PROSPECT - A Proposal for a New Test Specification Language and Its Implementation

Thomas Walter and Bernhard Plattner
 Computer Engineering and Networks Laboratory
 Swiss Federal Institute of Technology Zürich
 ETH Zürich - ETZ Building, 8092 Zürich, Switzerland
 e-mail: [walter | plattner]@tik.ee.ethz.ch

Abstract

Although internationally standardized (or are in the process of becoming international standard), the definition of TTCN and concurrent TTCN can still be improved. To solve some of the known problems we propose a test specification language, called *PROSPECT* - *PRO*to*CO*l test *SP*Eci*FI*cation language for *CO*nformance *TE*sting. The proposed language has a formally defined syntax and has a formally defined semantics. We have introduced *test modules* in our language so that the distribution of test components over real systems can be specified. As we assume that in every test module a main test component exists responsible for creating and terminating other test components, we have been in a position to give our language a precise semantics. In order to prove PROSPECT's suitability we have implemented a compiler and a runtime library. The implementation has been done in a UNIXTM environment. The details of our approach are described in this paper.

Keywords

Conformance testing, test specification, syntax, semantics, compilation

1 Introduction and Motivation

Research and development in test notations have been focussed on *TTCN* (*Tree and Tabular Combined Notation*) [11] and *concurrent TTCN* [12]. As recommended in [10], TTCN and concurrent TTCN are to be used for the specification of test suites. Whereas in TTCN only one *lower tester* (*LT*) and optionally one *upper tester* (*UT*) are active, concurrent TTCN supports the definition of complex test configurations, called *multi party testing context* (Figure 1), which, in general, consists of several concurrently running *test components* (*TC*). In every multi party testing context, the *lower tester control function* (*LTCF*) takes responsibility for setting up the test configuration, i.e. creation of lower and

upper testers, and the calculation of a final verdict. TCs are connected by channels, called *coordination points (CP)*, through which TCs exchange so-called *coordination messages (CM)*. The rules that govern the exchange of CMs and the coordination of pairs of LT and UT are defined in a *test coordination procedure (TCP)*.

Figure 1: Multi party Testing Context [10].

Communication between lower and upper tester and an *implementation under test (IUT)* is through *points of control and observation (PCO)* directly or via an underlying service provider (Figure 1). PCOs are “modeled as two queues: a) one output queue for control of test events to be sent towards the IUT; and b) one input queue for the observation of test events received from the IUT” [9]. Similarly, CPs “are modeled as two queues, one for each direction of communication” and may be realized “either by local communication or by communication that spans physical boundaries” [12].

Conceptually, a CP abstracts from the real communication infrastructure that connects test components (on either the same system or on different systems). Although CPs “are used to facilitate the exchange of coordination messages between test components”, “CPs shall not be used between an LT and a UT ...” [12]. As a consequence, CPs and CMs should be used only for communication between LTs and for communication between UTs. Communication between pairs of LT and UT is supported by different mechanisms. We believe that this is unnecessarily restrictive. Unfortunately, this is not the only flaw in the definition of TTCN and concurrent TTCN (see also [2]). Although LTs and UTs are to be run on different systems, the distribution of test components over systems cannot be made explicit. Having only one lower tester control function (LTCF) responsible for creation and termination of parallel test components (PTC) it is not obvious how the LTCF should perform these operations. One obvious solution is that the LTCF communicates with an “imaginary” entity on the remote system which is instructed by the LTCF to create or to terminate a specific PTC. However, this is not defined in concurrent TTCN and thus is only an ad hoc solution.

To solve the identified problems we propose a new test specification language *PROSPECT - PROtocol test SPECification language for Conformance Testing*. The main

design considerations are summarized as follows: firstly, in order to make the distribution of test components over systems explicit, we introduce *test modules*. A test module comprises all test components running on the same system. Secondly, on every system a lower tester control function exists managing all test components on that system. Thirdly, PROSPECT does not restrict the use of CPs; any two test components may be connected by a CP. The following sections discuss the design of the PROSPECT testing architecture (Section 2), the definition of syntax and semantics of PROSPECT (Section 3) and its implementation (Section 4). We conclude with a summary of related work and identify possible extensions of PROSPECT.

2 PROSPECT Testing Architecture

PROSPECT is a *test specification language*. Its definition is based on a specific testing architecture which is general in the sense that the defined conformance [10] and interoperability [8] abstract test methods (Figures 3 a) and 4 a)) are covered. Our testing architecture (Figure 2) is a slight modification of an existing testing architecture described in [6]. The PROSPECT testing architecture consists of a *test system*, an *implementation under test (IUT)* and a number of *implementation access points (IAP)*. A test system consists of *test components* and *links*. Test components are connected by links. *Links* are uni-directional communication channels. Test system and IUT communicate synchronously at IAPs. The difference of our approach with respect to [6] is that we do not distinguish between test components (i.e. LTs and UTs) and *test context*, i.e. an entity relating test actions performed by test components to actions at IAPs. Note that in [6] communication between test context and IUT at IAPs is not assumed to be synchronous. Furthermore, in [6] the test context is not part of a test system, whereas in our approach the test context is realized as a test component too, like UTs and LTs. All components except the IUT but including the test context are considered to be parts of a test system (Figure 2).

Figure 2: PROSPECT Testing Architecture.

The described testing architecture is the result of the following design considerations:

- It cannot be assumed that a test component can communicate directly with an IUT as, generally, an IUT is embedded in a real system. The test context is an abstraction of this embedding. It comprises the communication infrastructure needed for the interconnection of test components and IUT.

Figure 3: Distributed Test Method.

Figure 4: Interoperability Testing Architecture.

- In [9] we find the following statement: “It is assumed that the underlying service offered is sufficiently reliable for control and observation to take place remotely.” In order to increase the reliability of an underlying service we assume that even the underlying service is part of the test system and is (more or less) under the control of a test system operator.
- In most conformance testing scenarios it is not sufficient to have only one test component. In the *distributed test method* (Figure 3) and in the interoperability testing architecture (Figure 4) at least two components are participating in the execution of a test case. Therefore, in our testing architecture, a test system can consist of several test components.
- Since test execution involves several test components, it becomes essential to provide the instrumentation with which test components can coordinate their behaviour. An important part of this instrumentation are communication channels or links.
- Implementation access points are an abstraction of any mechanism for controlling and observing the behaviour of an IUT. It is for further study how IAPs are to be defined (or even standardized).

Figures 3 and 4 show how the distributed test method [10] and a proposed interoperability test architecture [8] map to our testing architecture: underlying service provider, upper and lower testers simply become test components.

3 Syntax and Semantics of PROSPECT

PROSPECT supports the specification of the behaviour of a test system. The hierarchical structure of the entities of PROSPECT is shown in Figure 5.

Figure 5: PROSPECT Entities and Structure.

Basic Process (BP): A basic process is a behaviour description of a test component. A behaviour description is an expression over the syntax given in Table 1.

Input Process (IP): An input process stores all messages for a test component which have been received from other test components in a queue-like structure, called *input stream*. Newly received messages are inserted at the end of the input stream. Stored messages can be read and retrieved in any order not just first-in-first-out.

Timer Process (TP): A timer process maintains two finite sequences of timers; one for running or active timers and one for expired timers. The sequence of running timers is ordered according to the timers' expiration times.

Test Component (TC): A test component is composed from a basic process, an input process, a timer process and a storage environment for variables and values. Test components can be created and terminated on demand by a specific test component, called *main test component*.

Test Module (TM): A test module consists of all test components that are supposed to be executed on the same real system. Test components in the same test module communicate asynchronously through communication channels called *routes*. Every test module has its own main test component.

Link Process (LP): A link process connects test components in different test modules. Link processes are unidirectional communication channels. The time required to transmit a message between TCs is a priori unknown but assumed to be finite.

Test System (Sys): A test system consists of link processes and test modules. It communicates synchronously with an IUT at interfaces called *implementation access points (IAP)*.

Routes are a concept which has already been used in SDL [22]. As in [22] we assume that routes do not introduce a delay on messages they convey. Every message which

is sent on a route to another test component is directly transferred into the receiving test component's input process. Thus, routes and input processes resemble much the semantics of *signal routes* and *input ports* [22]. Note that we distinguish between the reception of a message (in an input process of a test component) and its processing (by a test component). While reception of a message is always possible and is performed *synchronously* with the sending of the message, the processing is, however, performed *asynchronously* with the sending. Therefore we have stated above that communication between test components is asynchronous. With the obvious changes these arguments are also valid for the communication between link processes and test components. Routes and links have been introduced so that the realization of CPs "either by local communication or by communication that spans physical boundaries" [12] can be made explicit.

We assume a discrete time model where time is counted in time units of arbitrary but constant length. A global clock holds the current system time. In addition to the global clock, every test component maintains a local clock which is periodically updated with the current system time.

Example 1 We map the interoperability testing architecture (Figure 4) to PROSPECT (Figure 6) as follows: PROSPECT test components implement test components (Figure 4 b)). Test components that run on the same real system belong to the same test module. We have adopted the approach of mapping the service provider (Figure 4 b)) to several test components that belong to different test modules thus representing the physical distribution of the service provider. □

Figure 6: PROSPECT Testing Architecture of Figure 4.

3.1 Syntax of PROSPECT

PROSPECT defines a syntax for basic processes only (Table 1) since all other entities are either composed from lower layer entities (e.g. a test system consists of test modules and link processes) or are completely described by their characteristic property: A link, and this is also valid (with the obvious changes) for input processes and timer processes, is implemented as a FIFO-queue with the following operations pre-defined: the *append* operation to append a new message to the end of the queue, and the *first* operation to remove the first message from the queue.

For the description of data types and data terms we do not give an explicit syntax but assume that an algebraic specification is given from which data terms can be derived. An

interpretation of data terms is given in an algebra. Similarly, we assume existence of sets of identifiers for test components, timers etc.

CommunicationAction	::=	"input" "(" lrid "," dt ")" "otherwise" "(" lrid ")" "output" "(" lrid "," dt ")" "timeout" "(" tid ")"
BasicAction	::=	x "x=" dt "start" "(" tid "," dt ")" "cancel" "(" tid ")" "stop" "return" "create" tcid "(" bpid "[" lrid ₁ "," ... "," lrid _k "]" "(" dt ₁ "," ... "," dt _k ")" ")" "call" pcid "(" bpid "[" lrid ₁ "," ... "," lrid _k "]" "(" dt ₁ "," ... "," dt _k ")" "(" y ₁ "," ... "," y _m ")" ")" "terminate" "(" tcid ")"
Guard	::=	["(" bt ")"] [CommunicationAction]
ActionSequence	::=	BasicAction [";" ActionSequence] PriorityChoiceExpression
GuardedActionSequence	::=	Guard "->" ActionSequence
PriorityChoiceExpression	::=	GuardedActionSequence { "+" GuardedActionSequence }
BasicProcess	::=	PriorityChoiceExpression
dt, dt ₁ , ..., dt _k		are data terms
bt		is a Boolean term
lrid, lrid ₁ , ..., lrid _k		are link, route or IAP identifiers
tid		is a timer identifier
bpid		is a basic process identifier
tcid		is a test component identifier
pcid		is a procedure identifier
x, y ₁ , ..., y _m		are variable identifiers

Table 1: Basic Process Syntax.

PROSPECT distinguishes three types of actions: communication actions, basic actions and guards. Communication actions allow a basic process to communicate with its environment and to determine the status of timers. The communication actions (Table 1) are:

- **input and otherwise actions:** a basic process reads a specific message from the input process or from the IUT by performing an input action. A basic process performs an **otherwise** action to read any message that has been received from a specific route or link or the IUT.
- **output actions:** a basic process sends a message to another test component or an IUT by performing an output action.
- **timeout actions:** performing a **timeout** action enables a basic process to check whether a timer has expired.

Basic actions (Table 1) are for the assignment of data terms to variables, for the management of timers, for controlling test components and for dealing with procedures (**call** and **return**). Actions controlling the execution of test components are:

- **stop**: a basic process performs a **stop** action to indicate that it (and implicitly the test component to which it belongs) has successfully terminated.
- **create**: a **create** action instantiates a test component and assigns it a basic process. The instantiated test component runs in parallel with all other active test components.
- **terminate**: a **terminate** action explicitly terminates a test component irrespectively of its current activity. The **terminate** action should be used only by the basic process of the main test component.

A *guard* is a Boolean term followed by a communication action or only a Boolean term or only a communication action. Its meaning (see also Section 3.2) is that the guard is executed if the Boolean term holds and the communication action is possible. If the Boolean term holds and the communication action is possible we say the guard is *fulfilled*. An *action sequence* is either a basic action followed by an action sequence, which means that first the basic action is performed and then the basic process behaves as described by the action sequence; or is a priority choice expression. A *priority choice expression* is similar to a set of alternatives in TTCN and concurrent TTCN: It consists of a finite number of guarded action sequences (i.e. alternatives in TTCN). A guarded action sequence is composed from a guard and an action sequence. If the guard is fulfilled then the guard is executed and the basic process next executes the action sequence. The guards of the guarded action sequences of a priority choice expression are evaluated in sequence which means that the guard of each guarded action sequence is evaluated. The first guard which is fulfilled is executed and execution of the basic process continues with the action sequence that follows the guard. While evaluation of guards takes place no updates to the input stream or expired timers sequence should happen. A *basic process* is a priority choice expression.

3.2 An Operational Semantics of PROSPECT

An operational semantics is defined for every PROSPECT entity. This is done starting with the basic components and proceeding towards the top-level components. This way we emphasize the compositionality of the model (see also [5, 7, 18, 19]).

For every entity an operational semantics is defined by a *labelled transition system* (LTS) [16]:

$$LTS = (S, \epsilon, \rightarrow)$$

where S is a set of states, ϵ is a set of actions and $\rightarrow \subseteq S \times \epsilon \times S$ is a transition relation. An element in $\rightarrow$ is called *transition* and is denoted $s \xrightarrow{e} s'$. A transition is to be interpreted as follows: in state s an entity can perform action e and evolves into state

Figure 7: An Example of a PROSPECT Test System.

s' . The behaviour of each entity is given by all transitions which can be derived using inference rules of the form

$$\frac{t_1 \dots t_n}{t} \quad C$$

where $t_1, \dots, t_n$ and t are transitions and C is a predicate, called *side-condition*. Transitions $t_1, \dots, t_n$ can be transitions of the subcomponents of the entity under consideration or are transitions of the entity itself (this is true for all basic entities). Transition t is the inferred behaviour. An inference rule is applicable if the premises and predicate C hold. The concurrent behaviour of components, e.g. test components or links and test components, is represented by interleaving of actions (for more details see also [20]).

Example 2 Assuming we would like to assess conformance of an IUT which is expected to behave as follows: If the IUT receives a character string from its lower layer service provider, then the received character string should be forwarded to the IUT's upper layer service user.

The test system (Figure 7) initiates a service primitive at the IUT's lower layer interface with "Hello world" as parameter and then waits for an interaction with the IUT at the IUT's upper layer interface. The character string expected is the string "Hello world". The described behaviour is specified using two parallel test components interacting with the IUT and a main test component (MTC) that performs the test coordination. This design is made explicit in the following definition of basic processes:

```

MTC: (* Main Test Component *)
[true] -> create UT(UT [u, r3, r4]);           /* line 1 */
create LT(LT [l, r1, r2]);                     /* line 2 */
input(r2, cm_r) -> start(timer, now + 5);      /* line 3 */
input(r4, cm_v(pass)) -> v := pass; stop        /* line 4 */
+ input(r4, cm_v(fail)) -> v := fail; stop      /* line 5 */
+ timeout(timer) -> v := inconc; stop          /* line 6 */

UT: (* Upper Tester *)
input(u, ind('Hello world')) ->                /* line 7 */
output(r4, cm_v(pass)) -> stop                  /* line 8 */
+ otherwise(u) -> output(r4, cm_v(fail)) -> stop /* line 9 */

LT: (* Lower Tester *)
output(l, req('Hello world')) ->               /* line 10 */
output(r2, cm_r) -> stop                       /* line 11 */

```

Test component MTC creates parallel test components UT and LT (lines 1 and 2). Then MTC waits for an input from LT (line 3). LT indicates that it has sent the character string "Hello world" to the IUT (line 10) by sending a coordination message `cm_r` to MTC (line 11). If MTC receives this coordination message (line 3), it starts timer `timer` (line 3) and then it waits for a coordination message from test component UT (lines 4 - 6). Upon reception of the character string "Hello world" from the IUT (line 7), UT sends a coordination message with verdict `pass` to MTC (line 8); otherwise (line 9) it sends a coordination message with verdict `fail` (line 9). Test component MTC is prepared to accept either the `pass` (line 4) or the `fail` CM (line 5). If due to some unforeseen conditions, MTC does not receive any coordination message at all, then, as a third alternative (line 6), MTC checks whether timer `timer` has expired in which case the test result is `inconclusive`.

We assume that all actions are atomic, i.e. if two or more actions are performed simultaneously, then this is an indication that entities synchronize and communicate. Although actions are atomic we do not assume that actions are instantaneous in the sense that actions occur "without consuming time" as in [3]. Furthermore, the time interval between the end of one action and the start of the next action is unknown but assumed to be finite. Figure 8 represents the execution of test component LT under the stated assumptions. Note that after an update of system time, the test component's local clock has to be updated (dotted lines) before a next action is executed.

Figure 8: Execution of Test Component LT.

Given the following inference rules for basic processes (BP) and test components (TC)

$$\frac{}{GAS \xrightarrow{\text{output}_{BP}} AS} \quad \frac{GAS \xrightarrow{\text{output}_{BP}} AS}{TC \xrightarrow{\text{output}_{TC}} TC'}$$

we can derive by applying the inference rules that PTC LT can perform an output action in its initial state as follows:

$$\frac{GAS \xrightarrow{\text{output}_{BP}} AS}{TC \xrightarrow{\text{output}_{TC}} TC'}$$

Similarly, we infer that PTC LT can perform another output and **stop** which results in the action sequence shown in Figure 8. Note that before an action is executed the test component performs an internal action due to an update of LT's local clock.

Since we model the parallel execution of actions by interleaving, a specific schedule of the interleaved execution of test components MTC, UT and LT is shown in Figure 9 a). The execution of the test system is shown in Figure 9 b). Communication between test components LT and UT and the IUT is synchronous (dotted lines).

Figure 9: Execution of a Test Module and Test System.

The initial state of a test system is defined as follows: every link process is initialized with an empty message queue; every test module is initialized with the main test component only; the MTC has an empty input stream and empty sequences of running and expired timers, and its storage environment is initialized, i.e. variables are assigned to locations and get their initial values. $\square$

PROSPECT is flexible so that all conformance abstract test methods and interoperability testing architectures can be mapped to PROSPECT test systems. By introducing the notion of a test module we are providing a means that enables test case specifiers to make the distribution of test components over real systems explicit. As we assume that in every test module an MTC exists which is responsible for creating and terminating parallel test components, we are able to define a semantics for the **create** and **terminate** basic actions.

In our approach test components communicate through routes (if the test components are in the same test module) or through link processes: No difference exists whether a test

component communicates with a local or remote test component. This is of advantage for the definition of test coordination procedures. (In concurrent TTCN test coordination between upper and lower testers and between lower testers and between upper testers are separate issues. Coordination messages and coordination points are used for the communication between lower testers and between upper testers. For the communication between lower tester and upper tester a test management protocol must be defined).

4 PROSPECT Implementation

In order to prove our approach feasible we have implemented PROSPECT in a UNIXTM environment. The implementation consists of a compiler for the behaviour part of test cases and a runtime library.

4.1 Compilation

We have implemented a PROSPECT compiler using *lex* and *yacc* [14]. Compilation of a PROSPECT test case is done in two steps: firstly, programme P2C (PROSPECT to C) (Figure 10) is executed that translates PROSPECT source files to C [13] source files. Secondly, the generated C source files are compiled and linked. Besides the generated C sources, the C compiler makes use of the following files (Figure 10):

- `_asp.*`: These files contain declarations and definitions of message types, i.e. abstract service primitive types and coordination message types.
- `_constraints.*`: These files contain declarations and definitions of test data (or constraints), i.e. instances of abstract service primitives and coordination messages.
- `_support.*`: These files contain type definitions, function definitions etc. that are referenced from within a PROSPECT test case. As the previously mentioned files, these files are application specific.

Note that these files are not generated by P2C but are assumed to be supplied by a PROSPECT user. As their names suggest the files have to be (manually) derived from the algebraic specifications of *abstract service primitives (ASP)*, coordination messages (CM), *protocol data units (PDU)* and the data terms for ASPs, CMs and PDUs. Furthermore, if necessary, a PROSPECT user has to provide supporting code for encoding and decoding of PDUs and CMs that are exchanged between test components and test system and implementation under test (IUT).

4.1.1 Compilation of a create

A `create` is compiled as follows: The MTC first executes a `fork` system call. This system call creates a copy (called the *child* process) of the process that is being executed (called the *parent* process). The child process then executes an `execlp` system call which substitutes the old process image with a new one. As part of the `execlp` system call, actual parameters are passed to the newly instantiated programme. After the MTC has

Figure 10: PROSPECT Compilation.

forked a PTC, the MTC continues execution of its basic process. The code fragment for the translation of `create LT(LT [1, r1, r2]())` looks as follows:

```

if ((pid = fork()) == -1) {
    /* error handling */
    /* ... */
}
else if (pid == 0) {          /* child process = PTC */
    execlp("LT", "LT", itoa(LT), (char *) 0);
    /* error handling */
    /* ... */
}
else {                        /* parent process = MTC */
    /* ... */
}

```

where the first LT identifies the object code file to be executed and the second LT is the parallel test component's name. The third argument passed is the parallel test components unique (with respect to a specific test system configuration) identifier. Some other information needed by the created PTC, e.g. link and route identifiers, is read from a configuration file by the PTC itself during its initialization phase.

4.1.2 Compilation of a terminate

In the parent process, upon return from the `fork` system call, the process identifier (an integer) of the child process is known. The parent process maintains an array of process identifiers of child processes. In order to `terminate` a parallel test component, an MTC retrieves the PTC's process identifier and then performs a `kill` system call which terminates the parallel test component unconditionally. For instance, the PROSPECT statement `terminate(LT)` is translated to

```

int tcs[MAXPTC];
/* any other code */
kill(tcs[LT], 9);
/* any other code */

```

where parameter LT is used as an index into array `tcs` of process identifiers. The array `tcs` is updated whenever an MTC creates a parallel test component (see also the code fragment above):

```

if                                /* code as shown above */
/* ... */
else {                            /* parent process = MTC */
    tcs[LT] = pid;                /* <--- change performed here */
}

```

4.1.3 Compilation of Priority Choice Expressions

The evaluation of a priority choice expression is assumed to be atomic (Section 3.1). If a test component evaluates a priority choice expression, no other process (link process, test component or timer process) shall change resources used by the test component, i.e. input stream or expired timers sequence. In order to exclude other processes from accessing critical resources we use semaphores (Section 4.2) to synchronize parallel processes.

Evaluation of a priority choice expression may find all guards unfulfilled. In such a case evaluation of guards is repeated until one guard is fulfilled. Thus, translation of a priority choice expression becomes:

```

for (;;)    /* loop forever until a guard succeeds */
    sem_p(); /* semaphore operation 'P' */
    if ( /* translation of first guard comes here */ ) {
        /* success */
        sem_v(); /* semaphore operation 'V' */
        /* ... */
    }
    /* ... */
    if ( /* translation of last guard comes here */ ) {
        /* success */
        sem_v();
        /* ... */
    }
    sem_v();

```

The `sem_v()` function calls are necessary to enable other processes to access critical resources, since otherwise, if all guards are once unfulfilled no progress can ever be made.

4.1.4 Miscellaneous

The translation of other PROSPECT statements is also simple. We consider a few examples: `stop` is compiled to `exit(0)`. `return` (in PROSPECT) becomes `return` (in C). An assignment like `v := pass` is mapped to `v = pass`. An input action is translated to a function call in an `if` control statement:

```
input(r2, cm_r);
```

is compiled to

```
if (input(r2, &cm_r) == 0) /* ... */
```

where the constraint, whose declaration and definition are in `_constraints.*`, is passed as a call-by-reference parameter and 0 is returned upon success of the `input` function call.

Timers are implemented as follows: If a timer is started the process performs a `fork` system call and (as the parent process) continues execution. The child process (`u`)sleeps for the specified amount of time. If the `usleep` system call returns, the child process puts a message into its parent's message queue (Section 4.2.1). The message's type identifies the timer that has expired.

4.2 PROSPECT Runtime Support - PROSPECT library

The implementation of a runtime library for PROSPECT has been influenced by the structure of a test system (Figure 5) and by our choice of the UNIXTM environment.

4.2.1 Message Queues as an Implementation of Input Processes

Message queues [17] are interprocess communication mechanisms that enable processes to communicate by message passing. A message queue can be shared between several processes. Processes can read and write messages from and to the queue. In our application, link processes or test components write messages to a message queue and the test component for which the message queue is implementing the input process reads messages from the queue. Messages have the following attributes: a message type, an indication of the length of the data part of the message, and the data. The message type is used to identify the link process or route or the timer process which has written the message. The message's data part is, in our context, either an abstract service primitive or a coordination message or empty.

A test component can read a message from its associated message queue by calling the message queue's `receive` function. To identify a specific message the test component has to set the message type appropriately. The mapping from link processes, routes and timer processes to message types is done statically during test case compilation.

There are two advantages of message queues which makes them particularly suitable for our application:

- Sending of a message can be done asynchronously to the receiving and processing of a message.
- Reception of a message of a particular message type (i.e. from a specific link or route) is strictly sequential. This complies with the requirement that only the first message received from a link or route and stored in the message queue should be retrieved.

4.2.2 Sockets as an Implementation of Link Processes

The UNIXTM socket library provides functions to interconnect processes on different systems. Our implementation of link processes is based on *stream sockets* and uses the TCP/IP protocol stack.

A link process is implemented as a client-server pair. The link process client establishes a connection to a link process server and subsequently transmits coordination messages to

the link process server. The link process server reads the messages and puts the messages into a test component's message queue.

4.2.3 Semaphores for the Synchronization of Processes

The updating of message queues is critical for the correct evaluation of priority choice expressions. While a test component evaluates a priority choice expression, the content of its message queue should not be changed. Access to a test component's message queues is guarded by semaphores.

In System V semaphores are integer valued variables which are implemented in the UNIXTM kernel. This guarantees that the sequence of operations to update a semaphore is executed atomically. The functions implemented for a semaphore are generalized versions of the P and V operations [4].

To access a resource, a process calls the P operation. If the resource is currently used by another process the calling process is put on a *wait* queue. If the resource becomes available, then the first process in the *wait* queue is activated. A process releases a resource by calling the V operation.

A link process or test component or timer process which intends to update a message queue has to obey the following protocol:

```
P(); /* execute P operation */
/* perform update of message queue */
V(); /* execute V operation */
```

A test component that is about to evaluate a priority choice expression has to perform the protocol steps:

```
P(); /* execute P operation */
/* evaluate the guards of a priority choice expression in sequence */
V(); /* execute V operation */
```

In combination, both protocols prevent the updating of a message queue while evaluation of a priority choice expression is in progress. Furthermore, it is guaranteed that if a process calls a P operation then it will eventually gain access to the message queue.

5 Conclusions

In this paper we have proposed the test specification language PROSPECT and have discussed its implementation. The definition of PROSPECT has been influenced by TTCN and concurrent TTCN [11, 12] and SDL [22]. For instance, main and parallel test components are known from concurrent TTCN and routes are a concept introduced in SDL. PROSPECT extends current approaches for the definition of test notations (see [1, 15] for a definition of *TELL - Test Language LOTOS*): We have introduced test modules in order to enable test case specifiers to define the distribution of test components over real systems. With respect to TELL, PROSPECT supports the use of timers which is not

possible in TELL. However, we admit that our time model is rather high level so that it has to be refined to be applicable in e.g. QoS assessment or testing real-time applications.

We have also discussed the implementation of PROSPECT which consists of a compiler for PROSPECT test cases and a PROSPECT runtime library. The implementation has been done in a UNIXTM environment. We have made the experience that the implementation of some parts of PROSPECT has been simple in this environment. Particularly useful have been the different interprocess communication facilities available in UNIXTM.

The applicability of PROSPECT is limited since, as pointed out above, timing constraints cannot precisely be specified. With the upcoming deployment of multi-media applications, like video-conferencing, multi-media archiving and retrieval, tele-teaching, etc. not only *functional* properties of an implementation are to be assessed during conformance testing but also *non-functional* properties, e.g. timing constraints (as in real-time applications), quality-of-service (QoS) aspects, synchronization of different data streams (audio, video, textual information). None of the known test notations can deal with these requirements. To close this gap, we have started a research project with the goal to define and implement a test specification language that supports test case specification for multi-media applications [21]. PROSPECT is supposed to be used in this project.

Acknowledgment The authors would like to gratefully acknowledge the anonymous referees for fruitful comments to the previous version of this paper.

References

- [1] Ahooja, R., Burmeister, J., de Meer, J., Rennoch, A., *Method for Open Test Sequences and their Implementation generalized in a Conformance Test Tool*, Proceedings PTS II, North-Holland, 1989.
- [2] Baumgarten, B., *Open Issues in Conformance Test Specification*, Proceedings PTS VII, 1994.
- [3] Bolognesi, T., Brinksma, E., *Introduction to the ISO Specification Language LOTOS*, Computer Networks and ISDN Systems, Vol. 14, North-Holland, 1987.
- [4] Dijkstra, E., *Cooperating Sequential Processes*, Genuys, F. (Ed.), *Programming Languages*, Academic Press, 1968.
- [5] Walter, T., Ellsberger, J., Kristoffersen, F., Merkhof, P.v.d., *Methods for Testing and Specification (MTS) Semantical relationship between SDL and TTCN A Common Semantics Representation*, European Telecommunications Standards Institute, ETR 071, 1993.
- [6] ISO/ITU-T, *Formal Methods in Conformance Testing*, ITU-T TS SG10 Q8 and ISO SC21 WG1 P54 Southampton output, 1994.
- [7] Godskesen, J., *An Operational Semantic Model for Basic SDL*, Tele Danmark Research Report TFL RR 1991-2, 1991.

- [8] Hogrefe, D., *Conformance testing based on formal methods*, Proceedings FORTE '90, North-Holland, 1990.
- [9] ISO/IEC, *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts*, ISO/IEC 9646-1, 1994.
- [10] ISO/IEC, *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 2: Abstract Test Suite specification*, ISO/IEC 9646-2, 1994.
- [11] ISO/IEC, *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)*, ISO/IEC 9646-3, 1992.
- [12] ISO/IEC, *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN): Amendment 1: TTCN Extensions*, ISO/IEC 9646-3 DAM 1, 1993.
- [13] Kernighan, B., Ritchie, D., *The C Programming Language*, Prentice Hall Software Series, 2nd Edition, 1988.
- [14] Levine, J., Mason, T., Brown, D., *lex & yacc*, O'Reilly & Associates, 1992.
- [15] de Meer, J., Burmeister, J., Rennoch, A., Schröer, I., *An Approach to a Conformance Testing Methodology and the COAST Test Systems*, Proceedings PTS III, North-Holland, 1991.
- [16] Plotkin, G., *A structural approach to operational semantics*, Aarhus University, Computer Science Department, 1981.
- [17] Stevens, W. Richard, *UNIX Network Programming*, Prentice Hall Software Series, 1990.
- [18] Walter, T., Ellsberger, J., Kristoffersen, F., Merkhof, P.v.d., *A Common Semantics Representation for SDL and TTCN*, Proceedings PSTV XII, North-Holland, 1992.
- [19] Walter, T., Plattner, B., *An Operational Semantics for Concurrent TTCN*, Proceedings PTS V, North-Holland, 1992.
- [20] Walter, T., *PROSPECT - A Contribution to Protocol Specification, Conformance, and Interoperability*, Dissertation 10419, ETH Zürich, 1993.
- [21] Walter, T., Grabowski, J., *Towards the new Test Specification and Implementation Language 'TelCom TSL'*, 5. GI/ITG Workshop Formal Description Techniques for Distributed Systems, Kaiserslautern, June 1995.
- [22] IUT, *Functional Specification and Description Language SDL*, ITU-T Recommendation, 1992.