TTCN-3 for Large Systems*

Thomas Deiß†
Nokia Research Center
thomas.deiss@nokia.com

April 5, 2005

Abstract

TTCN-3 supports several concepts that allow to describe quite complex test scenarios in a convenient way. For example, concurrency and both message and procedure-based communication are supported. Nevertheless there are several problems that repeatedly occur when writing large test systems and where TTCN-3 does not have built-in solutions. In this paper we will identify several such problems and propose solutions.

1 Introduction

The telecommunications industry has been using TTCN-3 [3] and its predecessor TTCN successfully to test telecommunication systems. The still increasing complexity of telecommunication systems implies that the complexity of the test systems is also increasing. To achieve both the necessary quality of the telecommunication systems that are delivered to the markets as well as a reduction in the time to market the test systems need to be built in an effective way. One way to increase the effectiveness of building the system is to increase the degree of reuse. But if this reuse occurs in an unplanned way, then there is a high probability that the reuse does not pay off. More time might be spent in finding reusable parts from some repositories and to adapt them to the situation at hand than is gained by avoiding to write new code. And as the ultimate prerequisite for reuse, there must be any reusable parts.

TTCN-3 as a language offers already two communication paradigms: message-based and procedure-based communication. It also offers a notion of concurrency by the possibility to execute statements on several parallel test components in a single test case. Although TTCN-3 is typically used to describe the behaviour of test cases, but not as a programming language for communicating systems, there are problems from the area of concurrent programming that occur also when writing test cases. These and other features of TTCN-3 allow to define complex test scenarios in a reasonable way.

In this paper we describe several problems that occur in the context of testing large systems and that we already experienced or anticipate to occur within the next test systems we have to build using TTCN-3. The problems are in both areas mentioned above: how should the TTCN-3 test cases be written such that the degree of reuse can be increased and as a consequence the development effort be reduced and how could solutions to repeatedly

*This report is a largely extended version of [2]

†The opinions in this report are those of the author, but not of his employer.

occurring problems be written in TTCN-3. Because the development of solutions to standard problems naturally has implications on writing reusable TTCN-3 code we will cover both aspects in this paper. As a side effect of providing solutions to the problems we could evaluate whether TTCN-3 as a language is suitable, or whether it has specific shortcomings.

The structure of this paper is as follows: At first we explain problematic areas with several scenarios, cf. Section 2. Thereafter we explain solutions that each cover one or more scenarios in Sections 4 and 5. In Section 6 we some more high-level information about the TTCN-3 implementations. In Section 7 we discuss related work. We give a summary in Section 8. The code is added as appendices.

Note that the code examples in this paper have been written in different ways to show that that there is not a single way to write TTCN-3 code. The reader is encouraged to look at the examples and to play around with these various styles. If a reader intends to use the examples themselves, then the examples should be adapted to the TTCN-3 style already in use.

2 Usage Scenarios

In this section we will identify several problems that might occur when describing complex test cases. We will do so by describing scenarios of TTCN-3 usage to highlight the problems. For some of the scenarios we will explain why certain specific solutions do not work or do not scale up.

Clearly, not all of these problems will turn up in each complex test case.

2.1 Replication

There are a lot of test cases that try to spot errors that occur if one instance of a specific communication endpoint is interacting with the system under test or the behaviour on a single connection is investigated. But then there is also the question whether the system under test behaves correctly if several instances of the same communication endpoint interact with the system under test or what happens if there are several connections.

The question is how must the TTCN-3 code for the single instance be written such that it can easily be instantiated several times. As an example if some identifier for the communication endpoint is defined by a module parameter and directly accessed from the functions defining the behaviour, then this behaviour cannot be executed on several test components in parallel because all of them would use the same module parameter and hence the same identifier. By using a module parameter that is of an array like type several different values can be provided. But then the test components must know their index position in this array.

2.2 Broadcast

If several test components are executing concurrently in a test case, then one of them might want to deliver information to all test components. Such information could be

- configuration information, e.g. how the system under test should be accessed, that is known already at the start of a test case
- plainly data, e.g. state information about the system under test that is obtained by one test component and forwarded to the other test components

Regarding configuration information this is typically solved by either using module parameters or by passing the information as parameters in function calls. The usage of module parameters has the problem that the definitions are fixed and does not fit nicely when parts of the test systems are replicated and each part should have its own instance of the module parameter. Passing parameters is fine because it makes the information flow explicit, but can become cumbersome when the list of parameters becomes longer and longer. For data that is not available at the beginning of a test case neither module parameters nor passing data as a parameter is appropriate. Here the data has to be distributed explicitly by message passing or by calling procedures.

2.3 Access to Shared Data

Several test components might need access to some common data. Because TTCN-3 has no concept of global variables, some other means must be used to hold data and to provide access to the data for several test components. Such data might also have to be made available from one test case to another. Throughout a single test campaign execution this could be achieved by passing this data through **inout** parameters of the test cases from one test case to another. But if the amount of data is large, then it might be more convenient to store the data to file and load it again.

2.4 Synchronization

When several test components are executing code simultaneously this concurrent execution has to be synchronized. The most typical phases are the *preamble*, the *testbody*, and the *postamble* that must be synchronized among the test components. But there are also other synchronization points, e.g. start and stop of data transmission from several test components with the system under test. Another example is to open several connections and to await termination of each of these connections before further actions are taken.

2.5 Unique Identifiers

Throughout a test case different test components might need unique identifiers, e.g. as transaction identifiers, towards the system under test. This could be achieved by passing around the 'latest' used identifier and use this one as the basis to derive a new identifier. But passing this around works only to a certain extent, because it must be guaranteed that at most one test component at a given time wants to generate a unique identifier. Also, passing the latest used identifier around all the time is just cumbersome.

3 Absence of Context Restrictions

To be reusable, a test component and the behaviour executed on it should put as few as possible restrictions on its environment. This absence of context restrictions is nothing specific to TTCN-3, it is actually standard software engineering practice. Therefore we do not go into more detail here. Instead we will just give a single example.

Assume a TTCN-3 function that maps a port of the component on which it is executing to a port of the test system interface. As an example consider the following code:

```
function f_configure () runs on TCType
{
  \\ ...
  map( self:pt_a, system:pt_b );
  \\ ...
}
```

where `TCType` is the name of a component type and `pt_a` and `pt_b` are port names. The context in which this test component can be used is restricted: The component cannot be executed in a topology of test components where there is an intermediate test component between the test component itself and the test system interface. In such a configuration the port of the test component has to be *connected* using a **connect** statement. As a consequence **connect** and **map** statements for a test component should not be executed on the test component itself, but by another test component. A good candidate to do so would be the test component that created that specific test component.

4 Functions with Optional Parameters

In some of the examples we use functions where some parameters are considered as optional ones. But in TTCN-3, functions have a fixed number of parameters. Therefore we will define a wrapper function where all optional parameters are collected in a record with optional fields. Because default values for optional parameters cannot be defined directly in the definition of the record, we define a constant that defines the default values. This usage of records for optional parameters is applicable to **in** parameters.

For example let `f` be the name of a function with two mandatory and three optional parameters. For the sake of simplicity the type of each of these parameters is **integer**, the names of mandatory parameters are `m1` and `m2` and the names of the optional parameters are `p1`, `p2`, and `p3`. We define a type `OptPar_f` to hold the optional parameters

```
type record OptPar_f {
  integer o1 optional,
  integer o2 optional,
  integer o3 optional
  } //endtype OptPar_f
```

and a constant `c_defaultPar_f` to hold the default values for omitted optional parameters

```
const OptPar_f c_defaultPar_f := {
  o1 := 3,
  o2 := 5,
  o3 := 0
} //endconst
```

The function `f` would itself be defined as

```
function f ( in integer   p_v_m1,
             in integer   p_v_m2,
             in OptPar_f  p_v_o )
  ...
} //endfunction f
```

This function is just be a wrapper around the actual function that will be called with all omitted optional parameters replaced by their defaults. Because we consider the wrapper function `f` as the interface and the actual function as its implementation we will call the actual function `f_impl`. The function `f_impl` will have mandatory parameters and will also not use the record of optional parameters. To replace the omitted optional parameters with their default values we will use a function `f_optPar`:

```
function f_optPar ( in OptPar_f p_v_o ) return OptPar_f {
  var OptPar_f o := p_v_o;
  if (not ispresent(p_v_o.o1)) { o.o1 := c_defaultPar_f.o1 };
  if (not ispresent(p_v_o.o2)) { o.o2 := c_defaultPar_f.o2 };
  if (not ispresent(p_v_o.o3)) { o.o3 := c_defaultPar_f.o3 };
  return (o);
} //endfunction f_optPar
```

Using such a function, the function `f` itself can be defined stereotypically as

```
function f ( in integer   p_v_m1,
             in integer   p_v_m2,
             in OptPar_f  p_v_o )
return integer
{
  var OptPar_f o := f_optPar(p_v_o);
  return ( f_impl( p_v_m1, p_v_m2, o.o1, o.o2, o.o3) );
} //endfunction f
```

Alternatively to the solution proposed above, all parameters, mandatory and optional ones, can be kept in a single record which would then be the sole parameter of `f`.

5 Component Clusters

To address broadcast, storage of data, generation of unique identifiers, and synchronization in a replicable or scalable way we propose test configurations that consist of test components, where the components take different roles. In Section 5.1 we explain the general idea and define general types that can be used in test configurations. To become independent of the specific types used in a test system we define a type to hold basic values of basic types in Section 5.2. In the remainder of this section we explain how commonly used functionality can be implemented in TTCN-3.

5.1 General Architecture

We propose to use test configurations where test component takes one of three different roles:

Worker: These are the test components actually determining the outcome of the test case

Supervisor: Such a test component is responsible to create, control, and monitor the other test components

Servers: These are test components that provide a specific functionality to the other workers

Each worker will use a dedicated port to exchange messages with a specific server.

5.2 Datatypes

The TTCN-3 type **anytype** is restricted to the *known* datatypes in the module in which it is used. This means that only basic TTCN-3 types, types defined in this module, and types imported from other modules can be used as facets of **anytype**. Therefore the type **anytype** cannot hold arbitrary values defined by a test system developer. Nevertheless, this **anytype** is still a union over all the basic types and string types. This allows to store already a number of reasonable values. Therefore we define a module, of which actually just its anytype will be used by other modules. To make this **anytype** accessible to other modules we define a synonym `UserData` of it.

```
module userData {

  type anytype UserData;

  type record of UserData RoUserData;

} // endmodule userData
```

To extend the set of values that can be stored, we define an unrestricted record over `UserData`. Note that this record of type is known in this module, and therefore can be a facet of the type **anytype**. Therefore it becomes possible to create more deeply nested values.

5.3 Broadcast

To distribute a message to several other test components we define a broadcast server. Each worker and the supervisor can send a message to the server, which in turn sends the message to each of the other components.

5.3.1 Broadcast Synchronization

Message broadcast can be done in three different ways:

Unsynchronized: The message is forwarded by the server to each test component except the requesting test component. No confirmation is sent to the requesting component.

Partially synchronized: The message is forwarded by the server to each test component except the requesting component. When the message has been sent to each of the other components the server sends a confirmation to the requesting test component.

Fully synchronized: The message is forwarded by the server to each test component except the requesting component. Each of these test components has to acknowledge the receipt of the message. Only then the server will send a confirmation to the requesting test component.

As an example where fully synchronized broadcast is useful consider a request to terminate a complete test configuration. Upon receiving a shutdown request each test component can perform its shutdown or postamble operations and thereafter acknowledge the receipt of the shutdown message.

5.3.2 Registration

TTCN-3 has no built in broadcast or multicast operation. Therefore the possible recipients of broadcast messages have to be registered with a broadcast server. A test component can register either itself or it can be registered by another component. To register itself a component can send a message of type `RegisterReq` with the optional parameter omitted. In this case the sender of the message will be registered. Otherwise, the component referred to in the parameter is registered.

```
type record RegisterReq { Worker recipient optional };
type record RegisterRsp {};
type record UnRegisterReq { Worker recipient optional };
type record UnRegisterRsp {};
```

5.3.3 Server Structure

The server contains a component variable `g_v_workers` to store those workers to which a message will be broadcasted. This variable is of the type `WorkerSet` and is updated according to received `RegisterReq` and `UnRegisterReq` messages.

```
type component BcComp {
  port BcPort    g_pt_bc;
  var  Worker    g_v_sender;
  var  WorkerSet g_v_workers;
} // endtype component BcComp
```

5.3.4 Broadcasting

To broadcast a message is requested by a message of the type `DataReq`. Messages of the type `DataRsp` are used to respond when the message has been broadcasted or when all recipients have responded to the broadcasted message.

```
type record DataReq { UserData data, SyncKind syncKind };
type record DataRsp {};
```

The server has two states, which correspond to two functions `f_idle` and `f_awaitReplies`, respectively. In the function the various requests can be received and are processed . On reception of messages of the types `RegisterReq` and `UnregisterReq` the set of registered workers is changed accordingly. Messages of the type `DataReq` are forwarded to all registered workers. If this request is unsynchronized or partially synchronized then either no response is sent or a response is send without awaiting a response from the recipients.

The actual processing of the requests takes place in the altsteps. If one considers a request as consisting of an input, an action, an output, and a next state, then the input and the action are handled in the altsteps, the output to the originator and the next state are handled with in the alternatives. If needed, the altsteps can pass information to the alternatives either by changing component variables or by an **out** parameter.

```
function f_idle () runs on BcComp {
  var SyncKind  syncKind;
  var UserData  data;
  alt {
    [] alt_registerReq() { repeat };
    [] alt_unregisterReq() { repeat };
    [] alt_dataReq(syncKind) {
      if ( e_none == syncKind ) {
        // nothing to do
        repeat;
      } else if ( e_partial == syncKind ) {
        //acknowledge broadcast immediately
        g_pt_bc.send( { dataRsp := {} } ) to g_v_sender;
        repeat;
      } else {
        // e_full == syncKind
        // await the replies
        f_awaitReplies();
      }
    }
    [] g_pt_bc.receive { repeat }
  } //endalt
} //endfunction f_idle
```

If the request is fully synchronized, the function corresponding to the second state will be called. In this function the correct number of messages of type `dataRsp` are expected and if these are received then again the function `f_idle` is called. While these responses are awaited unregister requests are handled, all other requests are silently discarded. The unregister requests must be handled here because a client might unregister immediately after sending a message of type `dataRsp`. In that case the message queue of the server will contain both messages of type `dataRsp` and of type `unregisterReq`.

```
function f_awaitReplies () runs on BcComp {
  var integer recipientAmount := sizeof(g_v_workers);

  // message has not been broadcast to sender
  if ( isElement( g_v_workers, g_v_sender ) ) {
    recipientAmount := recipientAmount - 1;
  }

  alt {
    [recipientAmount > 0] g_pt_bc.receive( { dataRsp := ? } ) {
      recipientAmount := recipientAmount - 1;
      repeat;
    }
    [recipientAmount > 0] alt_unregisterReq () {
      // allow handling of unregister
      // this will cause problems if a client unregisters instead of
      // providing a data response
      repeat;
    }
    [recipientAmount > 0] g_pt_bc.receive {
```

```
        // consume any other message, including new data requests
        repeat;
      }
      [recipientAmount <= 0] alt_else() {
        g_pt_bc.send( { dataRsp := {} } ) to g_v_sender;
        // wait for next request
        f_idle();
      }
    } //endalt
  } //endfunction f_awaitReplies
```

Note, that instead describing the loop in the code above using **repeat** statements, it would also be possible to use a **while** statement.

5.3.5 Possible Extensions

The code shown so far for the broadcast server implements only basic behaviour and can be extended both in terms of functionality as well as usefulness:

Performance There has been no attempt to write specifically efficient code. As an example, the set of registered workers has been implemented as a list-like type, where insertion and removal operations require time linear in the number of previously inserted elements.

Robustness I The code can be made more robust. As an example, after broadcasting data and if full synchronization has been requested, then only the correct number of responses is awaited. It is not checked whether each recipient send exactly one response.

Some issues remain that cannot be resolved: When a registered worker stops execution and there is an attempt to broadcast data to it before it is deregistered this will inevitably lead to a runtime error. Even when checking directly before forwarding the data whether the recipient is still executing, the recipient could stop its execution between the check whether its running and before forwarding the data.

Robustness II The data request and response messages do not have a special field that allows to identify a specific request. The broadcast server is therefore not able to decide whether a data response is actually corresponding to a specific data request or whether it is a duplicate or superfluous response for a previous request.

Broadcasting groups Instead of a single set of registered workers the server could maintain several such sets and requests to broadcast messages could be made specifically for the workers in such a group.

Overlapping operations With the code shown here new requests are silently discarded while the server awaits responses for fully synchronized broadcasts. Instead of discarding such requests these could be saved in a queue for later processing. Or by creating a parallel test component for each fully synchronized broadcast, awaiting responses could be completely decoupled from processing of further requests. The latter could be especially helpful to implement simultaneous broadcast request to different groups of workers.

Although some kind of broadcasting can be implemented in TTCN-3 this way, the approach here is cumbersome to use. Firstly, the necessary connections in the test configuration need to be created and the intended recipients need to be registered. Secondly, and more importantly, as it is not possible to store messages of arbitrary type, which is need to send it to one worker after another, in the broadcast server the actual message must be converted to a value of type `UserData` and this one is broadcast and must be interpreted on the receiver side correctly. This is both cumbersome to handle and additional effort is needed to retain type information in the broadcasted message.

This certainly shows an inherent limitation of TTCN-3. To overcome this it seems to be most suitable to extend the language, e.g. by a allowing a statement like `pt.send(a_msg) to all` to broadcast `a_msg` to all component ports connected to the port `pt`.

Another solution to overcome this cumbersome handling without language extensions could be to implement such a broadcast server not as a parallel test component, but as a part of a System Under Test Adapter (SA), cf. [4] for a general architecture of TTCN-3 test systems. In the SA it might be possible to avoid the limitation that only values of the known types in a module can be handled. Such an SA has not been implemented by the author yet and therefore it still needs to be investigated whether this really is possible.

In both alternative approaches, it is no longer possible to distinguish between unsynchronized, partially synchronized, and fully synchronized broadcast requests. The distinction between unsynchronized and partially synchronized is not that big. One might drop unsynchronized requests in favor of partially synchronized ones. This still leaves the distinction among partially and fully synchronized broadcast requests. This distinction could be expressed by using port arrays of length two instead of single ports. By convention one of the ports could be used for partially and the other one for fully synchronized broadcast requests and the corresponding responses.

5.4 Data Storage

In this section we describe a server that can store values. Actually, tuples of values can be stored in one or more *tuple spaces*. A tuple space can be seen as a storage of records or tuples. A tuple has at least one entry. All tuples in a single tuple space have the same length. On of the fields in the tuples is designated as the key field and there are no two tuples with the same key in a single tuple space. This allows to look up tuples in a tuple space by using the key field as an index.

5.4.1 Data Type

The key position in a tuple space is a non-negative integer, the first position has index 0. The tuples in a tuple space have at least length one. A single server can hold several tuple spaces, to distinguish them each one has its own identifier, which is again a non-negative integer. To avoid the exchange of tuple space ids between the worker components, each tuple space will also have a name. Both names and identifiers can be used to refer to a tuple space. And there will be operations to request the identifier for a tuple space name. A tuple space is then a record consisting of its name and its identifier, the position of the key field, and the actual data in the tuple space.

```
// which of the elements is the key
type integer KeyPos (0 .. infinity);
```

```
// tuples are non-empty records of UserData
// tuples of lenght 0 are used to indicate cleared tuples
type record length ( 0 .. infinity ) of UserData Tuple;
type record of Tuple Tuples;
// tuple spaces can be addressed using names and ids
type charstring TupleSpaceName;
type integer TupleSpaceId ( -1 .. infinity );
const TupleSpaceId c_invalidId := -1;
type record TupleSpace {
  TupleSpaceName  name,
  TupleSpaceId    id,
  KeyPos          keyPos,
  Tuples          data
  }
type record of TupleSpace TupleSpaceList;
```

A tuple space server holds a mapping of names to identifiers of tuple spaces. This mapping is stored as a tuple space, with name-id pairs as tuples and where the key position corresponds to the name field. Actually, the identifier field will also be unique. For easier reference, there is a second tuple space for the reverse mapping, i.e. the same data, but where the key field is the identifier field. The names of these two tuple spaces are "Name2Id" and "Id2Name", these names and the corresponding identifiers are defined as constants. To refer to a tuple space both names and identifiers can be used.

5.4.2 Message Definition

The messages sent to a tuple space server are requests for a certain functionality. Before introducing various groups of commands, we have a short look at error handling. To indicate that a request cannot be handled correctly, it is necessary to have the possibility to indicate errors. The information is passed as values of a type `ErrorCode`:

```
type record TupleSpaceNotFound {};
type record InvalidLength { integer min }; // minimal required length
type record ErrorDuplicate {};

type union ErrorCode {
  boolean         noError,
  ErrorDuplicate  duplicateName,
  Iterator        iterator,
  InvalidLength   invalidLength,
  TupleSpaceNotFound notFound,
  file.ErrorCode  fileError
};

const ErrorCode c_wrappingLast := { iterator := c_lastTuple };
const ErrorCode c_noError := { noError := true };
const ErrorCode c_notFound := { notFound := {} };
const ErrorCode c_duplicate := { duplicateName := {} };
```

Two messages are used to retrieve the identifier of a tuple space with a given name:

```
type record IdReq { TupleSpaceName name };
type record IdRsp { TupleSpaceId id        optional,
                    ErrorCode    errorCode optional } ;
```

There are commands to create a tuple space, indicating its name and the position of the key field, to clear a tuple space completely, to write to a file or to retrieve its contents again from a file:

```
type record CreateReq { TupleSpaceName name,
                        KeyPos         keyPos };
type record CreateRsp { TupleSpaceName name,
                        TupleSpaceId   id        optional,
                        ErrorCode      errorCode optional} ;
type record ClearReq { TupleSpaceRef ref};
type record ClearRsp { ErrorCode errorCode optional };
type record SaveReq { TupleSpaceRef ref,
                      file.FileName fileName };
type record SaveRsp { ErrorCode errorCode optional };
type record LoadReq { TupleSpaceRef ref,
                      file.FileName fileName };
type record LoadRsp { ErrorCode errorCode optional };
```

There are commands to store a tuple in a tuple space, to remove a tuple with a given key, and to retrieve a tuple with a given key. On retrieval, the complete tuple is returned:

```
type record StoreReq  { TupleSpaceRef ref,
                        Tuple         tuple };
type record StoreRsp  { ErrorCode errorCode optional };
type record RemoveReq { TupleSpaceRef ref,
                        UserData      key };
type record RemoveRsp { ErrorCode errorCode optional };
type record RetrieveReq { TupleSpaceRef ref,
                          UserData      key };
type record RetrieveRsp { Tuple     tuple optional,
                          ErrorCode errorCode optional  };
```

A tuple space server can operate in two modes, a usual mode and a privileged one that allows only to store and retrieve the contents of a tuple space to and from file, respectively. There are commands to switch among the two modes.

```
// to switch among controlled and uncontrolled mode
type record ControlReq { boolean mode };
type record ControlRsp {};
```

To get access to all the tuples in a tuple space without knowing all the keys we define a datatype for iterators and commands to request an initial and a next iterator.

```
type integer Iterator ( -1 .. infinity);
const Iterator c_lastTuple := -1 ;
type record StartIteratorReq { TupleSpaceRef ref };
type record StartIteratorRsp { Iterator  iter optional,
                               ErrorCode errorCode optional  };
```

```
type record NextIteratorReq { TupleSpaceRef ref,
                              Iterator      iter };
type record NextIteratorRsp { Tuple     tuple optional,
                              Iterator  iter optional,
                              ErrorCode errorCode optional  };
```

5.4.3 Interface

To be able to refer to all possible messages in single declarations, all the different messages are used as alternatives of a single type. A tuple space server itself uses a port of type `AnyPort`. In addition it uses variables to store the sender of the latest request, and to hold the tuple spaces.

```
type component TsComp {
  port TsPort          g_pt_ts;
  var  Worker          g_v_sender;
  var  TupleSpaceList  g_v_tupleSpaces
  } //endtype component TsComp
```

5.4.4 Auxiliary Functions

To define the behaviour of a tuple space component a number of internal or auxiliary functions are used. These functions provide

- conversion among identifiers and names of tuple spaces
- actual access to the elements in a tuple space
- storage to and retrieval from file
- handling of iterators

These functions are straightforward to implement. Note that the data structure of a tuple space is quite simple. Henceforth the access function are similarly simple.

5.4.5 Behaviour

The actual behaviour of a tuple space component is defined in a very uniform way. For each of the request messages there is an altstep, that accepts a message of such a type and calls a boolean-valued function with the information passed in the message. The actual processing of the message is done in this function. The boolean return value indicates whether the function was successful. Depending on this a positive or negative reply is sent.

For each of the two modes a function is defined with an **alt**-statement in its body that contains the altsteps for all the messages that are handled in this mode and an alternative that handles all other messages. In this **alt**-statement the alternatives indicate what is the next mode by calling the corresponding function. This is shown below for one of the modes and one of the commands.

```
function f_control () runs on TsComp {
  alt {
  [] alt_controlOff() { f_idle() };
  [] alt_controlOn()  { f_control() };
  [] alt_load()       { f_control() };
  [] alt_save()       { f_control() };
  [] alt_otherCommand() {
    g_pt_ts.send(a_controlRsp_s) to g_v_sender;
    repeat }
  } // alt
} //endfunction f_control


altstep alt_id () runs on TsComp {
  var TsPrims idReq;

[] g_pt_ts.receive( a_idReq_r ) -> value idReq sender g_v_sender {
  var TupleSpaceName name := idReq.idReq.name;
  var TupleSpaceId   id := c_invalidId;
  var ErrorCode      errorCode := c_noError;

  if ( f_processIdReq(name, id, errorCode) ) {
    g_pt_ts.send( a_idRspSuccess_s( id ));
  } else {
    g_pt_ts.send( a_idRspFailure_s( errorCode ));
  }
}
} // endaltstep alt_id
```

The underlying run time system used to execute this TTCN-3 code must be able to optimize these function calls by using the fact that the functions are tail recursive. If this is not possible, the call stack will increase for each request. In this case another scheme for implementing the state machine should be used. For example, a variable can be defined to hold the state. This variable can be used in the guards of the alternatives and manipulated in the alternatives.

5.4.6 Procedure-based Interface

Although the interface of the tuple space component is defined using messages, it is closely resembling a procedure-based interface: It consists of a number of commands or request messages, for each of them there is a reply message type. These reply message types uniformly have a positive and a negative facet. Therefore it is natural to provide also a procedure-based interface – or even replace the message-based interface with the procedure-based one. In a procedure-based interface each command is mapped to a signature, a request corresponds to a procedure call, a positive reply corresponds to a procedure reply, and a negative reply corresponds to raising an exception. Because there is a response with two fields we do not use return values, but uniformly use **out** parameters to pass information in a reply.

The following code corresponds to the altstep `alt_id`, rewritten as a procedure. In this code the difference between the successful and the unsuccessful case becomes more obvious by using either a reply or by raising an exception. Note that the actual code for processing

the request is the same as for using message based communication. Processing the request is done as before by calling the function `f_processIdReq`.

```
altstep alt_procId () runs on TsCompProc {
  var TupleSpaceName name;

  [] g_pt_ts.getcall( a_idProc_r ) -> param ( name ) sender g_v_sender {
    var TupleSpaceId id := c_invalidId;
    var ErrorCode    errorCode := c_noError;
    if ( f_processIdReq(name, id, errorCode) ) {
      g_pt_ts.reply( a_idProc_s value id );
    } else {
      g_pt_ts.raise(IdProc, errorCode );
    }
  }
} // endaltstep alt_procId
```

The signature of the procedure is defined as below and shows the clear separation between the successful and the unsuccessful case.

```
signature IdProc ( in TupleSpaceName name )
return TupleSpaceId
exception ( ErrorCode ); //endsignature IdProc
```

The behaviour of a whole server component can be defined exactly the same way as it has been defined as when using message-based communication. It would be even possible to define a server that accepts both messages and procedures.

5.5 Unique Identifiers

In this section we will define a server that issues unique identifiers. Two kinds of identifiers will be be distinguished: Local identifiers that are unique for only some components and global identifiers that are unique within the whole test configuration. We will represent local identifiers as values of the type **integer** and global identifiers as lists of local identifiers:

```
type integer LocalId ( 0 .. infinity );
type record of LocalId GlobalId;

const LocalId  c_localId  := 0;
const GlobalId c_globalId := {};
```

The global ids are kept unique using a hierarchy of ids: On the highest level the local and global id are the same, more precisely, the global id is the list of length one consisting of the local id. The global id of a lower layer consists of the global id of its direct ancestor, extended with its own local id.

Consider a hierarchy of ids of height 2. Assume, that when creating the id component for level 2 the global id of its ancestor is `{ 248 }`, then the global ids of this new component all have the form `{ 248, x }`. Here `x` is a local id of this component.

An id component is initialized with the global id of its ancestor and optionally with a local id. Such an id component can be requested to issue a new id, which will be returned as

either local or global id. No assumptions can be made what will be the next id and whether ids will be monotonically increasing. There are two commands to control such a component, one to stop the component, and one to reset the local id. The latter must be used with care because it breaks the guarantee that all delivered ids are unique.

It will be possible to send single commands as well as a list of commands and the component will respond with a list of responses. This can be useful to just send several commands, but also e.g. to get the latest id and then to stop the id component without having another component interfering. To achieve this we define a union type over the types corresponding to requests and a second union type over the types corresponding to responses.

```
type record LocalIdReq      {};
type record LocalIdRsp      { LocalId id };
type record GlobalIdReq     {};
type record GlobalIdRsp     { GlobalId id };
type record StopReq         {};
type record StopRsp         {};
type record LocalIdResetReq { LocalId id }
type record LocalIdResetRsp {}
type union IdReq {
  LocalIdResetReq  localIdResetReq,
  StopReq          stopReq,
  LocalIdReq       localIdReq,
  GlobalIdReq      globalIdReq
};

type union IdRsp {
  LocalIdResetRsp  localIdResetRsp,
  StopRsp          stopRsp,
  LocalIdRsp       localIdRsp,
  GlobalIdRsp      globalIdRsp
};

type record of IdReq IdReqs;
type record of IdRsp IdRsps;
```

The component has just one port defined in addition to the general one, no further component variables or timers are defined. As usual we define an empty component type as a place holder for instances of any component type:

```
type component IdServer {
  port IdServerPort g_pt_id
  } //endtype component IdServer
```

The main function `f_id` has a global id as mandatory parameter and a local id as optional parameter. Because this function will be used to start behaviour on an id component, it must have a **runs on** clause. After the optional argument is expanded to a default value, the function `f_id_impl` is called. This function is the actual implementation of the behaviour of an id component. When receiving a single stop request, the component sends an acknowledgment and then terminates.[1] When the component receives a single request, it computes

[1] To describe this server we did not use altsteps as in the previous examples to show the reader the differences among the two approaches. Also, there are no different states of the server and therefore there is no possibility

the response, sends this back to the sender, and then waits for the next message. Similarly, if a list of requests is received, all the responses are computed and send as a single message. Actually responses are computed only to the first stop request in such a list of requests.

This component does not have different states, therefore a single function with an **alt**-statement that is either left explicitly or repeated is sufficient to describe the behaviour.

```
function f_id_impl ( in GlobalId p_v_globalId,
                     in LocalId  p_v_localId ) runs on IdServer
{
  // component state
  var GlobalId globalId := p_v_globalId;
  var LocalId  localId  := p_v_localId;

  // auxiliary variables
  var EmptyComponent s := null;
  var IdReq  idReq;
  var IdReqs idReqs;
  var IdRsp  idRsp;
  var IdRsps idRsps;

  alt {
  [] g_pt_id.receive( a_stopReqAny_r ) -> value idReq sender s
  {
    g_pt_id.send( a_stopRsp_s ) to s;
    stop
    };
  [] g_pt_id.receive( a_localIdReqAny_r ) -> value idReq sender s
  {
    idRsp := f_response( idReq, globalId, localId );
    g_pt_id.send( idRsp ) to s;
    repeat;
  };
  [] g_pt_id.receive( a_globalIdReqAny_r ) -> value idReq sender s
  {
    idRsp := f_response( idReq, globalId, localId );
    g_pt_id.send( idRsp ) to s;
    repeat;
  };
  [] g_pt_id.receive( a_resetLocalIdReqAny_r ) -> value idReq sender s
  {
    idRsp := f_response( idReq, globalId, localId );
    g_pt_id.send( idRsp );
    repeat;
  }
  [] g_pt_id.receive( IdReqs:* ) -> value idReqs sender s
  {
    idRsps := f_responses( idReqs, globalId, localId );
    g_pt_id.send( idRsps ) to s;
    if ( ischosen( idRsps[sizeof(idRsps)-1].stopRsp ) ) { stop }
    repeat; /* else */
  };
```

to reuse such altsteps in separate functions.

```
  } // alt
} // function f_id_impl
```

The requests are handled by checking what kind of request it is and then acting appropriately to determine the response. Note that this check is done here by checking the facet of the alternative by conditional expressions but not by matching when receiving the request. This approach has been used to show how code written this way looks like, actually there is no advantage in terms of readability or compactness of the code.

As part of computing a response the local id might be changed.

```
function f_response( in    IdReq    p_v_idReq,
                     in    GlobalId p_v_globalId,
                     inout LocalId  p_v_localId)
return IdRsp
{
  var IdRsp idRsp;
  if ( ischosen (p_v_idReq.stopReq) )
  {
    idRsp := { stopRsp := {} }
  } else if ( ischosen ( p_v_idReq.localIdResetReq ))
  {
    idRsp := { localIdResetRsp := {} };
    p_v_localId := p_v_idReq.localIdResetReq.id;
  } else if ( ischosen ( p_v_idReq.localIdReq ))
  {
    idRsp := { localIdRsp := { id := p_v_localId }};
    p_v_localId := next( p_v_localId );
  } else if ( ischosen ( p_v_idReq.globalIdReq ))
  {
    idRsp := { globalIdRsp := { id := append( p_v_globalId,
                                              p_v_localId ) }};
    p_v_localId := next( p_v_localId );
  };
  return ( idRsp );
}  // function f_response
```

The auxiliary functions `next` and `append` compute a new and unused id and append a local to the global id, resp.

5.6 Synchronization

Throughout the execution of a test case several parallel test component can execute concurrently. Clearly, the parallel test components need to be synchronized. Several different synchronization problems can occur:

Temporal Order One parallel test component should start a specific behaviour only after another test component has come to a certain point in its execution. On a more abstract level this means that a temporal relationship among actions on two different test components must be fulfilled.

Mutual Exclusion Several test components might need access to a shared resource. Typically this access must be mutually exclusive to maintain a consistent state of the shared resource.

Progress in Phases The test components might need to proceed in certain phases. Some phases occurring often are the *preamble* of a test case – bringing the SUT in a state in which the test could meaningful start – the *testbody* – performing the actual test – and the *postamble* – bringing the SUT back to a stable state such that a subsequent test case can be executed.

A temporal order can easily be established by sending a message from one test component to another, where the second one has to block until this message arrives. Mutual exclusion can be achieved by a component maintaining queues of components that request access to some shared resource and by granting access to one component after another. We will not consider these two problems here further, but we will have closer look at the problem how test components could be made progressing in simultaneously through subsequent phases.

In general, we assume that one of the test components controls the progress of the other test components. This single test component could be the main test component, but could also be a dedicated test component for just this purpose. We will call this test component the *synchronizing* component, independent of whether it is the main test component or a dedicated one.

In general, to start a phase, the synchronizing component sends a message to each other test component. On receipt of the message, the other test components start the behaviour for this phase. After this behaviour terminated, the test component informs the synchronizing component by sending a message and starts waiting for the next message of the synchronizing component.The synchronizing component awaits such message from all the other components, then it sends the message to start the next phase to them.

This basic behaviour is rather simple for the test components that are under control of the synchronizing component, for the synchronizing component itself it is quite similar to the behaviour of the broadcast server in the case of fully synchronized broadcast.

Two problems have not been solved in this basic setting so far. Firstly, if the behaviour of one of the test components does not terminate at all, the whole phase will not terminate and there will be no next one. Secondly, if the behaviour of one of the test components has terminated with verdict **fail**, it does not make sense to continue the behaviour on the other test cases much longer.

There is no means in TTCN-3 to preempt the behaviour of a test component under all circumstances without stopping it. By stopping the test component it is not possible to execute the postamble any more. Therefore, to cope with this problem, it is the behaviours of the test components themselves, that must guard against infinite execution, e.g. by setting a timer with a time limit for a phase. But if the timer is not checked because for example the behaviour is stuck in an infinite loop, this does not help either.

Similarly, the execution of the behaviour of a phase on a test component cannot be terminated under all circumstances. Therefore, we will present here some code, that still allows to do so in most cases, thereby assuming that the behaviour on the test components is written to support this termination.

A server component is used to control and monitor how several synchronization clients pass through a sequence of phases.

5.6.1 Synchronization Interface

The interface among the server and the clients consists of just three messages. One message type to request the clients to start the next phase, a second message type to request the clients to stop the currently ongoing phase. The third message type is used by the clients to indicate that the latest phase has ended. Besides the phase there is a field to indicate the current verdict of the clients and whether the indication was sent because the phase ended or whether it has been requested to end.

```
type record StartPhaseReq { Phase phase };
type record PhaseEndInd {
  Phase       phase,
  verdicttype localVerdict,
  boolean     stoppedOnRequest
  };

type record StopPhaseReq { Phase phase };
type union SyncPrims {
  StartPhaseReq  startPhaseReq,
  PhaseEndInd    phaseEndInd,
  StopPhaseReq   stopPhaseReq
  };
```

5.6.2 Synchronization Server

The server is started with a list of clients, a list of phases, and two boolean flags to indicate whether processing the phases should be terminated when a **inconc** or **fail** verdict has occurred. After all phases have been handled, the clients are indicated that no further phase needs to be done. The function `f_break` is used to check whether further phases should be done or whether the verdict is bad enough to avoid processing further phases.

```
function f_server ( in ClientList p_clients,
                    in PhaseList  p_phases,
                    in boolean    p_breakOnInconc,
                    in boolean    p_breakOnFail)
runs on SyncServer {
  var integer i;
  var integer phaseAmount := sizeof(p_phases);
  var boolean broken := false;

  for ( i := 0; i < phaseAmount and not broken; i := i + 1 ) {
    f_onePhase( p_clients, p_phases[i], p_breakOnInconc, p_breakOnFail );
    broken := f_break( p_breakOnInconc, p_breakOnFail, getverdict );
  }
  // indicate to all clients that all phases are done
  f_onePhase( p_clients, { phase := c_final, maxDuration := omit },
              false, false );

} // endfunction f_server
```

In a single phase, first all the clients are triggered to start their phase, then the server waits until all clients have indicated that their current phase ended. When the returned verdict is not **pass** or when a timeout has occurred for the current phase then the still running clients are stopped and the verdict of the server is set accordingly.

```
function f_onePhase( in ClientList p_clients,
                     in PhaseItem  p_phase,
                     in boolean    p_breakOnInconc,
                     in boolean    p_breakOnFail ) runs on SyncServer {
  timer t_phase;
  var SyncPrims  phaseEndInd;
  var SyncClient client;
  var integer    clientAmount := sizeof(p_clients);
  var boolean    stopped := false;
  var boolean    cleared;

  // trigger all clients
  for (var integer i := 0; i < clientAmount; i := i + 1 ) {
    g_pt_sync.send( a_startPhaseReq_s( p_phase.phase )) to p_clients[i]
    };

  // start timer if there is a timeout
  if ( ispresent(p_phase.maxDuration ) ) {
    t_phase.start( p_phase.maxDuration )
    }

  // await end of phase
  cleared := allClientsCleared( p_clients );
  alt {
    [cleared] alt_else() { };
    [] g_pt_sync.receive( a_phaseEndInd_r )
    -> value phaseEndInd sender client {
      clearClient(p_clients, client);
      cleared := allClientsCleared( p_clients );
      setverdict( phaseEndInd.phaseEndInd.localVerdict );
      if ( f_break( p_breakOnInconc, p_breakOnFail,
                    phaseEndInd.phaseEndInd.localVerdict )) {
        if ( not stopped ) {
          f_stopRemaining( p_clients, p_phase.phase );
          stopped := true;
        }
      }
      repeat;
    }
    [] t_phase.timeout {
      setverdict( fail );
      if ( not stopped ) {
        f_stopRemaining( p_clients, p_phase.phase );
        stopped := true
        }
      repeat;
    }
    [] g_pt_sync.receive {
```

```
      setverdict( fail );
    }
  } // endalt

  t_phase.stop;

} // endfunction f_onePhase
```

Waiting for indications from the clients is done until all clients have indicated the end of the current phase. Whether this has already happened is maintained in the variable `cleared`, which is recomputed by the function `allClientsCleared` whenever the indication from a client is received.

5.6.3 Synchronization Client – General Parts

On the client side a general function `f_nextPhase` can be used to indicate both the end of the previous phase to the server, as well as wait for the request to start the next phase. This function returns a boolean value indicating whether further phases should be done at all.

```
function f_nextPhase() runs on SyncClient return boolean {
  var SyncPrims startPhaseReq;

  g_v_stopped := c_continue;

  // indicate to server that phase is done
  if ( g_v_phase != c_initial ) {
    g_pt_sync.send( a_phaseEndInd_s( g_v_phase, getverdict, g_v_stopped ));
  };

  // await next phase
  alt {
    // start next phase
    [] g_pt_sync.receive( a_startPhaseReqFinal_r ) {
      g_v_syncError := e_noError;
      g_v_stopped := c_stopped;
      g_pt_sync.send( a_phaseEndInd_s( c_final, getverdict, g_v_stopped ))
      }
    [] g_pt_sync.receive( a_startPhaseReqNonFinal_r ) -> value startPhaseReq {
      g_v_phase := startPhaseReq.startPhaseReq.phase;
      g_v_syncError := e_noError;
      g_v_stopped := c_continue
      }
    // all phases done
    [] g_pt_sync.receive( a_stopPhaseReq_r( c_final ) ) {
      g_v_syncError := e_noError;
      g_v_stopped := c_stopped;
      g_pt_sync.send( a_phaseEndInd_s( c_final, getverdict, g_v_stopped ));
      }
    // catch stop request due to race conditions
    [] g_pt_sync.receive( a_stopPhaseReq_r( g_v_phase ) ) {
      g_v_syncError := e_unexpectedStopReq;
      g_v_stopped := c_stopped;
```

```
        g_pt_sync.send( a_phaseEndInd_s( g_v_phase , getverdict , g_v_stopped ));
        }
      // something wrong
      [] g_pt_sync.receive( a_stopPhaseReq_r( ? ) ) {
        g_v_syncError := e_unexpectedStopReq;
        g_v_stopped := c_stopped
        }
      // completely wrong
       [] g_pt_sync.receive {
         g_v_syncError := e_unknown;
         g_v_stopped := c_stopped
       }
    } // end alt
    return g_v_stopped;

} //endfunction f_nextPhase
```

More specifically, the function does not wait only for requests to start the next phase, but also for requests to stop the current one. This can happen if the end of phase indication of this client has not yet been processed by the server.

This function works fine to proceed through several phases in synchrony, but only as long as the client actually finishes all its phases. To support preemption of the behaviour in a phase the altstep `alt_awaitStop` can be used as a default. This altstep checks whether a stop request from the server has occurred.

```
altstep alt_awaitStop() runs on SyncClient {
  [] g_pt_sync.receive( a_stopPhaseReq_r( g_v_phase ) ) {
    g_v_stopped := c_stopped
    }
  [] g_pt_sync.receive( a_stopPhaseReq_r( ? ) ) {
    g_v_syncError := e_phaseMismatch;
    g_v_stopped := c_stopped
    }
} // endaltstep alt_awaitStop
```

The same condition can be checked explicitly by calling the function `f_checkStop`.

```
function f_checkStop () runs on SyncClient return boolean {
  alt {
    [] alt_awaitStop() {};
    [else] {}
  }
  return g_v_stopped;
} // endfunction f_checkStop
```

5.6.4 Synchronization Client – Example

In this section we will show some code fragments of a synchronization client. There are several ways how the code for such a client can be written in a reasonable way. For example, if one knows in advance that in all test cases only the phases *preamble*, *testbody*, and *postamble* will occur, then the code can be written just supporting these three phases.

Subsequently I have taken a more generic approach, in principle supporting an arbitrary number of phases. The behaviour for each phase is defined as a function operating on the component variables on which the function is executed and communicating with other components and the system under test through the ports of this component. No details about the behaviour will be given. Note also that no precise explicit definition of the type `Phase` has been given, only two distinct values for an initial and a final – empty – phase. The client is written as a loop enclosing a dispatcher function. The call of the function `f_nextPhase` in the condition of the **while** loop is blocking. These function calls are the synchronization points for the behaviours on several components. To support preemption of the behaviour of a client under control of the server the altstep `alt_awaitStop` is activated as a default. This altstep checks whether a request to stop the phase has been received on the port used for synchronization.

```
function f_client_1() runs on SyncClient {
  activate(alt_awaitStop() );

  // f_nextPhase is blocking
  while( f_nextPhase() != c_stopped ) {
    f_dispatch1(g_v_phase)
    }
} // endfunction f_client_1
```

The dispatcher function calls the appropriate function for each phase or just skips it as shown for the second phase. Note that these dispatcher functions have to be written in such an explicit way, there are constructs in TTCN-3 that would allow to pass a list of functions as parameters to a more generic dispatcher.

```
function f_dispatch1( in Phase p_phase ) {

  if      ( p_phase == c_phase1 ) { f_client1_phase1() }
  // c_phase2
  else if ( p_phase == c_phase3 ) { f_client1_phase3() }
  else if ( p_phase == c_phase4 ) { f_client1_phase4() }

} // endfunction f_dispatch1
```

In some situations the altstep activated as a default is not sufficient to stop the behaviour of component. Imagine, that there is some computationally expensive functions – if that will happen at all in a test case. This might delay the reaction on a request to stop the behaviour for quite some time. In this case the function `f_checkStop` can be used to check explicitly whether the behaviour can continue or nor. An example is given below.

```
function f_checkExplicit () runs on SyncClient {
  var integer i;
  for ( i := 0;
(i < 10000) and (f_checkStop() == c_continue);
i := i + 1 ) {
    // something computational expensive
  }
} //endfunction f_checkExplicit
```

A second problematic situation is when there are a lot of messages from the system under test or from other components that can be handled or if there is simply an **else**-branch. In this case there might always be an alternative in an **alt**-statement that can be taken and the default is never checked: the default can be seen as having lower priority than all the other alternatives. But actually it should have higher priority. This can be achieved in TTCN-3 by explicitly adding the default again as the first alternative in an **alt**-statement. Note, this has to be done explicitly for each appropriate **alt**-statement, therefore this should be used only sparingly to avoid cluttering the code. Again, an example is given below.

```
function f_highPriority () runs on SyncClient {
  alt {
    [] alt_awaitStop() {};
    [] alt_another() { //some other receiving operation
      // ...
      repeat
      };
    [else] {
      // ...
    }
  } // endalt
} // endfunction f_highPriority
```

6 Implementation Issues

The amount of code to implement solutions for the problems described in this report is about 2000 lines of TTCN-3 code. As any other piece of software this code has to be tested. About 600 lines of additional TTCN-3 code has been written so far to find errors. And actually some errors have been found and could be corrected without major changes to the code. The test code so far is limited to test the basic functionality of the various servers.

Note that in the testing code the servers are not only tested explicitly. They are also tested implicitly by using them to describe tests. As an example, the tests for the broadcast server makes use of the synchronization server.

So far, only basic functionality has been implemented in a straightforward way. No attempt has been made to use e.g. more efficient data structures to improve the performance. We considered optimization of TTCN-3 a topic of its own beyond.

7 Related Work

The classification into components with different roles has been inspired by the architecture of Erlang/OTP and its separation into applications and supervisors, [1]. Also, what has been named a tuple space here is known from Erlang as *term storage*.

The general architecture and the components themselves can be seen as examples of test patterns, i.e. solutions to recurring testing problems. There is an ongoing activity in the MTS group (Methods for Testing and Specification) of ETSI (European Telecommunication Standardization Institute) to define such test patterns.

8 Summary

In this paper we have several general server components following a general architecture such that they can be used easily in large test systems. We have explained the implementation of these components.

When defining such general components also several shortcomings of TTCN-3 have become apparent that make it cumbersome to define such generic components. Especially to support broadcasting in a simpler manner and to handle unspecified data types in a reasonable way TTCN-3 as a language should be extended.

Despite of these shortcomings we have shown that generic components can be defined in TTCN-3. We hope that by writing the examples in different styles the reader has become interested to explore new ways of writing test cases to write test suites in the most effective way for the testing problem at hand.

Acknowledgments

Colin Willcock introduced me to TTCN-3. Stephan Tobies has read and commented several versions of this report and the corresponding code. Dirk Jebing has written some of the test cases and actually found some errors. My wife has provided the moral support. Without the help of all of them this work would not have been possible.

References

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[2] Thomas Deiß. TTCN-3 for Large Systems. In Juan Garbajosa, Jorgen Boegh, Patricia Rodriguez-Dapena, and Axel Rennoch, editors, *Proc. of 3rd Workshop on Systems Testing and Validation SV04*. Fraunhofer IRB Verlag, 2004.

[3] ETSI. *Methods for Testing and Specification (MTS) The Testing and Test Control Notation version 3; Part 1 TTCN-3 Core Language; ETSI ES 201 873-1, V2.2.1*, 2003.

[4] ETSI. *Methods for Testing and Specification (MTS) The Testing and Test Control Notation version 3; Part 5 TTCN-3 Runtime Interface (TRI); ETSI ES 201 873-5, V1.1.1*, 2003.

A module optPar

```
/* ------------------------------------------------------------
 *
 * $Id: optPar.ttcn,v 1.5 2005/02/10 07:33:55 deiss Exp $
 *
 * @author Thomas Deiss
 * @remark Copyright: Nokia, 2004, 2005
 *
 * @remark DISCLAIMER: This TTCN-3 code is experimental code.
 *       Its purpose is to highlight strengths and weaknesses of
 *       TTCN-3, but it is not intended to be directly added to
 *       real test suites. The reader is strongly advised to check the code
 *       whether it fits the readers purpose and adapt it accordingly.
 *
 * @desc This module contains example definitions how to use optional
 *       parameters in TTCN-3.
 */

module optPar {

  type record OptPar_f {
    integer o1 optional,
    integer o2 optional,
    integer o3 optional
    } //endtype OptPar_f

  const OptPar_f c_defaultPar_f := {
    o1 := 3,
    o2 := 5,
    o3 := 0
  } //endconst

  function f_optPar ( in OptPar_f p_v_o ) return OptPar_f {
    var OptPar_f o := p_v_o;
    if (not ispresent(p_v_o.o1)) { o.o1 := c_defaultPar_f.o1 };
    if (not ispresent(p_v_o.o2)) { o.o2 := c_defaultPar_f.o2 };
    if (not ispresent(p_v_o.o3)) { o.o3 := c_defaultPar_f.o3 };
    return (o);
  } //endfunction f_optPar

  function f ( in integer   p_v_m1,
               in integer   p_v_m2,
               in OptPar_f  p_v_o )
  return integer
  {
    var OptPar_f o := f_optPar(p_v_o);
    return ( f_impl( p_v_m1, p_v_m2, o.o1, o.o2, o.o3) );
  } //endfunction f

  function f_impl( integer p_v_m1,
                   integer p_v_m2,
                   integer p_v_o1,
                   integer p_v_o2,
                   integer p_v_o3)
  return integer
  {
    // some code
    return 0; // dummy return value for syntactical correctness
  } // endfunction f_impl

} //endmodule optPar
```

B module userData

```
/* ------------------------------------------------------------
 *
 * $Id: userData.ttcn,v 1.7 2005/02/10 07:33:00 deiss Exp $
 *
 * @author Thomas Deiss
 * @remark Copyright: Nokia, 2004, 2005
 *
 * @remark DISCLAIMER: This TTCN-3 code is experimental code.
 *       Its purpose is to highlight strengths and weaknesses of
 *       TTCN-3, but it is not intended to be directly added to
 *       real test suites. The reader is strongly advised to check the code
 *       whether it fits the readers purpose and adapt it accordingly.
 *
 * @desc This module defines a synonym of an anytype.
 *       The synonym type can be used as an open type.
 */

module userData {

  type anytype UserData;

  type record of UserData RoUserData;

} // endmodule userData
```

C module workerSet

```
/* ------------------------------------------------------------
 *
 * $Id: workerSet.ttcn,v 1.4 2005/02/10 07:32:59 deiss Exp $
 *
 * @author Thomas Deiss
 * @remark Copyright: Nokia, 2004, 2005
 *
 * @remark DISCLAIMER: This TTCN-3 code is experimental code.
 *       Its purpose is to highlight strengths and weaknesses of
 *       TTCN-3, but it is not intended to be directly added to
 *       real test suites. The reader is strongly advised to check the code
 *       whether it fits the readers purpose and adapt it accordingly.
 *
 * @desc This module defines a simple data structure for sets of
 *       component references. A record of type is used to hold the elements
 *       of the sets. Any element in the record of that is different from
 *       null is contained at most once. The element null may be contained
 *       more than once.
*/

module workerSet {

  type component Worker {};

  type record of Worker WorkerSet;

  function createEmpty () return WorkerSet {
    var WorkerSet ws := {};
    return ws;
  }

  function isElement ( in WorkerSet p_ws,
                       in Worker    p_w )
  return boolean
```

```
35    {
        var boolean found := false;
        var integer i := 0;
        var integer l := sizeof(p_ws);

40      for ( i := 0; (i < l) and not found ; i := i + 1 ) {
          found := (p_ws[i] != null ) and ( p_ws[i] == p_w )
          };
        return found;
      } //endfunction isElement
45
      function add ( inout WorkerSet p_ws,
                     in    Worker    p_w ) {
        if ( not isElement( p_ws, p_w ) ) {
          p_ws[sizeof(p_ws)] := p_w
50        }
      } //endfunction add

      function remove ( inout WorkerSet p_ws,
                        in    Worker    p_w ) {
55      var boolean found := false;
        var integer i := 0;
        var integer l := sizeof(p_ws);

        for ( i := 0; (i < l) and not found ; i := i + 1 ) {
60        found := (p_ws[i] != null ) and ( p_ws[i] == p_w );
          if (found) { p_ws[i] := null };
        };
      } //endfunction remove

65  } //endmodule workerSet
```

D module broadcast

```
    /* ------------------------------------------------------------
     *
     * $Id: broadcast.ttcn,v 1.12 2005/02/10 07:32:41 deiss Exp $
     *
5    * @author Thomas Deiss
     * @remark Copyright: Nokia, 2004, 2005
     *
     * @remark DISCLAIMER: This TTCN-3 code is experimental code.
     *       Its purpose is to highlight strengths and weaknesses of
10   *       TTCN-3, but it is not intended to be directly added to
     *       real test suites. The reader is strongly advised to check the code
     *       whether it fits the readers purpose and adapt it accordingly.
     *
     * @desc This module defines a broadcast server.
15   */

    module broadcast {

      import from userData { type UserData };
20    import from workerSet {
        type Worker, WorkerSet;
        function all
        };

25    group g_types {

        type enumerated SyncKind {
          e_none, e_partial, e_full
        }
30    }
```

```
group g_primitives {

  group g_register {

    type record RegisterReq { Worker recipient optional };
    type record RegisterRsp {};
    type record UnRegisterReq { Worker recipient optional };
    type record UnRegisterRsp {};

  } // endgroup g_register

  group g_broadcast {

    type record DataReq { UserData data, SyncKind syncKind };
    type record DataRsp {};

  } // endgroup g_broadcast

  type union BcPrims {
    RegisterReq   registerReq,
    RegisterRsp   registerRsp,
    UnRegisterReq unregisterReq,
    UnRegisterRsp unregisterRsp,
    DataReq       dataReq,
    DataRsp       dataRsp
  }

} // endgroup g_primitives

group g_struct {

  type port BcPort message {
    inout BcPrims
    }

  type component BcComp {
    port BcPort    g_pt_bc;
    var  Worker    g_v_sender;
    var  WorkerSet g_v_workers;
  } // endtype component BcComp

} // endgroup g_struct

group g_behaviour {

  function f_start () runs on BcComp {
    g_v_sender := null;
    g_v_workers := createEmpty();
    f_idle();
  }

  function f_idle () runs on BcComp {
    var SyncKind  syncKind;
    var UserData  data;
    alt {
      [] alt_registerReq() { repeat };
      [] alt_unregisterReq() { repeat };
      [] alt_dataReq(syncKind) {
        if ( e_none == syncKind ) {
          // nothing to do
          repeat;
        } else if ( e_partial == syncKind ) {
          //acknowledge broadcast immediately
          g_pt_bc.send( { dataRsp := {} } ) to g_v_sender;
          repeat;
```

```
        } else {
          // e_full == syncKind
          // await the replies
          f_awaitReplies();
        }
      }
      [] g_pt_bc.receive { repeat }
    } //endalt
  } //endfunction f_idle

  function f_awaitReplies () runs on BcComp {
    var integer recipientAmount := sizeof(g_v_workers);

    // message has not been broadcast to sender
    if ( isElement( g_v_workers, g_v_sender ) ) {
      recipientAmount := recipientAmount - 1;
    }

    alt {
      [recipientAmount > 0] g_pt_bc.receive( { dataRsp := ? } ) {
        recipientAmount := recipientAmount - 1;
        repeat;
      }
      [recipientAmount > 0] alt_unregisterReq () {
        // allow handling of unregister
        // this will cause problems if a client unregisters instead of
        // providing a data response
        repeat;
      }
      [recipientAmount > 0] g_pt_bc.receive {
        // consume any other message, including new data requests
        repeat;
      }
      [recipientAmount <= 0] alt_else() {
        g_pt_bc.send( { dataRsp := {} } ) to g_v_sender;
        // wait for next request
        f_idle();
      }
    } //endalt
  } //endfunction f_awaitReplies

  altstep alt_registerReq () runs on BcComp {
    var BcPrims msg;
    var Worker  recipient;
    var Worker  sender_;
    [] g_pt_bc.receive( { registerReq := ? } )
    -> value msg sender sender_ {
      if ( ispresent(msg.registerReq.recipient) ) {
        recipient := msg.registerReq.recipient; }
      else {
        recipient := sender_;
      }
      add(g_v_workers, recipient);
      g_pt_bc.send( { registerRsp := {} } ) to sender_
      }
  } //endaltstep alt_registerReq

  altstep alt_unregisterReq () runs on BcComp {
    var BcPrims msg;
    var Worker  recipient;
    var Worker  sender_;
    [] g_pt_bc.receive( { unregisterReq := ? } )
    -> value msg sender sender_ {
      if ( ispresent(msg.unregisterReq.recipient) ) {
        recipient := msg.unregisterReq.recipient; }
      else {
```

```
          recipient := sender_;
        }
        remove(g_v_workers, recipient);
        g_pt_bc.send( { unregisterRsp := {} } ) to sender_;
      }
    } //endaltstep alt_unregisterReq

    altstep alt_dataReq (out SyncKind p_syncKind)
    runs on BcComp {
      var BcPrims msg;
      var integer i;
      var integer amount := sizeof(g_v_workers);

      [] g_pt_bc.receive( { dataReq := ? } ) -> value msg sender g_v_sender {
        // send the data to each registered worker
        for ( i := 0; i < amount ; i := i + 1 ) {
          if ( (g_v_workers[i] != null)
               and ( g_v_workers[i] != g_v_sender)
               and g_v_workers[i].running ) {
            // no guarantee that recipient is still running here
            g_pt_bc.send(msg) to g_v_workers[i]
            }
        } //endfor
        p_syncKind := msg.dataReq.syncKind;
      }
    } //endaltstep alt_dataReq

    altstep alt_else () {
      [else] {}
    }

  } //endgroup g_behaviour

} // endmodule broadcast
```

E module synchronization

```
/* -------------------------------------------------------------
 *
 * $Id: synchronization.ttcn,v 1.13 2005/02/10 07:32:42 deiss Exp $
 *
 * @author Thomas Deiss
 * @remark Copyright: Nokia, 2004, 2005
 *
 * @remark DISCLAIMER: This TTCN-3 code is experimental code.
 *       Its purpose is to highlight strengths and weaknesses of
 *       TTCN-3, but it is not intended to be directly added to
 *       real test suites. The reader is strongly advised to check the code
 *       whether it fits the readers purpose and adapt it accordingly.
 *
 * @desc This module contains definitions to support synchronization on both
 *       client and server side.
 */

module synchronization {

  group g_auxiliaries {

    altstep alt_else () {
      [else] {}
    } // end alt_else

    // dummy to get example code complete
    altstep alt_another () {
```

```
      [else] {}
    }

  } // endgroup g_auxiliaries

  group g_types {

    type integer Phase;

    function nextPhase ( in Phase p_phase ) return Phase {
      return p_phase + 1
      }

    const Phase c_initial := 0;
    const Phase c_final := 0;

    // dummy phases
    const Phase c_phase1 := 1;
    const Phase c_phase2 := 2;
    const Phase c_phase3 := 3;
    const Phase c_phase4 := 4;

    type record PhaseItem {
      Phase  phase,
      float  maxDuration optional
      };

    type record of PhaseItem PhaseList;

    // indicate whether phase was requested to stop
    const boolean c_stopped := true;
    const boolean c_continue := false;

    type enumerated SyncError {
      e_noError, e_phaseMismatch, e_unexpectedStopReq, e_unknown
      }

  }  // endgroup g_types

  group g_interface {

    type record StartPhaseReq { Phase phase };

    type record PhaseEndInd {
      Phase       phase,
      verdicttype localVerdict,
      boolean     stoppedOnRequest
      };

    type record StopPhaseReq { Phase phase };

    type union SyncPrims {
      StartPhaseReq  startPhaseReq,
      PhaseEndInd    phaseEndInd,
      StopPhaseReq   stopPhaseReq
      };

  } // endgroup g_interface

  group g_structure {

    type port SyncPort message {
      inout SyncPrims
      }

    type component SyncServer {
```

```
    port SyncPort  g_pt_sync;
  }

  type component SyncClient {
    port SyncPort   g_pt_sync;
    var  SyncServer g_v_syncServer;
    var  Phase      g_v_phase     := c_initial;  // current phase
    var  SyncError  g_v_syncError := e_noError;
    var  boolean    g_v_stopped   := c_continue; // request to stop current
                                                 // phase has occurred
  }

  type record of SyncClient ClientList;

} // endgroup g_structure

group g_server {

  //
  // check whether phase should be terminated
  //
  function f_break( in boolean     p_breakOnInconc,
                    in boolean     p_breakOnFail,
                    in verdicttype p_verdict ) return boolean {
    return (    ( p_verdict == inconc and p_breakOnInconc )
             or ( p_verdict == fail  and ( p_breakOnInconc or p_breakOnFail ))
                )
    } // endfunction f_break

  //
  // keep track which clients already sent a response
  //
  function clearClient( inout ClientList p_clients,
                        in    SyncClient p_client) {
    var integer i;
    var integer clientAmount := sizeof( p_clients );
    var boolean found := false;

    for (i := 0; i < clientAmount; i := i + 1) {
      if ( p_clients[i] == p_client ) {
        p_clients[i] := null;
        found := true;
      }
    } //endfor
  } // endfunction clearClient

  //
  // check wether all clients are done
  //
  function allClientsCleared( in ClientList p_clients ) return boolean {
    var integer i;
    var integer clientAmount := sizeof( p_clients );
    var boolean cleared := true;

    for ( i := 0; (i < clientAmount) and cleared; i := i + 1 ) {
      cleared := ( p_clients[i] == null );
    }
    return cleared;
  } //endfunction allClientsCleared

  //
  // request remaining clients to stop
  //
  function f_stopRemaining( inout ClientList p_clients,
                            in    Phase      p_phase) runs on SyncServer {
    var integer i;
```

```
160       var integer clientAmount := sizeof( p_clients );

          for ( i := 0; i < clientAmount; i := i + 1 ) {
            if ( p_clients[i] != null ) {
              g_pt_sync.send( a_stopPhaseReq_s( p_phase ) ) to p_clients[i];
165         }
          }

        } //endfunction f_stopRemaining

170     //
        // handle a single phase
        //
        function f_onePhase( in ClientList p_clients,
                             in PhaseItem  p_phase,
175                          in boolean    p_breakOnInconc,
                             in boolean    p_breakOnFail ) runs on SyncServer {
          timer t_phase;
          var SyncPrims  phaseEndInd;
          var SyncClient client;
180       var integer    clientAmount := sizeof(p_clients);
          var boolean    stopped := false;
          var boolean    cleared;

          // trigger all clients
185       for (var integer i := 0; i < clientAmount; i := i + 1 ) {
            g_pt_sync.send( a_startPhaseReq_s( p_phase.phase )) to p_clients[i]
            };

          // start timer if there is a timeout
190       if ( ispresent(p_phase.maxDuration ) ) {
            t_phase.start( p_phase.maxDuration )
            }

          // await end of phase
195       cleared := allClientsCleared( p_clients );
          alt {
            [cleared] alt_else() { };
            [] g_pt_sync.receive( a_phaseEndInd_r )
            -> value phaseEndInd sender client {
200           clearClient(p_clients, client);
              cleared := allClientsCleared( p_clients );
              setverdict( phaseEndInd.phaseEndInd.localVerdict );
              if ( f_break( p_breakOnInconc, p_breakOnFail,
                            phaseEndInd.phaseEndInd.localVerdict )) {
205             if ( not stopped ) {
                  f_stopRemaining( p_clients, p_phase.phase );
                  stopped := true;
                }
              }
210           repeat;
            }
            [] t_phase.timeout {
              setverdict( fail );
              if ( not stopped ) {
215             f_stopRemaining( p_clients, p_phase.phase );
                stopped := true
                }
              repeat;
            }
220         [] g_pt_sync.receive {
              setverdict( fail );
            }
          } // endalt

225       t_phase.stop;
```

```
  } // endfunction f_onePhase

  //
  // handle all phases
  //
  function f_server ( in ClientList p_clients,
                      in PhaseList  p_phases,
                      in boolean    p_breakOnInconc,
                      in boolean    p_breakOnFail)
  runs on SyncServer {
    var integer i;
    var integer phaseAmount := sizeof(p_phases);
    var boolean broken := false;

    for ( i := 0; i < phaseAmount and not broken; i := i + 1 ) {
      f_onePhase( p_clients, p_phases[i], p_breakOnInconc, p_breakOnFail );
      broken := f_break( p_breakOnInconc, p_breakOnFail, getverdict );
    }
    // indicate to all clients that all phases are done
    f_onePhase( p_clients, { phase := c_final, maxDuration := omit },
                false, false );

  } // endfunction f_server

} // endgroup g_server

group g_client {

  //
  // take a snapshot and check whether there is a stop request
  function f_checkStop () runs on SyncClient return boolean {
    alt {
      [] alt_awaitStop() {};
      [else] {}
    }
    return g_v_stopped;
  } // endfunction f_checkStop

  //
  // wait for a stop request
  //
  altstep alt_awaitStop() runs on SyncClient {
    [] g_pt_sync.receive( a_stopPhaseReq_r( g_v_phase ) ) {
      g_v_stopped := c_stopped
      }
    [] g_pt_sync.receive( a_stopPhaseReq_r( ? ) ) {
      g_v_syncError := e_phaseMismatch;
      g_v_stopped := c_stopped
      }
  } // endaltstep alt_awaitStop

  //
  // indicate to server that current phase terminated and wait for request
  // to start next phase
  //
  function f_nextPhase() runs on SyncClient return boolean {
    var SyncPrims startPhaseReq;

    g_v_stopped := c_continue;

    // indicate to server that phase is done
    if ( g_v_phase != c_initial ) {
      g_pt_sync.send( a_phaseEndInd_s( g_v_phase, getverdict, g_v_stopped ));
    };
```

```
        // await next phase
        alt {
          // start next phase
          [] g_pt_sync.receive( a_startPhaseReqFinal_r ) {
            g_v_syncError := e_noError;
            g_v_stopped := c_stopped;
            g_pt_sync.send( a_phaseEndInd_s( c_final, getverdict, g_v_stopped ))
            }
          [] g_pt_sync.receive( a_startPhaseReqNonFinal_r ) -> value startPhaseReq {
            g_v_phase := startPhaseReq.startPhaseReq.phase;
            g_v_syncError := e_noError;
            g_v_stopped := c_continue
            }
          // all phases done
          [] g_pt_sync.receive( a_stopPhaseReq_r( c_final ) ) {
            g_v_syncError := e_noError;
            g_v_stopped := c_stopped;
            g_pt_sync.send( a_phaseEndInd_s( c_final, getverdict, g_v_stopped ));
            }
          // catch stop request due to race conditions
          [] g_pt_sync.receive( a_stopPhaseReq_r( g_v_phase ) ) {
            g_v_syncError := e_unexpectedStopReq;
            g_v_stopped := c_stopped;
            g_pt_sync.send( a_phaseEndInd_s( g_v_phase, getverdict, g_v_stopped ));
            }
          // something wrong
          [] g_pt_sync.receive( a_stopPhaseReq_r( ? ) ) {
            g_v_syncError := e_unexpectedStopReq;
            g_v_stopped := c_stopped
            }
          // completely wrong
           [] g_pt_sync.receive {
             g_v_syncError := e_unknown;
             g_v_stopped := c_stopped
           }
        } // end alt
        return g_v_stopped;

      } //endfunction f_nextPhase


    } // endgroup g_client

    group g_example {

      function f_client_1() runs on SyncClient {
        activate(alt_awaitStop() );

        // f_nextPhase is blocking
        while( f_nextPhase() != c_stopped ) {
          f_dispatch1(g_v_phase)
          }
        } // endfunction f_client_1

        function f_dispatch1( in Phase p_phase ) {

          if      ( p_phase == c_phase1 ) { f_client1_phase1() }
          // c_phase2
          else if ( p_phase == c_phase3 ) { f_client1_phase3() }
          else if ( p_phase == c_phase4 ) { f_client1_phase4() }

        } // endfunction f_dispatch1

        function f_client1_phase1 () {};
        function f_client1_phase3 () {};
        function f_client1_phase4 () {};
```

```
  function f_checkExplicit () runs on SyncClient {
    var integer i;
    for ( i := 0;
          (i < 10000) and (f_checkStop() == c_continue);
          i := i + 1 ) {
      // something computational expensive
    }
  } //endfunction f_checkExplicit

  function f_highPriority () runs on SyncClient {
    alt {
      [] alt_awaitStop() {};
      [] alt_another() { //some other receiving operation
        // ...
        repeat
        };
      [else] {
        // ...
      }
    } // endalt
  } // endfunction f_highPriority

} // endgroup g_example

group g_templates {

  template SyncPrims a_stopPhaseReq_r ( in template Phase p_phase ) := {
    stopPhaseReq := { phase := p_phase }
  }

  template SyncPrims a_stopPhaseReq_s ( in Phase p_phase ) := {
    stopPhaseReq := { phase := p_phase }
  }

  template SyncPrims a_startPhaseReq_r := {
    startPhaseReq := ?
  }

  template SyncPrims a_startPhaseReqFinal_r := {
    startPhaseReq := { c_final }
  }

  template SyncPrims a_startPhaseReqNonFinal_r := {
    startPhaseReq := { ((c_final + 1) .. infinity )}
  }

  template SyncPrims a_startPhaseReq_s ( in Phase p_phase ) := {
    startPhaseReq := { phase := p_phase }
  }

  template SyncPrims a_phaseEndInd_r := {
    phaseEndInd := ?
  }

  template SyncPrims a_phaseEndInd_s ( in Phase       p_phase,
                                       in verdicttype p_verdict,
                                       in boolean     p_stopped )
  := {
    phaseEndInd := { phase            := p_phase,
                     localVerdict     := p_verdict,
                     stoppedOnRequest := p_stopped }
  }

} // endgroup g_templates
```

```
} // endmodule synchronization
```

F module tupleSpace

```
/* ----------------------------------------------------------
 *
 * $Id: tupleSpace.ttcn,v 1.22 2005/02/18 07:32:11 deiss Exp $
 *
 * @author Thomas Deiss
 * @remark Copyright: Nokia, 2004, 2005
 *
 * @remark DISCLAIMER: This TTCN-3 code is experimental code.
 *       Its purpose is to highlight strengths and weaknesses of
 *       TTCN-3, but it is not intended to be directly added to
 *       real test suites. The reader is strongly advised to check the code
 *       whether it fits the readers purpose and adapt it accordingly.
 *
 * @desc This module defines a server for tuplespaces, including all the
 *       needed datatypes
 */

module tupleSpace {

  import from file all;
  import from userData { type UserData, RoUserData };

  group g_types {

    // which of the elements is the key
    type integer KeyPos (0 .. infinity);

    // tuples are non-empty records of UserData
    // tuples of lenght 0 are used to indicate cleared tuples
    type record length ( 0 .. infinity ) of UserData Tuple;
    type record of Tuple Tuples;

    // tuple spaces can be addressed using names and ids
    type charstring TupleSpaceName;
    type integer TupleSpaceId ( -1 .. infinity );
    const TupleSpaceId c_invalidId := -1;
    type record TupleSpace {
      TupleSpaceName  name,
      TupleSpaceId    id,
      KeyPos          keyPos,
      Tuples          data
      }
    type record of TupleSpace TupleSpaceList;

  } //endgroup g_types

  group g_const {

    // names and ids of tuplespaces for internal use
    const TupleSpaceName c_name2Id_name := "Name2Id";
    const TupleSpaceName c_id2Name_name := "Id2Name";
    const TupleSpaceId c_name2Id_id := 0;
    const TupleSpaceId c_id2Name_id := 1;

  } //endgroup g_const

  // either use names or ids to address tuple spaces
  type union TupleSpaceRef { TupleSpaceName name,
                             TupleSpaceId id };

```

```
group g_error {

  type record TupleSpaceNotFound {};
  type record InvalidLength { integer min }; // minimal required length
  type record ErrorDuplicate {};

  type union ErrorCode {
    boolean         noError,
    ErrorDuplicate  duplicateName,
    Iterator        iterator,
    InvalidLength   invalidLength,
    TupleSpaceNotFound notFound,
    file.ErrorCode  fileError
  };

  const ErrorCode c_wrappingLast := { iterator := c_lastTuple };
  const ErrorCode c_noError := { noError := true };
  const ErrorCode c_notFound := { notFound := {} };
  const ErrorCode c_duplicate := { duplicateName := {} };

} //endgroup g_error

group g_commands {

  group g_control {

    // to switch among controlled and uncontrolled mode
    type record ControlReq { boolean mode };
    type record ControlRsp {};

  } //endgroup g_control

  group g_idReq {

    type record IdReq { TupleSpaceName name };
    type record IdRsp { TupleSpaceId id        optional,
                        ErrorCode    errorCode optional } ;

  } //endgroup g_idReq

  group g_create {

    type record CreateReq { TupleSpaceName name,
                            KeyPos         keyPos };
    type record CreateRsp { TupleSpaceName name,
                            TupleSpaceId   id        optional,
                            ErrorCode      errorCode optional} ;
    type record ClearReq { TupleSpaceRef ref};
    type record ClearRsp { ErrorCode errorCode optional };
    type record SaveReq { TupleSpaceRef ref,
                          file.FileName fileName };
    type record SaveRsp { ErrorCode errorCode optional };
    type record LoadReq { TupleSpaceRef ref,
                          file.FileName fileName };
    type record LoadRsp { ErrorCode errorCode optional };

  } //endgroup g_create

  group g_store {

    type record StoreReq  { TupleSpaceRef ref,
                            Tuple         tuple };
    type record StoreRsp  { ErrorCode errorCode optional };
    type record RemoveReq { TupleSpaceRef ref,
                            UserData      key };
    type record RemoveRsp { ErrorCode errorCode optional };
```

```
    type record RetrieveReq { TupleSpaceRef ref,
                              UserData      key };
    type record RetrieveRsp { Tuple     tuple optional,
                              ErrorCode errorCode optional  };

  } //endgroup g_store

  group g_iterator {

    type integer Iterator ( -1 .. infinity);
    const Iterator c_lastTuple := -1 ;
    type record StartIteratorReq { TupleSpaceRef ref };
    type record StartIteratorRsp { Iterator  iter optional,
                                   ErrorCode errorCode optional  };
    type record NextIteratorReq { TupleSpaceRef ref,
                                  Iterator      iter };
    type record NextIteratorRsp { Tuple     tuple optional,
                                  Iterator  iter optional,
                                  ErrorCode errorCode optional  };

  } //endgroup g_iterator

  type union TsPrims {
    ControlReq controlReq,
    ControlRsp controlRsp,
    IdReq idReq,
    IdRsp idRsp,
    CreateReq createReq,
    CreateRsp createRsp,
    ClearReq  clearReq,
    ClearRsp  clearRsp,
    SaveReq   saveReq,
    SaveRsp   saveRsp,
    LoadReq   loadReq,
    LoadRsp   loadRsp,
    StoreReq  storeReq,
    StoreRsp  storeRsp,
    RemoveReq removeReq,
    RemoveRsp removeRsp,
    RetrieveReq retrieveReq,
    RetrieveRsp retrieveRsp,
    StartIteratorReq startIteratorReq,
    StartIteratorRsp startIteratorRsp,
    NextIteratorReq  nextIteratorReq,
    NextIteratorRsp  nextIteratorRsp
  }; // type union TsPrims
} //endgroup g_commands

group g_structure {

  // specific port type
  type port TsPort message { inout TsPrims };

  type component Worker {};

  // the server component type
  type component TsComp {
    port TsPort          g_pt_ts;
    var  Worker          g_v_sender;
    var  TupleSpaceList  g_v_tupleSpaces
    } //endtype component TsComp

} //endgroup g_structure

group g_auxiliary {
```

```
// conversion among tuples and UserData
function data2Tuple ( in UserData p_v_data ) return Tuple
{
  var RoUserData d := p_v_data.RoUserData;
  var Tuple t := d;

  return t;
};

function tuple2Data ( in Tuple p_v_tuple ) return UserData
{
  var RoUserData d := p_v_tuple;

  return { RoUserData := d };
};

// initial tuplespace,
// contains just the mappings from names to ids and vice versa
function initTupleSpaces () return TupleSpaceList {
  var Tuple name2id := { { charstring := c_name2Id_name},
                         { integer := c_name2Id_id }};
  var Tuple id2name := { { charstring := c_id2Name_name },
                         { integer := c_id2Name_id } };
  var Tuples data := { name2id, id2name };
  var TupleSpace tsName2Id := { name := c_name2Id_name,
                                id   := c_name2Id_id,
                                keyPos := 0,
                                data := data };
  var TupleSpace tsId2Name := { name := c_id2Name_name,
                                id   := c_id2Name_id,
                                keyPos := 1,
                                data := data };
  return { tsName2Id, tsId2Name };
}


group g_id {

  // determine the id from a reference,
  // return an invalidId if there is no corresponding tuple space
  function refToId ( in TupleSpaceRef p_v_ref ) runs on TsComp
  return TupleSpaceId {
    if (ischosen(p_v_ref.name)) {
      return nameToId(p_v_ref.name)
      } else if (idExists(p_v_ref.id)) {
        return p_v_ref.id;
        } else {
          return c_invalidId;
        }
  }

  // determine the id from a name
  // return an invalidId if there is no corresponding tuple space
  // use internal tuple space for the mapping name -> id
  function nameToId ( in TupleSpaceName p_v_name ) runs on TsComp
  return TupleSpaceId {
    var TupleSpace name2Id := g_v_tupleSpaces[c_name2Id_id];
    var Tuple tuple := retrieve( {charstring := p_v_name }, name2Id );
    if ( sizeof(tuple) == 0 ) {
      return c_invalidId
      } else {
        return tuple[1].integer;
      }
  };

  // check whether a tuple space with a given id exists
```

```
    // use internal tuplespace for the mapping id -> name
    function idExists ( in TupleSpaceId p_v_id ) runs on TsComp
    return boolean {
      var TupleSpace id2Name := g_v_tupleSpaces[c_id2Name_id];
      var Tuple tuple := retrieve( {integer := p_v_id}, id2Name );
      return (sizeof(tuple) != 0);
    };

    // get next free id
    // the next free id is the lowest one for which there is no name
    function getFreeId () runs on TsComp return TupleSpaceId
    {
      var TupleSpaceId id := c_id2Name_id;
      while ( idExists( id ) ) {
        id := id + 1;
      }
      return id;
    }

    // get the index of the first used tuple
    // the first free tuple is the one which is not empty
    function getFirstTuple ( in TupleSpace p_v_tupleSpace ) return Iterator {
      var Iterator iter := c_lastTuple;
      var integer i := 0;
      var integer l := sizeof(p_v_tupleSpace.data);

      while ( (i < l) and (iter == c_lastTuple) ) {
        if ( sizeof(p_v_tupleSpace) !=0 ) { iter := i; }
        i := i + 1;
      }

      return iter;
    }

    // get the index of the next used tuple
    function getNextTuple ( in TupleSpace p_v_tupleSpace,
                            in Iterator   p_v_iter )
    return Iterator {
      var Iterator iter := c_lastTuple;
      var integer i := p_v_iter;
      var integer l := sizeof(p_v_tupleSpace.data);

      while ( i < l and iter == c_lastTuple ) {
        if ( sizeof(p_v_tupleSpace) != 0 ) { iter := i; }
        i := i + 1;
      }

      return iter;
    }

  } // endgroup g_id

  group g_tupleaccess {

    function clearTupleSpace ( inout TupleSpace p_v_tupleSpace) {
      p_v_tupleSpace := {
        id := c_invalidId,
        data := {}
      }
    }

    function retrieve ( in UserData p_v_key,
                        in TupleSpace p_v_tupleSpace)
    return Tuple {
      var boolean found := false;
      var Tuple tmpTuple;
```

```
325         var Tuple tuple := {};  // empty tuple indicates nothing found

            for (var integer i := 0;
                 (not found) and (i < sizeof(p_v_tupleSpace.data));
                 i := i + 1 ) {
330           tmpTuple := p_v_tupleSpace.data[i];
              if (sizeof(tmpTuple) != 0) {
                // the tuple contains data;
                if (tmpTuple[p_v_tupleSpace.keyPos] == p_v_key) {
                  tuple := tmpTuple;
335               found := true
                  }
              }
            }
            return tuple;
340       }

          function remove ( in UserData p_v_key,
                            inout TupleSpace p_v_tupleSpace) {
            var boolean found := false;
345         var Tuple tmpTuple;

            for (var integer i := 0;
                 (not found) and (i < sizeof(p_v_tupleSpace.data));
                 i := i +1 ) {
350           tmpTuple := p_v_tupleSpace.data[i];
              if (sizeof(tmpTuple) != 0) {
                // the tuple contains data;
                if (tmpTuple[p_v_tupleSpace.keyPos] == p_v_key) {
                  tmpTuple := {};
355               p_v_tupleSpace.data[i] := tmpTuple;
                  found := true
                  }
              }
            }
360         return;
          }

          function contains ( in UserData p_v_key,
                              in TupleSpace p_v_tupleSpace)
365       return integer {
            var boolean found := false;
            var integer index := -1;
            var Tuple tmpTuple;

370         for (var integer i := 0;
                 (not found) and (i < sizeof(p_v_tupleSpace.data));
                 i := i +1 ) {
              tmpTuple := p_v_tupleSpace.data[i];
              if (sizeof(tmpTuple) != 0) {
375             // the tuple contains data;
                if (tmpTuple[p_v_tupleSpace.keyPos] == p_v_key) {
                  index := i;
                  found := true;
                }
380           }
            }
            return index;
          }

385       function store ( in Tuple p_v_tuple,
                           inout TupleSpace p_v_tupleSpace) {
            var boolean found := false;
            var integer index;
            var Tuple   tmpTuple;
390
```

```
        index := contains(p_v_tuple[p_v_tupleSpace.keyPos],
                          p_v_tupleSpace);
        if (index < 0) {
          var Tuples tuples := p_v_tupleSpace.data;
          for ( var integer i := 0;
                (not found) and (i < sizeof(tuples));
                i := i + 1 ) {
            tmpTuple := tuples[i];
            if (sizeof(tmpTuple) == 0) {
              tuples[i] := p_v_tuple;
              p_v_tupleSpace.data := tuples;
              found := true;
            }
          } // for
          if (not found) {
            tuples[sizeof(tuples)] := p_v_tuple;
            p_v_tupleSpace.data := tuples;
          }
        } else {
          p_v_tupleSpace.data[index] := p_v_tuple;
          }
        return;
      }

    } //endgroup g_tupleAccess

  } //endgroup g_auxiliary

  group g_server {
    // the behaviour of a tuple space component

    // the control component has two modes:
    // control: only some commands are available, other ones are ignored
    // idle:    responding requests
    // the control state can be used to temporarily make a tuple space
    // temporarily unavailable

    function f_start (in boolean p_v_controlMode) runs on TsComp {
      g_v_sender := null;
      g_v_tupleSpaces := initTupleSpaces();
      if (p_v_controlMode) {
        f_control();
      } else {
        f_idle();
      }
    }

    function f_control () runs on TsComp {
      alt {
      [] alt_controlOff() { f_idle() };
      [] alt_controlOn()  { f_control() };
      [] alt_load()       { f_control() };
      [] alt_save()       { f_control() };
      [] alt_otherCommand() {
        g_pt_ts.send(a_controlRsp_s) to g_v_sender;
        repeat }
      } // alt
    } //endfunction f_control

    function f_idle() runs on TsComp {
      alt {
      [] alt_controlOn()     { f_control() };
      [] alt_controlOff()    { f_idle() };
      [] alt_id()            { f_idle() };
      [] alt_create()        { f_idle() };
      [] alt_clear()         { f_idle() };
```

```
    [] alt_load()          { f_idle() };
    [] alt_save()          { f_idle() };
    [] alt_store()         { f_idle() };
    [] alt_remove()        { f_idle() };
    [] alt_retrieve()      { f_idle() };
    [] alt_startIterator() { f_idle() };
    [] alt_nextIterator()  { f_idle() };
    [] alt_otherCommand()  { repeat }
    } // alt
  } //endfunction f_idle

  group g_commandHandling {

    // on any command just store the sender
    altstep alt_otherCommand () runs on TsComp {
    [] g_pt_ts.receive -> sender g_v_sender {}
    }

    // on command send an acknowledge
    altstep alt_controlOff () runs on TsComp {
    [] g_pt_ts.receive( a_controlReqOff_r ) -> sender g_v_sender {
      g_pt_ts.send(a_controlRsp_s) to g_v_sender;
      }
    }

    // on command send an acknowledge
    altstep alt_controlOn () runs on TsComp {
    [] g_pt_ts.receive( a_controlReqOn_r ) -> sender g_v_sender {
      g_pt_ts.send(a_controlRsp_s) to g_v_sender;
    }
    }

    // on command retrieve id and send this as reply
    // if no id can be found reply with an error code
    altstep alt_id () runs on TsComp {
      var TsPrims idReq;

    [] g_pt_ts.receive( a_idReq_r ) -> value idReq sender g_v_sender {
      var TupleSpaceName name := idReq.idReq.name;
      var TupleSpaceId   id := c_invalidId;
      var ErrorCode      errorCode := c_noError;

      if ( f_processIdReq(name, id, errorCode) ) {
        g_pt_ts.send( a_idRspSuccess_s( id ));
      } else {
        g_pt_ts.send( a_idRspFailure_s( errorCode ));
      }
    }
    } // endaltstep alt_id

    // determine id,
    // return true if an id can be found
    function f_processIdReq( in    TupleSpaceName p_v_name,
                             inout TupleSpaceId   p_v_id,
                             inout ErrorCode      p_v_errorCode)
    runs on TsComp return boolean
    {
      p_v_id := nameToId(p_v_name);
      if ( p_v_id == c_invalidId) {
        p_v_errorCode := c_notFound;
      }
      return (p_v_errorCode == c_noError);
    }

    altstep alt_create () runs on TsComp {
      var TsPrims createReq;
```

```
  [] g_pt_ts.receive( a_createReq_r ) -> value createReq sender g_v_sender {
    var TupleSpaceName name := createReq.createReq.name;
    var KeyPos         keyPos := createReq.createReq.keyPos;
    var TupleSpaceId id := c_invalidId;
    var ErrorCode    errorCode := c_noError;
    if ( f_processCreateReq( name, keyPos, id, errorCode ) ) {
      g_pt_ts.send( a_createRspSuccess_s( name, id ));
    } else {
      g_pt_ts.send( a_createRspFailure_s( name, errorCode )) ;
    }
  }
  }

  function f_processCreateReq ( in    TupleSpaceName p_v_name,
                                in    KeyPos         p_v_keyPos,
                                inout TupleSpaceId   p_v_id,
                                inout ErrorCode      p_v_errorCode )
  runs on TsComp return boolean {
    p_v_id := nameToId( p_v_name );
    if ( p_v_id == c_invalidId ) {
      // no such tuple space exists, hence a new one can be created
      var Tuple nameToIdTuple;
      var Tuple idToNameTuple;
      var TupleSpace tmpTupleSpace;

      p_v_id := getFreeId();
      // update mappings between names and ids
      nameToIdTuple := { {charstring := p_v_name}, { integer := p_v_id} };
      idToNameTuple := { {integer := p_v_id}, {charstring := p_v_name} };
      store(nameToIdTuple, g_v_tupleSpaces[c_name2Id_id]);
      store(idToNameTuple, g_v_tupleSpaces[c_id2Name_id]);
      // store a new tuple space
      g_v_tupleSpaces[p_v_id] := { name := p_v_name,
                                   id   := p_v_id,
                                   keyPos := p_v_keyPos,
                                   data := {} };
    } else {
      // such a tuple space exists already
      p_v_errorCode := c_duplicate;
    }
    return ( p_v_errorCode == c_noError );
  }

  // clear a tuple space,
  // i.e. remove its data and remove the mapping between the name and the id
  altstep alt_clear () runs on TsComp {
    var TsPrims clearReq;

  [] g_pt_ts.receive( a_clearReq_r ) -> value clearReq sender g_v_sender {
    var TupleSpaceRef ref := clearReq.clearReq.ref;
    var ErrorCode errorCode := c_noError;
    if ( f_processClearReq( ref, errorCode ) ) {
      g_pt_ts.send( a_clearRspSuccess_s );
    } else {
      g_pt_ts.send( a_clearRspFailure_s( errorCode ) );
    }
  }
  }

  function f_processClearReq ( in    TupleSpaceRef p_v_ref,
                               inout ErrorCode     p_v_errorCode )
  runs on TsComp return boolean
  {
    var TupleSpaceId id := refToId( p_v_ref );
    var TupleSpaceName name;
```

```
    if ( id == c_invalidId ) {
      p_v_errorCode := c_notFound;
    } else {
      var Tuple nameTuple;
      nameTuple := retrieve( {integer := id}, g_v_tupleSpaces[c_id2Name_id]);
      name := nameTuple[1].charstring;
      clearTupleSpace( g_v_tupleSpaces[id]);
      remove( {integer := id}, g_v_tupleSpaces[c_id2Name_id] );
      remove( {charstring := name}, g_v_tupleSpaces[c_name2Id_id]);
    }
    return ( p_v_errorCode == c_notFound );
  }

  // save a tuplespace to a file
  // In the stored file,
  // the first element is the tuplespacename,
  // the second element is the keyposition,
  // the remaining elements are the tuples
  altstep alt_save () runs on TsComp {
    var TsPrims saveReq;

  [] g_pt_ts.receive( a_saveReq_r ) -> value saveReq sender g_v_sender {
    var TupleSpaceRef ref := saveReq.saveReq.ref;
    var file.FileName fileName := saveReq.saveReq.fileName;
    var ErrorCode    errorCode := c_noError;
    if ( f_processSaveReq( ref, fileName, errorCode ) ) {
      g_pt_ts.send( a_saveRspSuccess_s );
    } else {
      g_pt_ts.send( a_saveRspFailure_s( errorCode ) ) ;
    }
  }
  }

  function f_processSaveReq ( in TupleSpaceRef p_v_ref,
                              in file.FileName p_v_fileName,
                              inout ErrorCode p_v_errorCode )
  runs on TsComp return boolean
  {
    var TupleSpaceId id;
    var file.FileDescriptor fd;
    var file.ErrorCode fec;

    id := refToId( p_v_ref );
    if ( id == c_invalidId ) {
      p_v_errorCode := c_notFound;
      return false;
    }

    // open the file for writing
    fec := file.open(fd, p_v_fileName, c_write);
    if ( fec == file.c_noError ) {
      p_v_errorCode := { fileError := fec };
      return false;
    }

    // write the contents
    f_writeContents( id, fd, p_v_errorCode);
    // close the file
    file.close(fd);
    return ( p_v_errorCode == c_noError );
  }

  function f_writeContents ( in    TupleSpaceId p_v_id,
                             in    file.FileDescriptor p_v_fd,
                             inout ErrorCode p_v_errorCode )
  runs on TsComp return boolean
```

```
{
  var TupleSpace ts := g_v_tupleSpaces[p_v_id];
  var integer l := sizeof(ts.data);
  var integer i;
  var file.ErrorCode fec;

  fec := file.write( p_v_fd, {charstring := ts.name} );
  if ( fec != file.c_noError ) {
    p_v_errorCode := { fileError := fec };
    return false;
  }
  fec := file.write( p_v_fd, { integer := ts.keyPos } );
  if ( fec != file.c_noError ) {
    p_v_errorCode := { fileError := fec };
    return false;
  }

  // write tuples, break if an error occurred
  for ( i := 0; i < l and p_v_errorCode == c_noError; i := i + 1) {
    // write only non-empty tuples
    if ( sizeof(ts.data[i]) != 0 ) {
      fec := file.write(p_v_fd, tuple2Data(ts.data[i]));
      if ( fec != file.c_noError ) {
        p_v_errorCode := { fileError := fec };
      }
    }
  } // for
  return ( p_v_errorCode == c_noError );
}

// load a tuplespace from a file
altstep alt_load () runs on TsComp {
  var TsPrims loadReq;

[] g_pt_ts.receive( a_loadReq_r ) -> value loadReq sender g_v_sender {
  var TupleSpaceRef ref := loadReq.loadReq.ref;
  var file.FileName fileName := loadReq.loadReq.fileName;
  var ErrorCode    errorCode := c_noError;
  if ( f_processLoadReq( ref, fileName, errorCode ) ) {
    g_pt_ts.send( a_loadRspSuccess_s );
  } else {
    g_pt_ts.send( a_loadRspFailure_s( errorCode ) );
  }
}
}

function f_processLoadReq ( in TupleSpaceRef p_v_ref,
                            in file.FileName p_v_fileName,
                            inout ErrorCode p_v_errorCode )
runs on TsComp return boolean
{
  var TupleSpaceId id;
  var TupleSpace ts;
  var file.FileDescriptor fd;
  var file.ErrorCode fec;
  var Tuple nameToIdTuple;
  var Tuple idToNameTuple;

  id := refToId( p_v_ref );
  if ( id != c_invalidId ) {
    p_v_errorCode := c_duplicate;
    return false;
  }
  ts := g_v_tupleSpaces[id];

  // open the file for reading
```

```
    fec := file.open(fd, p_v_fileName, c_read);
    if ( fec == file.c_noError ) {
      p_v_errorCode := { fileError := fec };
      return false;
    }

    // read the contents
    f_loadContents( ts, fd, p_v_errorCode);
    // close the file
    file.close(fd);

    if ( p_v_errorCode == c_noError) {
      id := getFreeId();
      // update mappings between names and ids
      nameToIdTuple := { {charstring := ts.name}, {integer := id} };
      idToNameTuple := { {integer := id}, {charstring := ts.name} };
      store(nameToIdTuple, g_v_tupleSpaces[c_name2Id_id]);
      store(idToNameTuple, g_v_tupleSpaces[c_id2Name_id]);
      // store a new tuple space
      ts.id := id;
      g_v_tupleSpaces[id] := ts;
    }
    return ( p_v_errorCode == c_noError );
  }

  function f_loadContents ( inout TupleSpace p_v_ts,
                            in    file.FileDescriptor p_v_fd,
                            inout ErrorCode p_v_errorCode )
  runs on TsComp return boolean
  {
    var integer i := 0;
    var file.ErrorCode fec;
    var Tuple data;
    var UserData ud;

    fec := file.read_( p_v_fd, ud);
    p_v_ts.name := ud.charstring;
    if ( fec == file.c_noError ) {
      p_v_errorCode := { fileError := fec };
      return false;
    }

    fec := file.read_( p_v_fd, ud);
    p_v_ts.keyPos := ud.integer;
    if ( fec == file.c_noError ) {
      p_v_errorCode := { fileError := fec };
      return false;
    }

    // read tuples, break if an error occurred
    while ((not file.eof(p_v_fd)) and (p_v_errorCode == c_noError)) {
      fec := file.read_(p_v_fd, ud);
      p_v_ts.data[i] := data2Tuple(ud);
      if ( fec != file.c_noError ) {
        p_v_errorCode := { fileError := fec };
      }
    } // while
    return ( p_v_errorCode == c_noError );
  }

  // store a tuple,
  // overwrite an existing tuple with the same key
  altstep alt_store () runs on TsComp {
    var TsPrims storeReq;

  [] g_pt_ts.receive( a_storeReq_r ) -> value storeReq sender g_v_sender {
```

```
  var TupleSpaceRef ref := storeReq.storeReq.ref;
  var Tuple tuple := storeReq.storeReq.tuple;
  var ErrorCode    errorCode := c_noError;
  if ( f_processStoreReq( ref, tuple, errorCode ) ) {
    g_pt_ts.send( a_storeRspSuccess_s );
  } else {
    g_pt_ts.send( a_storeRspFailure_s( errorCode ) ) ;
  }
}
}

function f_processStoreReq ( in TupleSpaceRef p_v_ref,
                             in Tuple p_v_tuple,
                             inout ErrorCode p_v_errorCode )
runs on TsComp return boolean
{
  var TupleSpaceId id;
  var TupleSpace ts;

  id := refToId( p_v_ref );
  if ( id == c_invalidId ) {
    p_v_errorCode := c_notFound;
    return false;
  }

  ts := g_v_tupleSpaces[id];
  // check that the tuple is large enough to contain the key position
  if ( ts.keyPos > sizeof(p_v_tuple)) {
    p_v_errorCode := { invalidLength := { ts.keyPos } };
    return false;
  } else {
    store(p_v_tuple, ts);
    g_v_tupleSpaces[id] := ts;
    return ( p_v_errorCode == c_noError );
  }
}

// remove a tuple,
// overwrite an existing tuple with the same key
altstep alt_remove () runs on TsComp {
  var TsPrims removeReq;

[] g_pt_ts.receive( a_removeReq_r ) -> value removeReq sender g_v_sender {
  var TupleSpaceRef ref := removeReq.removeReq.ref;
  var UserData      key := removeReq.removeReq.key;
  var ErrorCode    errorCode := c_noError;
  if ( f_processRemoveReq( ref, key, errorCode ) ) {
    g_pt_ts.send( a_removeRspSuccess_s );
  } else {
    g_pt_ts.send( a_removeRspFailure_s( errorCode ) );
  }
}
}

function f_processRemoveReq ( in TupleSpaceRef p_v_ref,
                              in UserData p_v_key,
                              inout ErrorCode p_v_errorCode )
runs on TsComp return boolean
{
  var TupleSpaceId id;
  var TupleSpace ts;

  id := refToId( p_v_ref );
  if ( id == c_invalidId ) {
    p_v_errorCode := c_notFound;
    return false;
```

```
  }

  ts := g_v_tupleSpaces[id];
  remove( p_v_key, ts );
  g_v_tupleSpaces[id] := ts;
  return ( p_v_errorCode == c_noError );
}

// retrieve a tuple,
altstep alt_retrieve () runs on TsComp {
  var TsPrims retrieveReq;

[] g_pt_ts.receive( a_retrieveReq_r )
          -> value retrieveReq sender g_v_sender {
  var TupleSpaceRef ref := retrieveReq.retrieveReq.ref;
  var UserData      key := retrieveReq.retrieveReq.key;
  var Tuple         tuple;
  var ErrorCode     errorCode := c_noError;
  if ( f_processRetrieveReq( ref, key, tuple, errorCode ) ) {
    g_pt_ts.send( a_retrieveRspSuccess_s(tuple) );
  } else {
    g_pt_ts.send( a_retrieveRspFailure_s( errorCode ) );
  }
}
}

function f_processRetrieveReq ( in TupleSpaceRef p_v_ref,
                                 in UserData p_v_key,
                                 out Tuple p_v_tuple,
                                 inout ErrorCode p_v_errorCode )
runs on TsComp return boolean
{
  var TupleSpaceId id;
  var TupleSpace ts;

  id := refToId( p_v_ref );
  if ( id == c_invalidId ) {
    p_v_errorCode := c_notFound;
    return false;
  }

  ts := g_v_tupleSpaces[id];
  p_v_tuple := retrieve( p_v_key, ts );
  return ( p_v_errorCode == c_noError );
}

// get a new iterator
altstep alt_startIterator () runs on TsComp {
  var TsPrims startIteratorReq;

[] g_pt_ts.receive( a_startIteratorReq_r )
          -> value startIteratorReq sender g_v_sender {
  var TupleSpaceRef ref := startIteratorReq.startIteratorReq.ref;
  var Iterator      iter;
  var ErrorCode     errorCode := c_noError;
  if ( f_processStartIteratorReq( ref, iter, errorCode ) ) {
    g_pt_ts.send( a_startIteratorRspSuccess_s( iter ));
  } else {
    g_pt_ts.send( a_startIteratorRspFailure_s( errorCode )) ;
  }
}
}

function f_processStartIteratorReq ( in TupleSpaceRef p_v_ref,
                                     out Iterator p_v_iter,
                                     inout ErrorCode p_v_errorCode )
```

```
    runs on TsComp return boolean
    {
      var TupleSpaceId id;
      var TupleSpace ts;

      id := refToId( p_v_ref );
      if ( id == c_invalidId ) {
        p_v_errorCode := c_notFound;
        return false;
      }

      ts := g_v_tupleSpaces[id];
      p_v_iter := getFirstTuple( ts );
      return ( p_v_errorCode == c_noError );
    }

    // get a next iterator
    altstep alt_nextIterator () runs on TsComp {
      var TsPrims nextIteratorReq;

    [] g_pt_ts.receive( a_nextIteratorReq_r )
                -> value nextIteratorReq sender g_v_sender {
      var TupleSpaceRef ref := nextIteratorReq.nextIteratorReq.ref;
      var Iterator      iter := nextIteratorReq.nextIteratorReq.iter;
      var Tuple         tuple;
      var ErrorCode     errorCode := c_noError;
      if ( f_processNextIteratorReq( ref, iter, tuple, errorCode ) ) {
        g_pt_ts.send( a_nextIteratorRspSuccess_s( tuple, iter ));
      } else {
        g_pt_ts.send( a_nextIteratorRspFailure_s( errorCode )) ;
      }
    }
    }

    function f_processNextIteratorReq ( in    TupleSpaceRef p_v_ref,
                                        inout Iterator      p_v_iter,
                                        out   Tuple         p_v_tuple,
                                        inout ErrorCode     p_v_errorCode )
    runs on TsComp return boolean
    {
      var TupleSpaceId id;
      var TupleSpace ts;

      id := refToId( p_v_ref );
      if ( id == c_invalidId ) {
        p_v_errorCode := c_notFound;
        return false;
      }

      if ( p_v_iter != c_lastTuple ) {
        ts := g_v_tupleSpaces[id];
        p_v_iter := getNextTuple( ts, p_v_iter );
        p_v_tuple := ts.data[p_v_iter];
      } else {
        // if iter refers to the last element this is an error
        p_v_errorCode := c_wrappingLast;
      }
      return ( p_v_errorCode == c_noError );
    }

   } // endgroup g_commandHandling

  group g_template {

    template TsPrims a_controlRsp_s
    := { controlRsp := {} };
```

```
985
        template TsPrims a_controlReqOff_r
        := { controlReq := { mode := true } };
        template TsPrims a_controlReqOn_r
        := { controlReq := { mode := false } };
990
        template TsPrims a_idRspFailure_s ( in template ErrorCode p_errorCode )
        := { idRsp := { id := omit , errorCode := p_errorCode } };
        template TsPrims a_idRspSuccess_s ( in template TupleSpaceId p_id )
        := { idRsp := { id := p_id , errorCode := omit } };
995     template TsPrims a_idReq_r := { idReq := ? };

        template TsPrims a_clearRspFailure_s ( in ErrorCode p_errorCode )
        := { clearRsp := { errorCode := p_errorCode } };
        template TsPrims a_clearRspSuccess_s
1000    := { clearRsp := { errorCode := omit } };
        template TsPrims a_clearReq_r := { clearReq := ? };

        template TsPrims a_createRspSuccess_s ( in TupleSpaceName p_name,
                                                in TupleSpaceId   p_id )
1005    := { createRsp :=
             {
               name := p_name,
               id := p_id,
               errorCode := omit
1010         }
        }

        template TsPrims a_createRspFailure_s ( in TupleSpaceName p_name,
                                                in ErrorCode      p_errorCode )
1015    := { createRsp :=
             {
               name := p_name,
               id := omit,
               errorCode := p_errorCode
1020         }
        }
        template TsPrims a_createReq_r := { createReq := ? };

        template TsPrims a_saveRspSuccess_s
1025    := { saveRsp := { errorCode := omit} };
        template TsPrims a_saveRspFailure_s ( in ErrorCode p_errorCode )
        := { saveRsp := { errorCode := p_errorCode } };
        template TsPrims a_saveReq_r := { saveReq := ? };

1030    template TsPrims a_loadRspSuccess_s
        := { loadRsp := { errorCode := omit} };
        template TsPrims a_loadRspFailure_s ( in ErrorCode p_errorCode )
        := { loadRsp := { errorCode := p_errorCode } };
        template TsPrims a_loadReq_r := { loadReq := ? };
1035
        template TsPrims a_storeRspSuccess_s
        := { storeRsp := { errorCode := omit} };
        template TsPrims a_storeRspFailure_s ( in ErrorCode p_errorCode )
        := { storeRsp := { errorCode := p_errorCode } };
1040    template TsPrims a_storeReq_r := { storeReq := ? };

        template TsPrims a_removeRspSuccess_s
        := { removeRsp := { errorCode := omit} };
        template TsPrims a_removeRspFailure_s ( in ErrorCode p_errorCode )
1045    := { removeRsp := { errorCode := p_errorCode } };
        template TsPrims a_removeReq_r := { removeReq := ? };

        template TsPrims a_retrieveRspSuccess_s ( in Tuple p_tuple )
        := { retrieveRsp :=
1050         { tuple := p_tuple,
```

```
            errorCode := omit}
          };
      template TsPrims a_retrieveRspFailure_s ( in ErrorCode p_errorCode )
      := { retrieveRsp :=
           { tuple := omit,
             errorCode := p_errorCode }
          };
      template TsPrims a_retrieveReq_r := { retrieveReq := ? };

      template TsPrims a_startIteratorRspSuccess_s
      ( in Iterator p_iter )
      := { startIteratorRsp :=
           { iter := p_iter,
             errorCode := omit }
      };
      template TsPrims a_startIteratorRspFailure_s
      ( in ErrorCode p_errorCode )
      := { startIteratorRsp :=
           { iter := omit,
             errorCode := p_errorCode }
      };
      template TsPrims a_startIteratorReq_r
      := { startIteratorReq := ? };

      template TsPrims a_nextIteratorRspSuccess_s
      ( in Tuple p_tuple, in Iterator p_iter )
      := { nextIteratorRsp :=
           { tuple := p_tuple,
             iter := p_iter,
             errorCode := omit}
      };
      template TsPrims a_nextIteratorRspFailure_s
      ( in ErrorCode p_errorCode )
      := { nextIteratorRsp :=
           { tuple := omit,
             iter := omit,
             errorCode := p_errorCode }
      };
      template TsPrims a_nextIteratorReq_r := { nextIteratorReq := ? };

    } //endgroup g_template

  } //endgroup g_server

  group g_procedure {

    group g_structureProc {

      type port TsPortProc procedure {
        inout CreateProc;
        inout IdProc
        }

      type component TsCompProc {
        port TsPortProc g_pt_ts;
        var Worker      g_v_sender;
        var TupleSpaceList g_v_tupleSpaces
        }

    } //endgroup g_structureProc

    group g_signatures {

      signature CreateProc ( inout TupleSpaceName name,
                             in    KeyPos         keyPos )
      return TupleSpaceId
```

```
      exception ( ErrorCode );

      signature IdProc ( in TupleSpaceName name )
      return TupleSpaceId
      exception ( ErrorCode ); //endsignature IdProc

    } //endgroup g_signatures

    group g_templates {

      group g_createProcTemplates {

        template CreateProc a_createProc_r := { name := ?, keyPos := ? };

        template CreateProc a_createProc_s ( in TupleSpaceName p_name )
        := { p_name, - };

        template IdProc a_idProc_r := { name := ? };

        template IdProc a_idProc_s := { - };

      } //endgroup g_createProcTemplates

    } // endgroup g_templates

    group g_behaviour {

      altstep alt_procCreate () runs on TsCompProc {
        var TupleSpaceName name;
        var KeyPos         keyPos;

        [] g_pt_ts.getcall( a_createProc_r )
        -> param (name, keyPos) sender g_v_sender {
          var TupleSpaceId id := c_invalidId;
          var ErrorCode    errorCode := c_noError;

          if ( f_processCreateReq( name, keyPos, id, errorCode ) ) {
            g_pt_ts.reply(a_createProc_s( name ) value id );
          } else {
            g_pt_ts.raise(CreateProc, errorCode );
          }
        }
      }  //endaltstep alt_procCreate

      altstep alt_procId () runs on TsCompProc {
        var TupleSpaceName name;

        [] g_pt_ts.getcall( a_idProc_r ) -> param ( name ) sender g_v_sender {
          var TupleSpaceId id := c_invalidId;
          var ErrorCode    errorCode := c_noError;
          if ( f_processIdReq(name, id, errorCode) ) {
            g_pt_ts.reply( a_idProc_s value id );
          } else {
            g_pt_ts.raise(IdProc, errorCode );
          }
        }
      } // endaltstep alt_procId

    } //endgroup g_behaviour

  } //endgroup g_procedure

} //endmodule tupleSpace
```

G module uniqueId

```
/* ------------------------------------------------------------
 *
 * $Id: uniqueId.ttcn,v 1.12 2005/02/10 07:33:55 deiss Exp $
 *
 * @author Thomas Deiss
 * @remark Copyright: Nokia, 2004, 2005
 *
 * @remark DISCLAIMER: This TTCN-3 code is experimental code.
 *       Its purpose is to highlight strengths and weaknesses of
 *       TTCN-3, but it is not intended to be directly added to
 *       real test suites. The reader is strongly advised to check the code
 *       whether it fits the readers purpose and adapt it accordingly.
 *
 * @desc This module contains the definitions for a server to provide
 *       unique identifiers.
 */

module uniqueId {

  group g_types {

    type integer LocalId ( 0 .. infinity );
    type record of LocalId GlobalId;

    const LocalId  c_localId  := 0;
    const GlobalId c_globalId := {};

  } //endgroup g_types

  group g_prims {

    type record LocalIdReq      {};
    type record LocalIdRsp      { LocalId id };
    type record GlobalIdReq     {};
    type record GlobalIdRsp     { GlobalId id };
    type record StopReq         {};
    type record StopRsp         {};
    type record LocalIdResetReq { LocalId id }
    type record LocalIdResetRsp {}

    type union IdReq {
      LocalIdResetReq  localIdResetReq,
      StopReq          stopReq,
      LocalIdReq       localIdReq,
      GlobalIdReq      globalIdReq
    };

    type union IdRsp {
      LocalIdResetRsp  localIdResetRsp,
      StopRsp          stopRsp,
      LocalIdRsp       localIdRsp,
      GlobalIdRsp      globalIdRsp
    };

    type record of IdReq IdReqs;
    type record of IdRsp IdRsps;

  } //endgroup g_prims

  group g_configuration {

    group g_ports {
```

```
      type port IdServerPort message {
        in IdReq;
        in IdReqs;
        out IdRsp;
        out IdRsps
        }

      type port IdClientPort message {
         out IdReq;
         out IdReqs;
         in  IdRsp;
         in  IdRsps
         }

    } // endgroup g_ports

    group g_components {

      type component IdServer {
        port IdServerPort g_pt_id
        } //endtype component IdServer

      type component EmptyComponent {};


    } //endgroup g_components

  } //endgroup g_configuration

  group g_behaviour {

    type record OptPar_f_id {
      LocalId localId optional
      }

    const OptPar_f_id c_defaultPar_f := {
      localId := 0
    }

    function f_id_OptPar(in OptPar_f_id p_v_o) return OptPar_f_id
    {
      var OptPar_f_id o := p_v_o;
      if (not ispresent(p_v_o.localId)) {
        o.localId := c_defaultPar_f.localId;
      }
      return o;
    }

    function f_id ( in GlobalId p_v_globalId, in OptPar_f_id p_v_o )
    runs on IdServer
    {
      var OptPar_f_id o := f_id_OptPar( p_v_o );
      f_id_impl( p_v_globalId, o.localId );
    }

    function f_id_impl ( in GlobalId p_v_globalId,
                         in LocalId  p_v_localId ) runs on IdServer
    {
      // component state
      var GlobalId globalId := p_v_globalId;
      var LocalId  localId  := p_v_localId;

      // auxiliary variables
      var EmptyComponent s := null;
      var IdReq  idReq;
      var IdReqs idReqs;
```

```
130       var IdRsp  idRsp;
          var IdRsps idRsps;

          alt {
          [] g_pt_id.receive( a_stopReqAny_r ) -> value idReq sender s
135       {
            g_pt_id.send( a_stopRsp_s ) to s;
            stop
            };
          [] g_pt_id.receive( a_localIdReqAny_r ) -> value idReq sender s
140       {
            idRsp := f_response( idReq, globalId, localId );
            g_pt_id.send( idRsp ) to s;
            repeat;
          };
145       [] g_pt_id.receive( a_globalIdReqAny_r ) -> value idReq sender s
          {
            idRsp := f_response( idReq, globalId, localId );
            g_pt_id.send( idRsp ) to s;
            repeat;
150       };
          [] g_pt_id.receive( a_resetLocalIdReqAny_r ) -> value idReq sender s
          {
            idRsp := f_response( idReq, globalId, localId );
            g_pt_id.send( idRsp );
155         repeat;
          }
          [] g_pt_id.receive( IdReqs:* ) -> value idReqs sender s
          {
            idRsps := f_responses( idReqs, globalId, localId );
160         g_pt_id.send( idRsps ) to s;
            if ( ischosen( idRsps[sizeof(idRsps)-1].stopRsp ) ) { stop }
            repeat; /* else */
          };
          } // alt
165     } // function f_id_impl

        function f_response( in    IdReq    p_v_idReq,
                             in    GlobalId p_v_globalId,
                             inout LocalId  p_v_localId)
170     return IdRsp
        {
          var IdRsp idRsp;
          if ( ischosen (p_v_idReq.stopReq) )
          {
175         idRsp := { stopRsp := {} }
          } else if ( ischosen ( p_v_idReq.localIdResetReq ))
          {
            idRsp := { localIdResetRsp := {} };
            p_v_localId := p_v_idReq.localIdResetReq.id;
180       } else if ( ischosen ( p_v_idReq.localIdReq ))
          {
            idRsp := { localIdRsp := { id := p_v_localId }};
            p_v_localId := next( p_v_localId );
          } else if ( ischosen ( p_v_idReq.globalIdReq ))
185       {
            idRsp := { globalIdRsp := { id := append( p_v_globalId,
                                                      p_v_localId ) }};
            p_v_localId := next( p_v_localId );
          };
190       return ( idRsp );
        }  // function f_response


        function f_responses( in    IdReqs   p_v_idReqs,
195                           in    GlobalId p_v_globalId,
```

```
                              inout LocalId  p_v_localId)
      return IdRsps
      {
        var IdRsps idRsps := {};
        var integer reqAmount := sizeof(p_v_idReqs);
        var integer rspAmount := 0;
        var boolean stopRequested := false;

        for ( var integer i := 0;
              i < reqAmount and not stopRequested;
              i := i + 1 )
        {
          idRsps[i] := f_response( p_v_idReqs[i], p_v_globalId, p_v_localId
                                   );
                stopRequested := ischosen( idRsps[i].stopRsp );
        }; // for
        return ( idRsps ) ;
      } //function f_responses

  } //endgoup g_behaviour

  group g_auxiliary {

    function next ( in LocalId p_v_localId ) return LocalId
    {
      return p_v_localId + 1
      }

    function append ( in GlobalId p_v_globalId,
                      in LocalId p_v_localId )
    return GlobalId
    {
      var GlobalId completeId := p_v_globalId;
      // append localId to globalId
      completeId[ sizeof(p_v_globalId) ] := p_v_localId;
      return ( completeId );
    }

  } //endgroup g_auxiliary

  group g_templates {

    template IdRsp a_stopRsp_s     := {stopRsp := {} };

    template IdReq a_resetLocalIdReqAny_r := { localIdResetReq := ?};
    template IdReq a_stopReqAny_r         := { stopReq := ? };
    template IdReq a_localIdReqAny_r         := { localIdReq := ? };
    template IdReq a_globalIdReqAny_r        := { globalIdReq := ? };

  } //endgroup g_templates
} //endmodule uniqueId
```

H module file

```
/* ----------------------------------------------------------
 *
 * $Id: file.ttcn,v 1.6 2005/02/10 07:33:55 deiss Exp $
 *
 * @author Thomas Deiss
 * @remark Copyright: Nokia, 2004, 2005
 *
 * @remark DISCLAIMER: This TTCN-3 code is experimental code.
 *       Its purpose is to highlight strengths and weaknesses of
 *       TTCN-3, but it is not intended to be directly added to
```

```
 *       real test suites. The reader is strongly advised to check the code
 *       whether it fits the readers purpose and adapt it accordingly.
 *
 * @desc This module contains dummy definitions for file handling types
 *       and functions.
 */

module file {

  import from userData { type UserData };

  type integer ErrorCode;
  const ErrorCode c_noError := 0;

  type enumerated AccessMode { c_read, c_write };

  type integer FileDescriptor; // file descriptor

  type charstring FileName;

  external function close ( in FileDescriptor p_v_fd );

  external function eof ( in FileDescriptor p_v_fd )
  return boolean;

  external function open ( out FileDescriptor p_v_fd,
                           in FileName p_v_name,
                           in AccessMode p_v_mode )
  return ErrorCode;

  external function read_( in FileDescriptor p_v_fd, out UserData p_v_data )
  return ErrorCode;

  external function write ( in FileDescriptor p_v_fd,
                            in UserData p_v_data )
  return ErrorCode;

} //endmodule file
```