times A \rightharpoonup S$ is a transition function mapping a state s and an action a enabled in the state s ($a \in \Gamma(s)$) to a target state $\delta(s, a)$.

A state s for which $\Gamma(s) = \emptyset$ is called terminal. A nonterminal state s with $\Gamma^o(s) = \emptyset$ is called active, otherwise we call it passive.

The partition of the set of actions A into controllable and observable ones serves to distinguish the behavior that can be controlled from the behavior that can only be observed. In the context of model-based testing, the controllable actions are typically triggered by the model describing operational behavior of the IUT, while the observable actions are those which are to be observed about the IUT.

B. Model programs

In the Spec Explorer tool, models of reactive systems are not specified directly as interface automata, but via model programs written in high level specification language AsmL [10] or Spec# [1]. While AsmL is essentially based on abstract state machines, and thus requires a basic understanding of their underlying theoretical concepts for its proper use, Spec# is a design-by-contract extension of the mainstream object-oriented imperative language C#, targeting a possibly larger number of users, that are acquainted to the “usual” semantics. We use Spec# as a specification language in this paper.

Every model program P consists of a (finite) set Acts of action methods and a (finite) set Vars of state variables. A state of P is determined by values of all members in Vars. Note that depending on the types of state variables, a model program may have infinitely many states. The values of variables in Vars evolve during P’s execution in a way prescribed by action methods. Here each action method m (having a sequence of variables $\mathbf{x}$ as its input parameters) equals to ordinary C# method augmented with a state dependent boolean predicate $Pre_m(\mathbf{x})$ representing the precondition of m.

Given a model program P, it defines an interface automaton $\mathcal{M}$ as follows. The set S^{init} of initial states of $\mathcal{M}$ contains only the initial state of P, i.e., the set of initial assignment of Vars members. The set S then equals to the set of all states reachable from S^{init} by the transition function δ described further.

An action in $\mathcal{M}$ corresponds to an invocation of an action method in Acts with some actual parameters. Let us denote by $m(\mathbf{v})/\mathbf{w}$ an invocation of the action method m on the input parameters $\mathbf{v}$, combined with the output parameters $\mathbf{w}$. An action $a = m(\mathbf{v})/\mathbf{w}$ is enabled in a state s, i.e., $a \in \Gamma_m(s)$, if $Pre_m(\mathbf{v})$ is true in s and the invocation of $m(\mathbf{v})$ in s yields the output parameters $\mathbf{w}$. Given a source state s and an action $a = m(\mathbf{v})/\mathbf{w} \in \Gamma_m(s)$, the transition function δ maps (s, a) to the target state of the invocation $m(\mathbf{v})$. Note that the set $\Gamma(s)$ of all enabled actions in a state s equals $\bigcup_{m \in \mathsf{Acts}} \Gamma_m(s)$. The set A of all actions is then taken to be $\bigcup_{s \in S} \Gamma(s)$.

C. Test generation

Still, a model program P serves as an operational contract for the IUT without regard for any particular purpose. In many cases the corresponding interface automaton $\mathcal{M}$ has too large or even infinite number of states or actions. In order to generate an interface automaton with sets of states and actions of manageable size, various techniques for scenario control have to be employed. Their purpose is to generate from P a restricted form of $\mathcal{M}$, suited for the test goal that wants to be achieved.

The Spec Explorer tool incorporates the following techniques for scenario control: parameter selection (specifies the set of values for parameters of action methods in Acts), method restriction (selects a subset of possible actions in A satisfying a given state-based expression), state filtering (prunes away all states from S that fail to satisfy a given state-based condition), directed search (allows the user to limit and direct the (otherwise nondeterministic) traversal of the state space S by defining the probability space of the random variables used for selection; states and transitions not visited are pruned away), and state grouping (selects representative states from equivalence classes defined by state-based expressions).

The process of model-based testing comprises test generation and test execution. In case when from a given model program tests are generated in advance with an aim to achieve some predefined test goal we speak of offline testing. Here the test execution comes only afterwards, taking pregenerated test suites and running them against the IUT in order to find discrepancies between the IUT behavior and the behavior predicted by the model. However, it is also possible to combine the test generation and the test execution into a single process where tests are generated on-the-fly as the testing advances. This mode of model-based testing is called online testing.

In both cases, a test suite (or shortly, a test) can be seen as an interface automaton, say $\mathcal{T}$, produced by traversing the interface automaton $\mathcal{M}$ in some way. $\mathcal{T}$

is pregenerated in the offline case, while in the online case it is unfolded dynamically. In general, $\mathcal{T}$ may have additional state variables (e.g. to represent information about traversals of $\mathcal{M}$) and the set of actions is equipped with a new controllable action $Observe$ (representing the choice of waiting for an observable action) and a new observable action $Timeout$ (representing that no (other than $Timeout$) observable actions have happened). Each state of $\mathcal{T}$ inherits all properties of the corresponding state of $\mathcal{M}$, and each transition in $\mathcal{T}$ (other than $Observe$ and $Timeout$) corresponds to a transition in $\mathcal{M}$.

In offline testing, a traversal algorithm produces $\mathcal{T}$ from $\mathcal{M}$ in advance aiming for a particular test goal — to reach a state satisfying a given condition, to provide a full coverage of $\mathcal{M}$'s state space or to generate random walks satisfying some probability distribution on the set of actions. Depending on the presence of observable actions in $\mathcal{M}$ and whether $\mathcal{M}$ is deterministic (i.e., the set of enabled actions in each state of $\mathcal{M}$ is a singleton set) or nondeterministic, different traversal algorithms have to be employed. In the general case, Spec Explorer incorporates a variant of Dijkstra's shortest path algorithm for alternating reachability and a nondeterministic extensions of the Chinese postman tour algorithm [12]. For generating tests that optimize the expected cost with respect to action probabilities, adapted algorithms from Markov decision process theory are employed [3].

Instead of using pregenerated tests, in online testing tests are generated on-the-fly as the testing proceeds. This technique helps to resolve the nondeterministic behavior (typically arising in testing reactive systems) by avoiding large pregenerated tests $\mathcal{T}$ from $\mathcal{M}$ dealing with all possible responses of the IUT, and taking only a user-controlled stochastic sample of the state space. The algorithm implemented in Spec Explorer [5], [16] generates $\mathcal{T}$ by unfolding $\mathcal{M}$ nondeterministically. The choices of observable actions correspond to the observable actions taking place during the test execution, while the controllable actions are selected randomly with respect to the user-defined action weights.

D. Test execution

The purpose of the test execution is to determine whether the IUT meets the specification prescribed by the model the test has been generated from. This is achieved by checking the conformance relation between the model and the IUT, where the IUT is seen as a restricted form of the model. It is noteworthy to mention that in this context, the IUT is typically a "wrapper" of the actual system under test (which often happens to be multi-threaded, even distributed), or perhaps a particular aspect of it.

The behavior of the IUT is formalized through an interface automaton, say $\mathcal{N}$, obtained in the same way as for model programs and having the set of actions extended with the test actions $Observe$ and $Timeout$. Here the $Observe$ action is the only one leading to states of $\mathcal{N}$ where observable actions are enabled, while the $Timeout$ action is triggered after the amount of time determined to wait for occurrence of an observable action runs out.

The conformance relation is based on the notion of alternating simulation [5], [16]. Let $\mathcal{M} = (S_\mathcal{M}, S_\mathcal{M}^{init}, A_\mathcal{M}^c, A_\mathcal{M}^o, \Gamma_\mathcal{M}^c, \Gamma_\mathcal{M}^o, \delta_\mathcal{M})$ be the interface automaton generated from the model program, and $\mathcal{N} = (S_\mathcal{N}, S_\mathcal{N}^{init}, A_\mathcal{N}^c, A_\mathcal{N}^o, \Gamma_\mathcal{N}^c, \Gamma_\mathcal{N}^o, \delta_\mathcal{N})$ the interface automaton of the IUT. An alternating simulation from $\mathcal{M}$ to $\mathcal{N}$ is a binary relation $\succcurlyeq \subseteq S_\mathcal{M} \times S_\mathcal{N}$ such that for all $s \in S_\mathcal{M}$ and $t \in S_\mathcal{N}$ with $s \succcurlyeq t$, the following conditions are satisfied:

– $\Gamma_\mathcal{M}^c(s) \subseteq \Gamma_\mathcal{N}^c(t)$ and $\Gamma_\mathcal{N}^o \subseteq \Gamma_\mathcal{M}^o$;
– For all actions $a \in \Gamma_\mathcal{M}^o(s) \cup \Gamma_\mathcal{N}^c(t)$, it holds $\delta_\mathcal{M}(s,a) \succcurlyeq \delta_\mathcal{N}(t,a)$.

The intuition behind the two conditions is as follows. The first condition expresses the controllable-observable duality between states in the alternating simulation: every controllable action enabled in the model is also enabled in the IUT, and conversely, every possible response from the IUT is also enabled in the model. The second condition ensures that every step from s that corresponds to a controllable action enabled in the model or an observable action enabled in the implementation can be matched by a step from t.

A refinement from interface automaton $\mathcal{M}$ to interface automaton $\mathcal{N}$ is an alternating simulation $\succcurlyeq \subseteq S_\mathcal{M} \times S_\mathcal{N}$ from $\mathcal{M}$ to $\mathcal{N}$ such that $S_\mathcal{M}^{init} \times S_\mathcal{N}^{init} \subseteq \succcurlyeq$. We say that $\mathcal{M}$ specifies $\mathcal{N}$, or $\mathcal{N}$ conforms to $\mathcal{M}$, if there exists a refinement from $\mathcal{M}$ to $\mathcal{N}$.

Let us now describe how the Spec Explorer tool performs checking of the conformance relation. Spec Explorer stipulates that action methods from the model program are bound to methods with matching signatures in the IUT. Therefore, assuming equality of the signatures, the corresponding actions in interface automata $\mathcal{M}$ and $\mathcal{N}$ differ only in their output values (when enabled in both $\mathcal{M}$ and $\mathcal{N}$).

Given a state $s \in S_\mathcal{M}$, a controllable action $a = m(\mathbf{v})/\mathbf{w}_\mathcal{M}$ is chosen in $\mathcal{M}$ such that for the generated input parameters $\mathbf{v}$, the precondition $Pre_m(\mathbf{v})$ holds in s. After the method call $m(\mathbf{v})$ has been executed both in the model and the IUT, and produced output parameters $\mathbf{w}_\mathcal{M}$ and $\mathbf{w}_\mathcal{N}$, respectively, the values in $\mathbf{w}_\mathcal{M}$ and $\mathbf{w}_\mathcal{N}$ are compared for equality. In case when $\mathbf{w}_\mathcal{M} \neq \mathbf{w}_\mathcal{N}$, the action a is enabled in the model but not in the IUT, which results in a conformance failure.

On the other hand, an observable action happens impromptu on the IUT side. In order to track execution of observable actions, Spec Explorer instruments the IUT at the binary level, and the instrumented IUT signals the Spec Explorer's conformance engine occurrences of observable actions. Upon occurrence of an observable action $a = m(\mathbf{v})/\mathbf{w}_\mathcal{N}$, Spec Explorer acts as follows. First, the precondition $Pre_m(\mathbf{v})$ is checked in the model program, and if it does not hold, then a is not enabled in $\mathcal{M}$ and a precondition conformance failure occurs. Otherwise, the method call $m(\mathbf{v})$ is executed in the model program, yielding either a conformance failure (if invariant or postcondition does not hold), or output parameters $\mathbf{w}_\mathcal{M}$. An unexpected return value conformance failure is generated if $\mathbf{w}_\mathcal{M} \neq \mathbf{w}_\mathcal{N}$.

III. The Conference Protocol

A. Informal description

The Conference Protocol is a simple distributed protocol that resembles a multicast chat box. At the highest level of abstraction, it can be described in the terms of *conferences*, *partners*, and *messages*. A *message* is just a (possibly empty) sequence of no more than 255 ASCII characters that is communicated among all partners participating in a given conference. A *potential partner* is an entity (a physical user, i.e., software component acting on its behalf) that can opt to participate in an existing conference or to create a new one. There exists a fixed set (a universum U) of all potential partners. A member of U participating in some conference will be referred to as a *partner* of all other members taking part in the same conference. A *conference* is a named dynamic non-empty set of partners, i.e., each conference is a subset of U. Any potential partner u can join any existing conference (provided that she knows the conference's name), create a new one, or leave the one she is currently participating in at any time. As u can participate in no more than one conference simultaneously, all conferences are pairwise disjoint at any given time, and their union is a subset of U.

Conferences and partners have names not longer than 10 ASCII characters. When a conference ceases to exist, its name can be reused to create a new one. Apart from that, conferences names are unique. Names of the partners in an arbitrary conference need not be unique, even more, any partner can change its name when joining another conference (but not while residing inside a conference).

When a partner sends a message in the conference, it is distributed to other partners. The Conference Protocol is not reliable in the sense that it does not guarantee the messages being delivered nor the relative order of them being preserved, since its primary goal is to mimic multicasting over a connectionless network service. However, messages that do arrive are unchanged and not misdelivered.

The Conference Protocol specification is refined by providing interfaces of and defining interactions among the various protocol entities.

The conference service is accessed by a partner using Conference Service Access Point (CSAP), which serves as an interface for the adjoined Conference Protocol Entity (CPE). Each CSAP exposes the following set of control and data primitives:

- $\mathsf{join}(PartnerName, ConferenceName)$
- $\mathsf{leave}()$
- $\mathsf{datareq}(Message)$
- ○ $\mathsf{dataind}(PartnerName, Message)$

Control primitives are join and leave, because they establish membership of the partner to a given conference (providing partner's name), or terminate the active membership, respectively. The rest are data primitives, of which datareq sends a message to other partners, while dataind represents an event triggered upon the arrival of a message from a partner. That is, messages are received asynchronously.

All invocations of a given primitive on a single CSAP form a sequence. Disregarding their parameters, let them be noted as $\mathsf{primitive}_{index \in \mathbb{N}}$. Let $\prec$ denote an irreflexive and transitive relation, such that $\mathsf{p1}_i \prec \mathsf{p2}_j$ iff the i-th invocation of a primitive p1 happens strictly before the j-th invocation of a primitive p2. Every interaction with a single CSAP is therefore a partially ordered set $(I, \prec)$ of made invocations (for some $\prec$) which – if non-empty – must satisfy the following conditions:

$$(\forall i)(\forall j \neq i)(\exists k)(\mathsf{join}_i \prec \mathsf{leave}_k \prec \mathsf{join}_j),$$
$$(\forall \mathsf{p} \in \{\mathsf{leave}, \mathsf{datareq}, \mathsf{dataind}\})(\forall i)(\exists j)(\nexists k)$$
$$((\mathsf{join}_j \prec \mathsf{p}_i) \wedge (\mathsf{join}_j \prec \mathsf{leave}_k \prec \mathsf{p}_i)).$$

It follows that every such non-empty partial order of invocations has join_0 for the unique $\prec$-minimal element.

The universum U of all potential partners is available to every CPE as a sequence of pairs $(UID, IPendpoint)$, where $UID \in \mathbb{N}^+$ and $IPendpoint = (Host, Port)$, for an IP address of $Host$ and its UDP $Port \geq 1024$. UID is the unique identifer of a partner (i.e., her CPE), and all $IPendpoint$s, as network addresses of partners, are also unique in U. The CPE partitions U into two disjoint sets AU and IU of potential partners – those who are also active partners and those who are not, respectively. At any time the basic invariant $U = AU \uplus IU$ must hold, and if a CPE is in the idle state (i.e., if it is not participating), it must ensure that $AU = \emptyset$. A CPE also has to maintain a mapping $Names$ of partners' addresses to their names.

CPEs generate, send and receive data chunks known as the Protocol Data Units (PDUs), that are binary representations of primitive actions:

- JoinPDU informs a potential partner that the sender is joining the named conference.
- LeavePDU informs a partner that the sender is leaving the conference.
- AnswerPDU informs the recipient that the sender is participating in the same conference.
- DataPDU conveys a message to another partner.

Every PDU starts with an octet designating the PDU type. The names of partners and conferences are encoded in 10 octets, without a terminator and left-padded with zeroes:

PDU	1 octet	10 octets	10 octets
Join	`01`	*PartnerName*	*ConferenceName*
Leave	`02`	*PartnerName*	*ConferenceName*
Answer	`03`	*PartnerName*	*ConferenceName*

The messages are represented as Pascal-style strings:

PDU	1 octet	1 octet	$\leq$ 255 octets
Data	`04`	*length*	*data*

PDUs are exchanged between CPEs over network and can fit into a single UDP datagram. Communication facility of a CPE is called Multicasting Protocol Entity (MCPE), and its interface to the correspondent CPE is Multicasting Service Access Point MCSAP, which offers the following primitives:

- send$(PDU, User)$
- multicast(PDU)
- broadcast(PDU)
- ◦ receive$(PDU, User)$

$User$ can be either a partner or a potential partner, as we shall see later. The first three primitives can be synchronous (though in our implementation they are asynchronous), and the fourth is inherently asynchronous. Multicasting differs from broadcasting in the intended set of recepients. Multicast primitive sends a PDU to all members of the associated CPE's AU set (i.e., to all partners), in an unspecified manner (it is implemented as an iteration through AU that calls the send primitive). Broadcast primitive does the same for all potential partners (universum U). Sending and receiving primitives intentionally resemble *sendto* and *recvfrom* POSIX operations on UDP sockets, respectively.

Each CPE maintains its internal state, which can be classified as either *idle* or *active*. The classification is based on the current conference's name. If it is undefined (**null**), the state is idle, and it is active otherwise. Transitions between the two classes of states are governed by the control primitives invoked. In an idle state there is no conference a CPE actively participates in, therefore, its set of partners AU is empty. A CPE becomes active by join primitive only, and idle again by executing leave, as depicted in Fig. 1. At the idle state, all incoming PDUs are disregarded and there are no outgoing PDUs as well. All operations in an active state are synchronized with respect to the entire state. That is, a state can be updated by at most one operation simultaneously. The state machine for a CPE, focusing on incoming PDUs and allowed primitives, is summarized briefly in Table I. Ramifications are ommited for brevity.

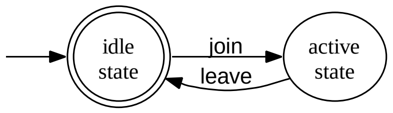

Fig. 1. A simple automaton view of state transitions

TABLE I
THE STATE MACHINE FOR A CPE

PDU	*Idle*	*Active*
Answer	ignore	*updatePartners*
Join	ignore	*updatePartners* ‖ *answer*
Leave	ignore	*updatePartners*
Data	ignore	*processDataInd*

primitive	*Idle*	*Active*
join	*doJoin*	not allowed
leave	not allowed	*doLeave*
datareq	not allowed	*doDataReq*

Symbol ‖ should be read as "do both, maybe in parallel". By $Sender \in U$ we denote a (potential) partner that sends a PDU to the given CPE, and by the symbol $\triangle$, the symmetric difference of two sets.

```
updatePartners:
  if PDU.$ConferenceName$ == CPE.$ConferenceName$
    then CPE.$AU$ := CPE.$AU \triangle \{Sender\}$
answer:
  MCSAP.send(AnswerPDU, $Sender$)
processDataInd:
  if $Sender \notin$ CPE.$AU$
    then MCSAP.send(JoinPDU, $Sender$)
  CSAP.dataind(CPE.$Names\,[Sender]$, PDU.$data$)
doJoin:
  set CPE.$ConferenceName$ (become Active)
  MCSAP.broadcast(JoinPDU)
doLeave:
  MCSAP.multicast(LeavePDU)
  CPE.$AU$ := $\emptyset$
  unset CPE.$ConferenceName$ (become Idle)
doDataReq:
  MCSAP.multicast(DataPDU)
```

B. The Spec# model

Our Spec# model focuses on the CPE, emphasizing the transitions of its internal state and disregarding communication or user interface parts of an implementation. We first study types of the state variables, concerning the ability to restrict their ranges – providing in that way an upper limit on the cardinality of the set of states of the model program:

```
type Name = string where
  (value == null) || (value.Length <= 10);
type Message = string! where
  value.Length <= 255;
type Nick = Name;
type ConfID = Name;
type UID = int where value > 0;
type PeerList = Map<UID,Nick>;
```

Type *Name* represents the allowed names of conferences and partners, and is constrained to null-references and strings of no more than 10 characters. *Message* is a non-null type restriction of strings with their length not exceeding 255 characters, *UID* is a type of positive integers, and *PeerList* is a mapping between unique identifiers of partners and their (possibly undefined) names.

Each state is a valuation of the following variables:

```
var ConfID conference = null;
readonly UID selfUID = 1;
readonly Name selfNick = "ConfProt";
var PeerList partners = Map{};
```

Variable "conference" is undefined in the initial state, and a mapping "partners" (which serves both as AU and $Names$ constituents of a CPE observed) is initially empty. Values for "selfUID" and "selfNick" are arbitrarily chosen, but constant in every state.

Controllable action methods of the model program are just those which correspond to the CSAP primitives. They are attributed as "Action" and given preconditions (formulæ in the **requires** clauses) and postconditions (formulæ in the **ensures** clauses). Notice how preconditions and postconditions depend on the current state and the values of the actual parameters.

```
[Action]
void Join(ConfID newConference)
  requires conference == null;
  requires newConference != null;
  requires partners.IsEmpty;
  ensures conference == newConference;
{ conference = newConference; }

[Action]
void Leave()
  requires conference != null;
  ensures conference == null;
  ensures partners.IsEmpty;
{
  conference = null;
  partners = Map;
}

[Action]
void DataReq(Message msg)
  requires conference != null;
{}
```

Observable action methods are specifically attributed. They are triggered by an implementation wrapper upon the arrival of a corresponding PDU.

```
[Action(Kind=ActionAttributeKind.Observable)]
void OnJoin(UID uid, ConfID confID, Nick nick)
  requires uid != selfUID;
  requires confID != null;
  requires (nick != null)
    && (nick.CompareTo(selfNick) != 0);
{
  if (confID == conference)
    partners[uid] = nick;
}

[Action(Kind=ActionAttributeKind.Observable)]
void OnAnswer(UID uid, ConfID confID, Nick nick)
  requires uid != selfUID;
  requires confID != null;
  requires (nick != null)
    && (nick.CompareTo(selfNick) != 0);
{
  if (confID == conference)
    partners[uid] = nick;
}

[Action(Kind=ActionAttributeKind.Observable)]
void OnLeave(UID uid, ConfID confID, Nick nick)
  requires uid != selfUID;
  requires confID != null;
  requires (nick != null)
    && (nick.CompareTo(selfNick) != 0);
  ensures uid notin partners;
{ partners[uid] = none; }

[Action(Kind=ActionAttributeKind.Observable)]
void OnDataInd(UID uid, Message msg)
  requires uid != selfUID;
  requires msg != null;
{}
```

IV. Experimental Results

This section gives a partial report of the performed experimental activities regarding model-based testing of the Conference Protocol against the Spec # model given in the previous section by using the Spec Explorer model-based testing tool.

A. Test architecture

In order to make experimental results sensible, we first briefly describe the test architecture, i.e., the environment in which the testing process has been performed. The abstract view of the test architecture is shown in Fig. 2. The IUT in our case is the CPE. However, as the methods signatures in the IUT and in the Spec# model do not match, there is a "wrapper" around the IUT (i.e., a conformance stub in the terminology used by Spec Explorer) providing methods with matching signatures that can be bound to the methods in the Spec# model.

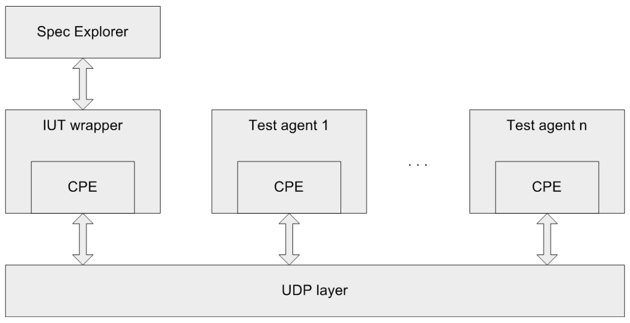

Fig. 2. The test architecture

The single CPE governed by Spec Explorer is run together with a number of other CPE's (governed by test agents simulating user actions) connected to the same conference. Test agents (assuming that there are n of them) are assigned names "TestBot1", …, "TestBot$n$" and have UIDs ranging from 2 to $n+1$ (the UID value 1 is assigned to the CPE governed by Spec Explorer). For simplicity and without much loss of generality we have chosen that test agents together with the IUT exchange messages having the same content.

Test agents simulate typical user actions; agent's cycle consists of joining a conference, exchanging few messages (if any), and leaving the conference. We distinguish two types of agents with respect to their periodic behavior — those which have pauses of predetermined duration between single steps of a cycle, and those for which the pauses were randomized. As the experimental results will show, the latter behavior produces more chaotic environment with larger number of states, and, therefore, more difficult to test.

B. Offline and online testing with a single test agent

It is instructive to compare models (i.e., interface automata) generated by performing the offline test generation (i.e., by running the transition coverage traversal algorithm) to the ones obtained by the online testing. In order to ensure models of a size small enough to make sense to visualize them, we present the case obtained by testing the IUT together with a single test agent.

Fig. 3 shows the model obtained by offline testing the IUT with a single test agent, while Fig. 4 shows the corresponding model obtained by online testing. The former model contains both active and passive states (denoted by circles and diamonds, respectively) as the model program contains observable action methods and the timeout is nonzero. However, as in online testing the timeout is neglected (or, equivalently, is a priori set to 0), the later model contains only passive states (i.e., all states in the model are denoted by diamonds). In a model having the structure like the one on Fig. 4, a controllable action is going to be executed by the conformance engine only after observing that no observable actions have occurred.

C. Detection of errors in the IUT

In order to quantitatively measure effectiveness of the offline and the online approach to model-based testing, we have made several mutant implementations of the CPE containing deliberate errors and counted the number of steps each testing procedure needed to make in order to detect a conformance error. Table II contains the data obtained for three mutant implementations run together with a single test agent (the data in the table are average values obtained through 100 testing runs; the obtained values were not dependant on whether the employed agent had predetermined or randomized pauses).

TABLE II
NUMBER OF STEPS NEEDED TO DETECT AN ERROR IN MUTANT IMPLEMENTATIONS OF THE CPE

	Offline testing	Online testing #
Mutant #1	5	50
Mutant #2	5	50
Mutant #3	3	10

D. Performance of the online testing with multiple test agents

The final experimental result we report in this paper concerns performance of the online testing when multiple test agents are present in the testing environment. Its goal is to show scalability of the online testing with respect to the number of present test agents by measuring the number of transitions and states revealed during the online testing process. It is easy to see that the overall number of states in the model when n test agents are present equals to $2^n + 1$. Therefore, the state coverage of the model is calculated as the ratio of the number of discovered states over $2^n + 1$.

Table III and Table IV summarize the obtained results for online testing of the IUT with respect to the number of test agents with pauses of predetermined duration between single steps, and with randomized pauses, respectively (the data in the table are average values obtained through 10 testing runs).

V. CONCLUSIONS

In this paper, we have presented a case study of model-based testing of the Conference Protocol by using the Spec Explorer model-based testing tool. Our first goal was to provide a clear and precise specification of the Conference Protocol in the natural language and rewrite one particular aspect of it in the Spec# modeling language. The given Spec# model turned out to be rich enough for performing meaningful model-based testing activities. This lead us to our second goal of providing a set of experimental results which would be comparable to existing results on testing the Conference Protocol in the literature (that were obtained by using other model-based testing tools).

As already noted, this paper does not provide a full Spec# model of the Conference Protocol, yet only the model of the CPE. Therefore, the primary continuation of the present work would be writing the full Spec# model of the Conference Protocol. Although we have already made certain progress in this direction, there exist subtle problems with implementing the new model within the current version of the Spec Explorer tool. We hope to address further discussion of this topic in a subsequent paper.

REFERENCES

[1] M. Barnett, R. Leino, and W. Schulte, “The Spec# programming system: An overview”. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, M. Huisman, Editor, vol. 3362 of LNCS. Springer, 2005, pp. 49–69.

[2] A. F. E. Belinfante, J. Feenstra, R. G. de Vries, G. J. Tretmans, N. Goga, L. M. G Feijs, S. Mauw and A. W. Heerink, “Formal Test Automation: A Simple Experiment.”, in *Proceedings of IFIP 13th International Workshop on Testing Communicating Systems: Method and Applications*. Kluwer Academic Publishers, 1999, pp. 179–196.

[3] A. Blass, Y. Gurevich, L. Nachmanson and M. Veanes, “Play to test”. Technical Report MSR-TR-2005-04, Microsoft Research, 2005.

[4] L. D. Bousquet, S. Ramangalahy, C. Viho, A. Belinfante and R. G. de Vries, “Formal Test Automation: The Conference Protocol with TGV/TORX”, in *Proceedings of IFIP 13th International Conference on Testing of Communicating Systems (TestCom 2000)*, H. Ural, R. L. Probert and G. von Bochmann, Eds. Kluwer Academic Publishers, 2000, pp. 221–228.

[5] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann and M. Veanes, “Model-based testing of object-oriented reactive systems with Spec Explorer”. Technical Report MSR-TR-2005-59, Microsoft Research, 2005.

[6] Conference Protocol Case Study website: `http://fmt.cs.utwente.nl/ConfCase/`.

[7] L. de Alfaro and T. A. Henzinger, “Interface automata”, in *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, 2001, pp. 109-120.

[8] C. de Barros Barbosa, L. F. Pires and M. van Sinderen. “Frameworks for protocol implementation”, in *Proceedings of the 16th Brazilian Symposium on Computer Networks (SBRC’98)*, Instituto de Computacao, Universidade Federal Fluminense, Rio de Janeiro, Brazil, 1998, pp. 385–403.

[9] Y. Gurevich, “Evolving Algebras 1993: Lipari Guide”. In *Specification and Validation Methods*, E. Börger, Editor. Oxford University Press, 1995, pp. 9-36.

[10] Y. Gurevich, B. Rossman and W. Schulte,“Semantic essence of AsmL”. *Theoretical Computer Science*, vol. 343(3), pp. 370–412, 2005.

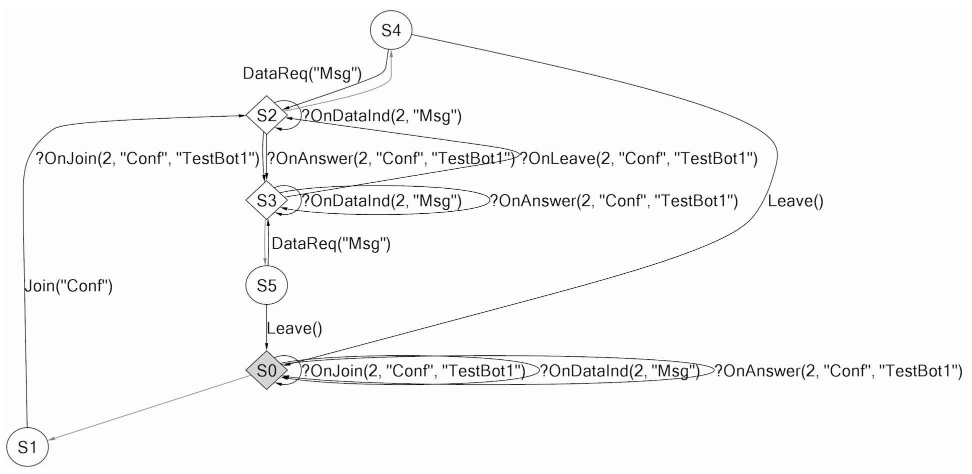
S4
DataReq("Msg")
S2
?OnDataInd(2, "Msg")
?OnJoin(2, "Conf", "TestBot1")
?OnAnswer(2, "Conf", "TestBot1")
?OnLeave(2, "Conf", "TestBot1")
S3
?OnDataInd(2, "Msg")
?OnAnswer(2, "Conf", "TestBot1")
Leave()
DataReq("Msg")
Join("Conf")
S5
Leave()
S0
?OnJoin(2, "Conf", "TestBot1")
?OnDataInd(2, "Msg")
?OnAnswer(2, "Conf", "TestBot1")
S1

Fig. 3. The model obtained by offline testing the IUT together with a single test agent

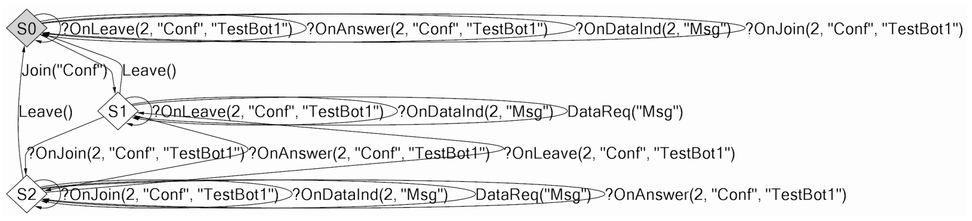
S0
?OnLeave(2, "Conf", "TestBot1")
?OnAnswer(2, "Conf", "TestBot1")
?OnDataInd(2, "Msg")
?OnJoin(2, "Conf", "TestBot1")
Join("Conf")
Leave()
Leave()
S1
?OnLeave(2, "Conf", "TestBot1")
?OnDataInd(2, "Msg")
DataReq("Msg")
?OnJoin(2, "Conf", "TestBot1")
?OnAnswer(2, "Conf", "TestBot1")
?OnLeave(2, "Conf", "TestBot1")
S2
?OnJoin(2, "Conf", "TestBot1")
?OnDataInd(2, "Msg")
DataReq("Msg")
?OnAnswer(2, "Conf", "TestBot1")

Fig. 4. The model obtained by online testing the IUT together with a single test agent

TABLE III

PERFORMANCE OF THE ONLINE TESTING COMPRISING TEST AGENTS WITH PREDETERMINED PAUSES

	After 1000 steps			After 10000 steps		
# Test agents	# Transitions	# States	State coverage	# Transitions	# States	State coverage
1	13	3	100%	17	3	100%
2	35	5	100%	44	5	100%
3	57	9	100%	86	9	100%
4	87	17	100%	190	17	100%
5	124	28	85%	292	33	100%
6	138	41	63%	414	59	91%
7	195	59	46%	499	95	74%
8	223	85	33%	844	160	62%

TABLE IV

PERFORMANCE OF THE ONLINE TESTING COMPRISING TEST AGENTS WITH RANDOMIZED PAUSES

	After 1000 steps			After 10000 steps		
# Test agents	# Transitions	# States	State coverage	# Transitions	# States	State coverage
1	13	3	100%	17	3	100%
2	36	5	100%	46	5	100%
3	73	9	100%	106	9	100%
4	170	17	100%	236	17	100%
5	205	32	97%	453	33	100%
6	428	61	94%	730	65	100%
7	493	112	87%	1255	128	99%
8	575	152	59%	2260	251	97%
9	689	238	46%	3341	447	87%
10	740	262	26%	4707	756	74%

[11] L. Heerink, J. Feenstra and J. Tretmans, "Formal Test Automation: The Conference Protocol with PHACT", in *Proceedings of IFIP 13th International Conference on Testing of Communicating Systems (TestCom 2000)*, H. Ural, R. L. Probert and G. von Bochmann, Eds. Kluwer Academic Publishers, 2000, pp. 211–220.

[12] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann and W. Grieskamp, "Optimal Strategies for Testing Nondeterministic Systems", in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA'04)*, ACM Press, 2004, pp. 55-64.

[13] V. Novakovic, "Formal Specification and Verification of Software" (in Croatian). Diploma Thesis, Department of Mathematics, University of Zagreb, 2006.

[14] Spec Explorer website: `http://research.microsoft.com/specexplorer/`.

[15] TorX website: `http://fmt.cs.utwente.nl/tools/torx/index.html`.

[16] M. Veanes, C. Campbell, W. Schulte and N. Tillmann, "Online Testing with Model Programs", in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)*, ACM Press, 2005, pp. 273-282.