UDC 004.415.53
Original Scientific Paper

# Creating TTCN-3 Test Suite from CPN Specification

**Marina Bagić Babac** *marina.bagic@fer.hr*
*University of Zagreb*
*Faculty of Electrical Engineering and Computing, Zagreb, Croatia*

**Dragan Jevtić** *dragan.jevtic@fer.hr*
*University of Zagreb*
*Faculty of Electrical Engineering and Computing, Zagreb, Croatia*

**Abstract**

Testing of a software product is the key activity before deploying it in the real-time environment. Therefore testing should be introduced into the product development process as early as possible in order to decrease the costs of repairing the damage in later phases. In this paper we use Coloured Petri Nets (CPN) for the system specification and also as a system to be tested instead of its implementation. This specification also serves as the basis for the test suite generation. Test cases are provided in the language of Testing and Test Control Notation version 3 (TTCN-3) due to its general application area and platform independence. Stop-and-wait communication protocol has been used as a tutorial example of our approach.
**Keywords:** specification, coloured Petri nets (CPN), testing, TTCN-3

## 1. Introduction

In this paper we use the relationship between the software system specification and testing in order to introduce testing phase of the software system development as early as possible. We focus on the high-level design of the development process and the system testing phase of the V-model. System tests check whether the system as a whole implements effectively the high-level design [12]. The whole system has to be tested to check if it delivers the features required. Our testing approach is a black box testing since we relate our testing with the specification rather than the implementation of the system.

Specification language that we have chosen in order to design the system is Coloured Petri nets (CPN) due to its general purpose, strong formal syntax, semantics and visuality supported by CPN Tools [9]. Furthermore, we believe that CPN is a lot more than merely a modeling language. It is a complete modeling philosophy which has been developing for decades and is supported with numerous papers and articles [8], [10], [11] etc. and also online tool support [2]. It has been recognized in both academia and industry as useful modeling and verification formalism with a wide area of application. Its major strength is its simplicity, clearness and availability for any kind of a user.

Our selection of a testing language is Test and Testing Control Notation version 3 (TTCN-3) which is an internationally standardized testing language. The language is designed to provide a general-purpose testing language suitable for a wide range of testing applications. It can be used across the whole product development cycle [21]. TTCN-3 supports specifications of test scenarios using textual and other (graphical, tabular, etc.) presentation formats. In this paper we use TTCN-3 textual notation which is referred to as a core language in the literature [5].

In this paper we specify and test an example of a communication protocol with CPN and TTCN-3 respectively in order to explain the use of specification for creating a test suite. Another aspect of motivation is to cover system specification with testing code, and also to introduce system testing as early as possible into the development process of the system in order to decrease the costs of repairing the damage in later phases. We explain some

similarities of the two languages and focus on mapping of their data types and architectural concepts.

The idea of using CPN model as both specification and a resource for creating TTCN-3 test suite has been recently published in [17]. However, the example in [17] is the specification of a banknote processing machine while we choose an example of communication protocol. Also, the paper lacks any details on transformation between the two languages, so it leaves us with the vague idea of how the transformation might have been done. In this paper we are strongly focused on details of the transformation starting from data types to ports and component translation using the keywords from both of the languages.

The contribution of this paper relates to the model-based testing (MBT) [19], as a systematic and automated test case generation technique, being successfully applied to verify industrial-scale systems and is supported by commercial tools [7].

There is more and more interest in MBT since it promises early testing activities, hopefully early fault detection. However, to get full benefits from MBT, automation support is required, and the level of automation support of a specific MBT technique is likely to drive its adoption by practitioners [1]. Furthermore, scalability is still an open issue for large systems as in practice there are limits to the amount of testing that can be performed in industrial contexts. Even with standard coverage criteria, the resulting test suites generated by MBT techniques can be very large and expensive to execute, especially for system level testing on real deployment platforms and network facilities [7].

The outline of this paper is as follows; after the introductory section, the second and the third sections of this paper explain the benefits of CPN and TTCN-3. After the specification of the communication protocol in CPN, the fourth section focuses on CPN specification translation to TTCN-3 test suite using the elements of both languages which are used in this example.

## 2. Specification of the Communication Protocol

After setting up all the requirements for the software system, the next step in the software development process is a formal specification, or a high-level design. We use a formal specification language of Coloured Petri nets (CPN, or CP-nets) for specifying an example of communication protocol.

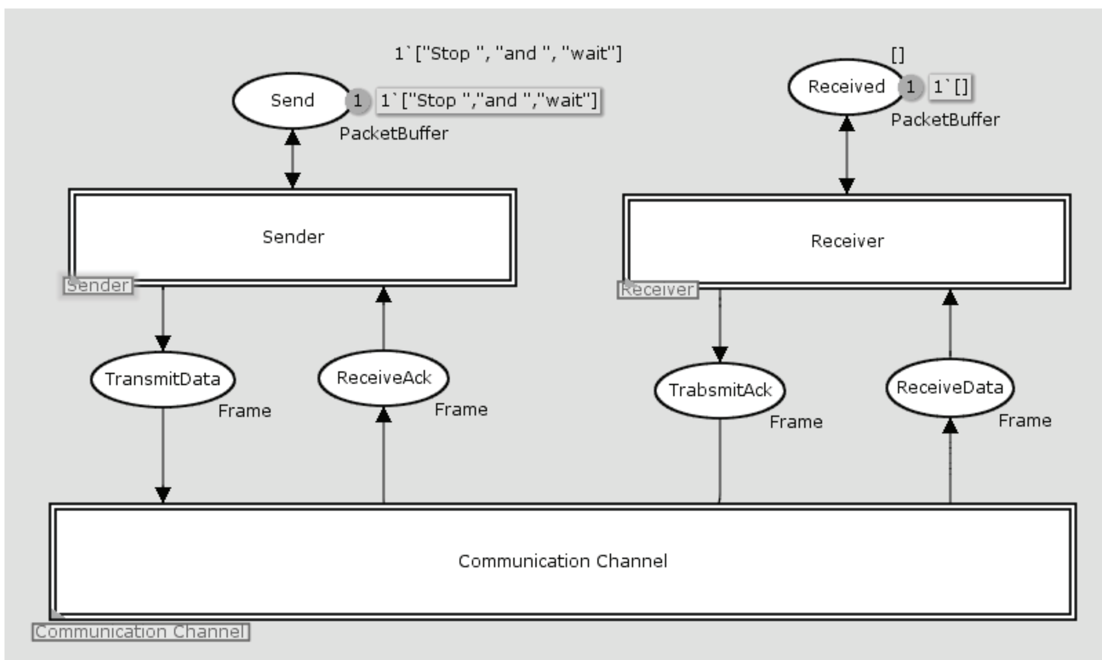


Figure 1. CPN specification of Stop-and-wait protocol

## 2.1. Coloured Petri Nets

We have chosen Coloured Petri nets as a graphical language to specify an example of communication protocol not only due to many qualities of CP-nets but also because of their strong tool support by CPN Tools [2]. The tool is maintained and extended with new releases regularly. CPN Tools provides us with the editor, simulator and state space analysis tools, and many more features [9].

CP-nets are not merely graphical representation of a system. It contains both graphical and textual elements. CPN is based on the extended finite state machine. States and transitions between them are represented via so called places, transitions and arcs between them. Places and transitions are nodes and nodes of the same type cannot be connected directly via arcs. They are used to carry tokens which are referred to as sets of colours. Colours stand for data types in CPN to emphasize the difference between the traditional Petri nets with ordinary token and the CPN which differentiate data types.

## 2.2. Stop-and-wait Protocol

In this section we specify so called Stop-and-wait protocol using CPN Tools. Stop-and-wait protocol describes the communication between the three basic participants: the sender, the receiver and the communication channel between them.

The protocol is initiated by the sender’s sending the data to the receiver. The communication channel forwards the data throughout the internet to the receiver. Once the receiver receives the data, he acknowledges the receipt of the data by sending the acknowledgement back to the sender via the same communication channel. Since the protocol is distributed and vulnerable due to the possible loss in the communication channel, or some unpredicted situations, we need to test all the communication participants

Stop-and-wait protocol has been fully specified in [10] using Design/CPN tool. Later it has been slightly changed and adapted to the latest CPN Tools releases in a form of so called Simple protocol. However, we have taken the initial Stop-and-wait protocol as it has been specified in [10] and we have adapted it to the latest CPN Tools version.

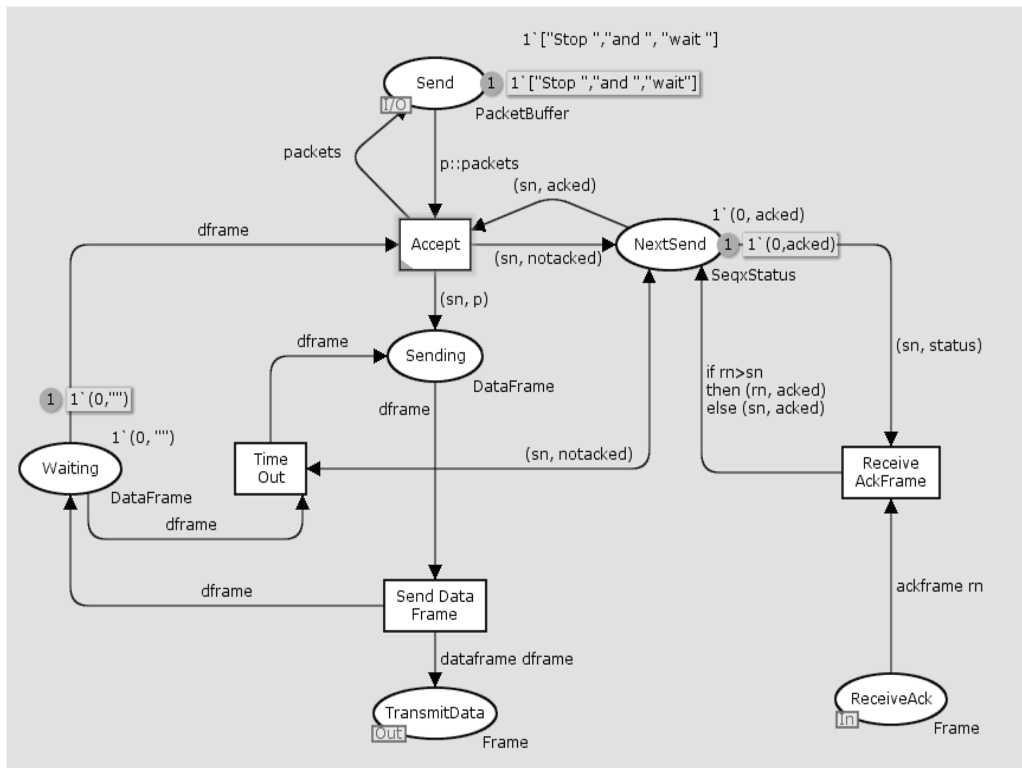


Figure 2. CPN specification of the sender module

The protocol originates from the datalink control layer of the OSI network architecture. The participants are depicted in Figure 1 as CPN modules. The protocol transfers a number of packets from the sender to a receiver. The communication medium may lose packets and packets may overtake each other. Hence, it may be necessary to retransmit packets and to ignore doublets and packets that are out of order.

Figure 1 depicts the protocol from the high-level perspective since the details of implementation are hidden within the module hierarchy. This model nicely corresponds to the TTCN-3 specification of test components since CPN has explicitly defined three components/modules and the communication ports/places between them. In order to specify TTCN-3 communication ports, interfaces of these components/modules are crucial. Ports represent interfaces between the test components and the test components and the protocol (this is why ports were historically called PCOs – Points of Control and Observation).

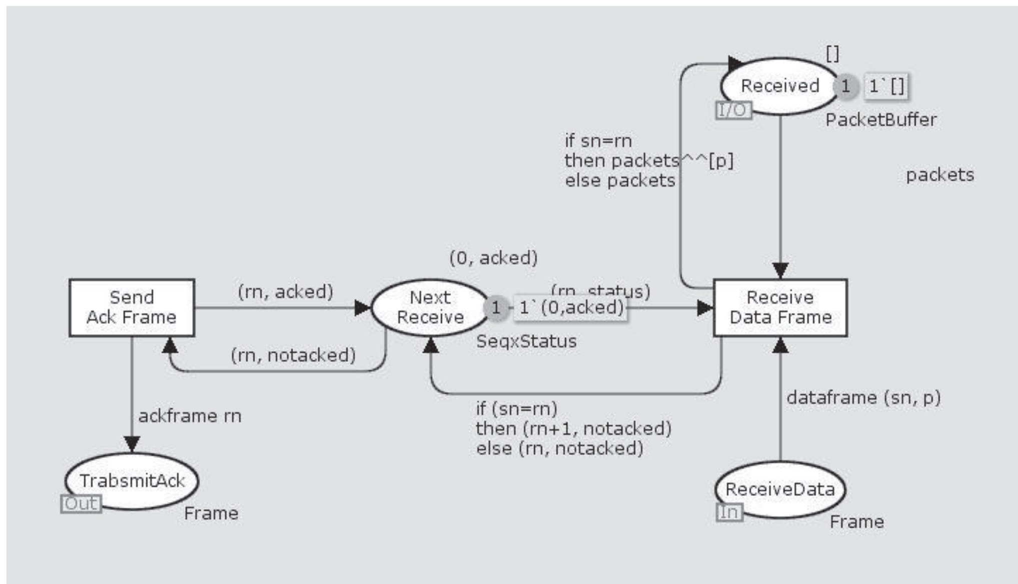


Figure 3. CPN specification of the receiver module

Lower-level design of the Sender module is shown in Figure 2. We do not explain each place and transition of the CP-net as it is thoroughly explained in [10]. Besides, in order to test the specified functionality in TTCN-3, it suffices to extract only input/outgoing places from this specification (places *Send*, *TransmitData* and *ReceiveAck*). These places are directly connected to the corresponding places of communication channel.

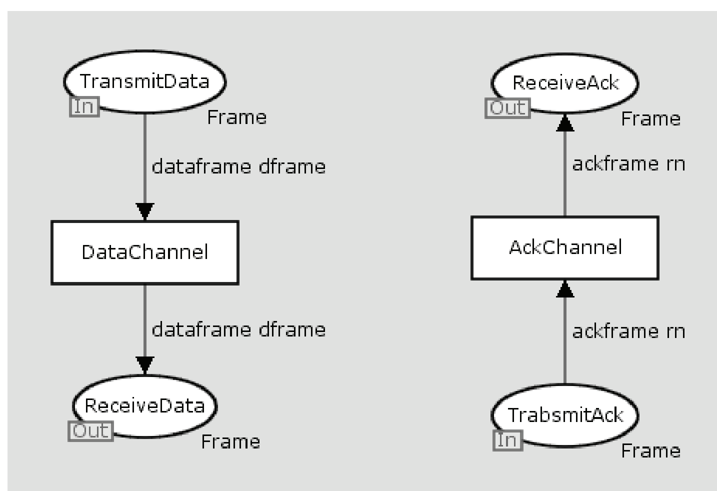


Figure 4. CPN Specification of the Channel Module

In the same manner the receiver module has been specified in Figure 3. The relevant places are *Received*, *ReceiveData* and *TransmitAck* being the interface for the communication channel. And last, but not least important is the specification of the channel in Figure 4. For the sake of simplicity here it is an ideal channel that cannot lose messages, so it only works as the medium which transfers the accepted messages towards the receiver/sender. It contains the relevant places *TransmitData*, *ReceiveData*, *ReceiveAck* and *TransmitAck*. which we are going to be used for testing purposes.

## 3. TTCN-3 Testing

TTCN-3 (Testing and Test Control Notation version 3) is a globally adopted standard test notation for the specification of test cases. TTCN-3 is intended for various application areas like protocol testing (e.g. mobile and internet protocols), supplementary service testing, module testing, testing of CORBA based platforms, testing of API's etc. TTCN-3 is an abstract language in the sense that it is test system independent which means that a test suite in TTCN-3 for one application can be used in any test environment for that application [12].

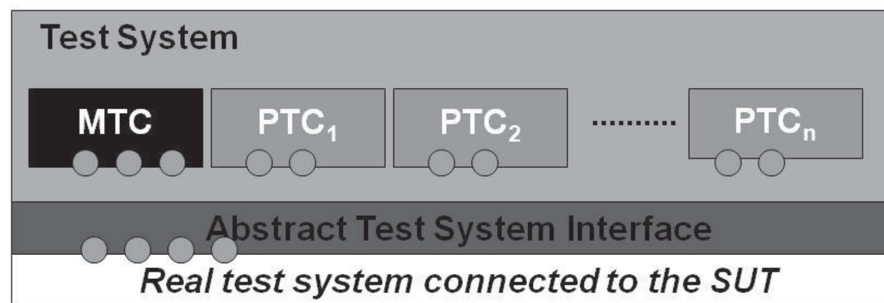


Figure 6. TTCN-3 test components [10]

### 3.1. TTCN-3 Test Suite Architecture

A TTCN-3 specified test suite is a collection of test cases together with all the declarations and components needed for the test. The top-level unit of a TTCN-3 test suite is a module, which can import definitions from other modules. A module consists of a declarations part and a control part. The declarations part of a module contains definitions, e.g. for test components, their communication interfaces (ports), type definitions, test data templates, functions, and test cases [12].

The control part of a module calls the test cases and describes the test campaign. For this, control statements like if-then-else and while loops are supported. They can be used to specify the selection and execution order of individual test cases. The module parameter list defines a set of values that are supplied by the test environment at runtime. During test execution these values are treated as constants and can be accessed within the module from any scope. Groups can be used to structure the declarations of a module to improve the readability.

The port and component types are used to define the configuration of a test system. Figures 6 shows test components in a test system. Each test system contains at least one main test component (MTC) and an arbitrary number of parallel test components (PTCs) which are created by MTC at the test system startup. These components communicate with the SUT (System Under Test) through abstract test system interface which is defined using ports.

Figure 7 shows the communication between the test case and the SUT via a single port. Port serves as a point of unidirectional or bidirectional message exchange between the test components or the test component and the SUT. During this communication tests produce sending and receiving messages which are stimulus to the SUT expecting a response from it. Depending on the kind of the response, we get test verdicts about passing or failing the test cases.

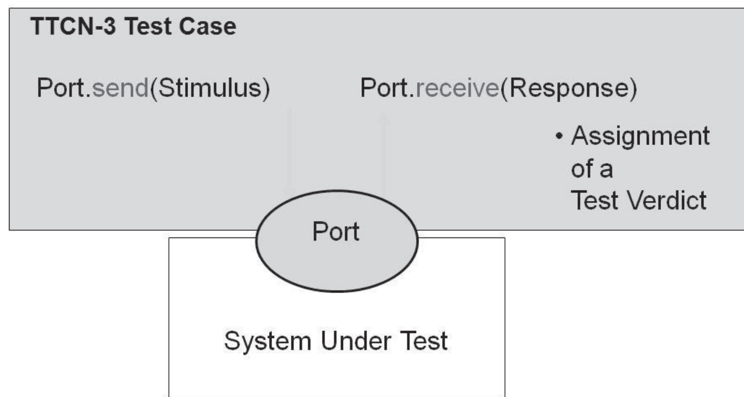


Figure 7. Black-box testing with TTCN-3 [10]

Types of protocol messages are defined as structured types, which can be constructed as record, set, enumerated types, etc. from basic types or other structured types. TTCN-3 supports a number of predefined basic types. They include typical programming basic types such as integer, boolean and string types, as well as testing-specific types such as verdict type, port and component type. It has pronouncedly a rich data type foundation in order to support various kinds of applications to be tested as a SUT.

Communication between the ports can be either message-based or procedure-based depending on the complexity of responses from the SUT. While message-based communication lies on the pure exchange of the data, procedure-based communication involves computation actions as well as exchange of data from either the SUT or a test system before sending a response to the requestor entity. In the next section we use the concept of message-based communication.

## 4. TTCN-3 Test Cases for Stop-and-wait Protocol

### 4.1. Mapping CPN to TTCN-3 Data Types

The most important thing about both CPN and TTCN-3 language specifications is to transfer data throughout the process of modeling or testing. CP-net carries tokens from the initial place through the sequence of places connected with the transitions while TTCN-3 test cases send and receive data exchanged between either the test components themselves or between the test components and SUT. Therefore, we begin by generating the corresponding data types and structures from CP-net to TTCN-3. Since there is a great amount of similarity in the specification of both data models, the process could be also automated. Some basic data types are the same, e.g. string corresponds to charstring, int to integer, etc.

```
▼colset Packet = string;
▼colset PacketBuffer = list Packet;
▼colset Seq = int;
▼colset Status = with acked | notacked;
▼colset SeqxStatus = product Seq * Status;
▼colset DataFrame = product Seq * Packet;
▼colset AckFrame = Seq;
▼colset Frame = union dataframe:DataFrame + ackframe : AckFrame;
▼var p : Packet;
▼var packets : PacketBuffer;
▼var dframe : DataFrame;
▶var sn rn
▼var status : Status;
▼var m : AckFrame;
```

Table 1. CP-net data definitions from CPN Tools

Data translation from CPN to TTCN-3 is more relieved than the opposite direction since TTCN-3 has much richer data vocabulary than CPN and supports wider diversity of user defined data structures of types and templates.

Table 1 shows colour set definitions for Stop-and-wait protocol (*colset* is a keyword for the colour set). Here, *Packet* is an *alias* for the string and *PacketBuffer* is a list of *Packets*. *Seq* is an alias for the *int*. *Status* is an enumeration type defined here with the values of *acked* and *notacked* for the acknowledgement manipulation.

| CPN colset | maps to TTCN-3 equivalent |
|---|---|
| string | charstring |
| int | integer |
| with | enumerated |
| record | record |
| union | union |
| list | record of |
| product | record |

Table 2. Correspondence between CPN colour sets and TTCN-3 types

*SeqxStatus* and *DataFrame* are both Cartesian products (ordered data types) placing one *Seq* (*int*) at the first position and *Status* or *Packet* at the second position, respectively. *AckFrame* is an alias for the *Seq*. And, *Frame* is a union structure which is consisted of either *DataFrame* Cartesian product or merely *int* (*AckFrame*). The rest of the Table 1 is a few variable definitions according to previously defined colour sets.

For each colour set from Stop-and-wait protocol we find the corresponding TTCN-3 data type (see Table 2) and according to these mappings we create corresponding TTCN-3 data type definitions for our protocol (see Table 3). Note that both CPN and TTCN-3 data definitions are textual. Even though CPN is a primarily graphical language and TTCN-3 also has its own graphical version [6], data definitions and their manipulation remain textual.

```
type charstring Packet;
type integer Seq;

type record of Packet PacketBuffer;
type enumerated Status {
   acked, notacked};
type record SeqStatus{
   Seq seq,
   Status status
}
type record of seqStatus SeqxStatus;
type record SeqPacket{
   Seq seq,
   Packet packet
}
type record of SeqPacket DataFrame;
type Seq AckFrame;
type union Frame {
   DataFrame dataframe,
   AckFrame ackframe
}
template Frame packet0 := {dataframe := {{seq := 0, packet := "Stop "}}}
template Frame ack0 := {ackframe := 0}
template Frame packet1 := {dataframe := {{seq := 1, packet := "and "}}}
template Frame ack1 := {ackframe := 1}
template Frame packet2 := {dataframe := {{seq := 2, packet := "wait"}}}
template Frame ack2 := {ackframe := 2}
```

Table 3. TTCN-3 data definitions of Stop-and-wait protocol

While CPN uses variables as the message carriers in the specifications, TTCN-3 uses so called templates. Therefore, in Table 3 type definitions are followed by template definitions.

### 4.2. Mapping CPN Modules to TTCN-3 Test Cases

After translating data types from CPN to TTCN-3, there is an important step left – test cases generation. In order to make test cases we first need to specify type components in which test cases are going to be run. CPN specification serves in a way that we just need to extract the required relevant information for the test components creation. Even the places between the modules are called ports in CPN. The same concept holds for the test components. Test components need to be defined with their interfaces (ports) towards the SUT.

```
type port TransmitData message {
   in Frame
}
type port ReceiveData message {
   out Frame
}
type port TransmitAck message {
   in Frame
}
type port ReceiveAck message {
   out Frame
}
type port Send message {
   inout PacketBuffer
}
type port Received message {
   inout PackBuffer
}
type component Channel {
   port TransmitData pt_transmitData;
   port ReceiveData pt_receiveData;
   port TransmitAck pt_transmitAck;
   port ReceiveAck pt_receiveAck
}
type component Sender {
   port TransmitData pt_transmitData;
   port ReceiveAck pt_receiveAck;
}
type component Receiver {
   port TransmitAck pt_transmitAck;
   port ReceiveData pt_receiveData;
}
```

Table 4. Test suite architecture: port and component types

Figure 8 shows test components that are used for testing our protocol. These are all parallel test components as the main test component creates and starts them all and delivers the final verdict.

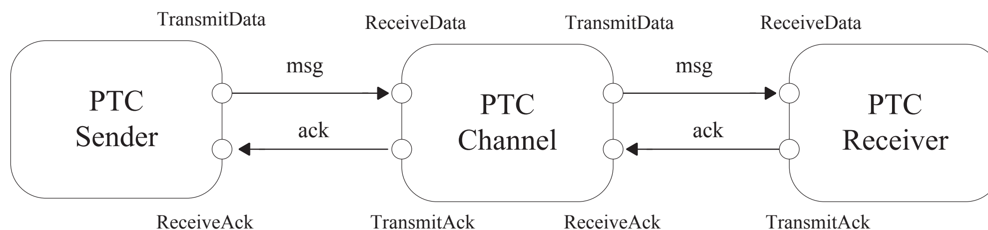


Figure 8. Parallel test components of the test suite

The main test component performs a test cast execution in Table 8. Figure 8 also emphasizes the act of ports and the directions of frames flow, i.e. from the sender towards the channel, and from the channel towards the receiver. Conversely, acknowledgements flow in the opposite direction, from the receiver to the channel and from the channel back to the sender.

In order to make a framework for running test cases it is necessary to define test system interface. However, when testing the whole protocol with a number of involved participants, it is required to first define what exactly is going to be tested. Therefore, we make here three test components, one for each of the involved participants. When testing the sender, he is our SUT and we use the channel and the receiver test components to act as reliable communication participants. When testing the communication channel, we use the sender and the receiver test components as reliable communication participants. And when testing the receiver, we involve the sender and the channel test components as reliable communication entities.

```
testcase tc_testChannel() runs on Channel {

    alt  {
       [] pt_receiveData.receive(packet0) {
          transmitData.send(packet0)
          }
       [] pt_receiveData.receive(ack0) {
          transmitAck.send(ack0)
          }
       [] pt_receiveData.receive(packet1) {
          transmitData.send(packet1)
       }
       [] pt_receiveData.receive(ack1) {
          transmitAck.send(ack1)
          }
       [] pt_receiveData.receive(packet2) {
          transmitData.send(packet2)
          }
       [] pt_receiveData.receive(ack2) {
          pt_transmitAck.send(ack2)
          setverdict(pass);
          }
     }
}
```

Table 5. Test case for the channel test component

All the relevant type ports and test components made out of them are shown in Table 4. For the sake of simplicity we have chosen the same names for ports as they were defined in CPN specification. The communication between these ports is message-based as it has been defined in CPN specification. Types of messages are also strictly taken from corresponding CPN definitions (Table 3). For simulation purposes we have specified three frames to be sent by the sender test component (“Stop ”, “and ” and “wait”) to the receiver test component via the channel test component.

```
testcase tc_testSender() runs on Sender {

   pt_transmitData.send(packet0);
   alt {
          [] pt_receiveAck.receive(ack0) {
             pt_transmitData.send(packet1);
             }
          [] pt_receiveAck.receive {
             setverdict(fail)
             }
   }
```

```
   alt {
            [] pt_receiveAck.receive(ack1)  {
               pt_transmitData.send(packet2)
               }
            [] pt_receiveAck.receive {
               setverdict(fail)
               }
      }
      alt {
            [] pt_receiveAck.receive(ack2) {
               setverdict(pass)
               }
            [] pt_receiveAck.receive {
               setverdict(fail)
               }
         }
   }
```

Table 6. Test case for the sender test component

Last but not least important, we make test cases to run within our test components. Table 5 shows test case to run on the channel test component. In the spirit of a black-box testing test case is based on stimulus – response communication. If it receives a data packet or an acknowledgement, it transfers it to the appropriate outgoing port.

```
testcase tc_testReceiver() runs on Receiver {

   alt {
   [] pt_receiveData.receive(packet0) {
      pt_transmitAck.send(ack0);
       }
   [] pt_receiveData.receive {
      setverdict(fail)
       }
   }
   alt {
   [] pt_receiveData.receive(packet1) {
      pt_transmitAck.send(ack1);
       }
   [] pt_receiveData.receive {
      setverdict(fail)
       }
   }
   alt {
   [] pt_receiveData.receive(packet2) {
      pt_transmitAck.send(ack2);
        setverdict(pass);
      }
   [] pt_receiveData.receive {
      setverdict(fail)
       }
   }
}
```

Table 7. Test case for the receiver test component

As it runs on the channel test component, it has four ports (Figure 8) which correspond to CPN port places (Figure 1) *TransmitData*, *ReceiveAck*, *TransmitAck* and *ReceiveData* coloured with *Frame* colset. We assume an ideal channel with no transmission loss of data and instant response on stimuli acceptance. Therefore, upon the receipt of a frame from the sender, e.g. *packet0*, it instantly sends the received *packet0* to the receiver. Or in the other direction, upon the receipt of an acknowledgement from the receiver, e.g. *ack0*, it instantly sends the received *ack0* back to the sender. If all the transmitted packets are properly delivered and acknowledged, the verdict of a test case is set to pass.

Similar principle remains for the rest of the test cases (Tables 6 and 7), i.e. *tc_testSender* and *tc_testReceiver*. The only difference is the *alt* keyword which is introduced for different options in case the messages come to the recipient out of order, when the verdict is set to fail.

Test case *tc_testProtocol* runs on the main test component. It starts first and creates all the parallel test components in the system by connecting the corresponding ports.

Once our test suite is complete and ready for the execution, we can simulate the test component communication. We have used Loong Testing tool [13] which has served not only for testing execution, but also as a verification tool for our testing code (syntax checker and the compiler). Test results were compatible with the initial CPN specification.

```
testcase tc_testProtocol() runs on EmptyComponent {

	var Sender v_sender;
	var Channel v_channel;
	var Receiver v_receiver;

	v_sender := Sender.create;
	v_channel := Channel.create;
	v_receiver := Receiver.create;

	connect(v_sender: pt_transmitData, v_channel: pt_receiveData);
	connect(v_sender: pt_receiveAck, v_channel: pt_transmitAck);
	connect(v_channel: pt_transmitData, v_receiver: pt_receiveData);
	connect(v_channel: pt_receiveAck, v_receiver: pt_transmitAck);

}
```

Table 8. Test case embedded in MTC

The main strength of the above approach is obtaining tests and test results very early in the development process. However, scalability issues and how to handle more complex examples and larger real world applications still remain an open issue. One step is to automate some of the process. Concerning the Stop-and-wait example, data transformation from CPN to TTCN-3 is proposed, as well as I/O (Input/Output) places (ports in CPN) to ports in TTCN-3, and CPN modules to test components. Also, expressions of directed arcs from CPN could have been transformed to send/receive statements in TTCN-3.

## 5. Conclusion and Future Work

Although at first glance specification of system requirements and testing do not seem to be very closely connected, we have given an example of their close relationship in order to introduce testing as early as possible into the development process of a software product. Furthermore, we have used the specification of the system as a system under test and also as the origin for writing test cases.

In the words of concrete languages and tools, we have used Coloured Petri nets (CPN) for system specification and Testing and Test Control Notation version 3 (TTCN-3) for testing to prove that it is possible to map specification model to testing model of a system.

Even more, revealing the great similarity between their underlying concepts (data types and extended finite state machine which they both support), the future work is to automate some of the process of their mapping. Using the example of a communication protocol, where the sender sends the message to the receiver via the communication channel and the receiver acknowledges it by sending the appropriate reply throughout the same channel, we have given an idea of how to map some of the basic data types from CPN to TTCN-3, CPN port places to communicating TTCN-3 ports, CPN modules to TTCN-3 test components, CPN message sending and receiving mechanism to TTCN-3 stimulus and response communication, etc. We have used elements of both languages showing how to provide an automatic approach in the future. Automated generation of test cases would not only introduce tests very early into the development process, but would also accelerate the process of testing itself.

## Acknowledgements

This work was carried out within research project 036-0362027-1640 "Knowledge-based network and service management", supported by the Ministry of Science, Education and Sports of the Republic of Croatia.

## References

[1] Bringmann, E.; Krämer, A. Model-Based Testing of Automotive Systems. 2008 International Conference on Software Testing, Verification, and Validation, International Conference on Software Testing, Verification, and Validation (ICST). pp. 485–493, ISBN 978-0-7695-3127-4, 2008.

[2] CPN Tools. http://cpntools.org/, Feb. 2013.

[3] Deiß, T: TTCN-3 for large systems, *Systems Validation workshop Paris*, Nokia Research Center, pp. 1-61, 2004.

[4] Ebner, M. *An introduction to TTCN-3 version 3*. ITU-T Study Group 17, Geneva, 5-14th Oct 2005.

[5] ETSI, Methods for Testing and Specifcation (MTS). *The Testing and Test Control Notation version 3; Part 1 TTCN-3 Core Language*. ETSI ES 201 873-1, V4.3.1, 2011.

[6] ETSI, Methods for Testing and Specifcation (MTS).*The Testing and Test Control Notation version 3: Graphical Presentation Format*. ES 201 873-3, V3.2.1, 2007.

[7] Hemmati, H.; Arcuri, A.; Briand, L. Achieving Scalable Model-Based Testing Through Test Case Diversity. technical report, Simula Research Laboratory, 2010.

[8] Jensen, K. An Introduction to the Practical Use of Coloured Petri Nets. Lectures on Petri Nets II: Applications, Lecture Notes in Computer Science vol. 1492, Springer-Verlag, pp. 237-292., 1998.

[9] Jensen, K. Kristensen, L.M. *Coloured Petri Nets*. Springer-Verlag Berlin Heidelberg 2009.

[10] Kristensen, L.M.; Christensen, S.; Jensen, K. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer* 2 pp. 98–132, 1998.

[11] Kristensen, L.M.; Jřrgensen, J.B., Jensen, K. Application of Coloured Petri Nets in System Development. Lectures on Concurrency and Petri Nets - Advanced in Petri Nets. Proc. of 4th Advanced Course on Petri Nets. Vol. 3098 of Lecture Notes in Computer Science, pp. 626-685. Springer-Verlag, 2004.

[12] Lalani, N. *Validation of Internet Applications*. Fachhochschule Wiesbaden, Karlstad University, University of Applied Sciences, Sweden, Master Thesis, 2005.

[13] Loong Testing, http://ttcn.ustc.edu.cn/MainPageEn.html, Feb. 2013.

[14] OpenTTCN Oy - OpenTTCN Ltd, OpenTTCN DocZone, TTCN-3 language reference, http://wiki.openttcn.com/media/index.php/OpenTTCN/Language_reference, Feb. 2013.

[15] Rennoch, A.; Desroches, C.; Vassiliou-Gioles, T., Schieferdecker, I. TTCN-3 Reference Card, http://www.blukaktus.com/card.html, online edition 4.4.1, Apr. 2012.

[16] Santos-Neto, P.; Resende, R.; Pádua, C. Requirements for information systems model-based testing. Proceedings of the 2007 ACM symposium on Applied computing - SAC '07. Symposium on Applied Computing. pp. 1409–1415, ISBN 1-59593-480-4. edit., 2007.

[17] Schnattinger, T.; Pietschker, A. Using Colored Petri Nets for System Specifications and as a System Under Test Prototype, *TTCN-3 User Conference 2011,* 2011.

[18] Shafique, M.; Labiche, Y. A Systematic Review of Model Based Testing Tool Support. Carleton University, Technical Report, May 2010.

[19] Stepien, B. TTCN-3 in a Nutshell. University of Ottawa, online tutorial, http://www.site.uottawa.ca/~bernard/ ttcn3_in_a_nutshell.html, Feb. 2013.

[20] TTCN-3 home page, http://www.ttcn-3.org/, Feb. 2013.

[21] Willcock, C.; Tobies, S.; Deiss, T.; Keil, S.; Engler, F.; Schulz, S. *An Introduction to TTCN-3*. John Wiley & Sons Ltd, 2005.

[22] Zander, J.; Schieferdecker, I.; Mosterman, P. J., eds. Model-Based Testing for Embedded Systems. Computational Analysis, Synthesis, and Design of Dynamic Systems. 13. Boca Raton: CRC Press. ISBN 978-1-4398-1845-9, 2011.