Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer GAUG-ZFI-MSC-2008-09

Masterarbeit

im Studiengang "Angewandte Informatik"

Lernen von Protokollautomaten

Sergei Polonski

Institut für Informatik

Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

01.05.2008

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel.	+49 (5 51) 39-1 44 14
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 01.05.2008

Master´s Thesis

Learning of protocol-based automata

Sergei Polonski

01.05.2008

Supervised by Prof. Dr. Jens Grabowski
Software Engineering for Distributed Systems Group
Institute for Computer Science
University of Göttingen, Germany

Acronyms

API	*Application Programming Interface*
CSV	*Comma Separated Values*
DFA	*Deterministic Finite Automaton*
DOM	*Document Object Model*
IDE	*Integrated Development Environment*
IDL	*Interface Definition Language*
Inres	*Initiator- Responder*
IUT	*Implementation Under Test*
J2ME	*Java 2 Micro Edition*
LTS	*Labeled Transition System*
LTSML	*Labeled Transition System Markup Language*
MAT	*Minimal Adequate Teacher*
PDU	*Protocol Data Unit*
SAP	*Service Access Point*
SAX	*Simple API for XML*
SDU	*Service Data Unit*
SP	*Service Primitive*
StAX	*Streaming API for XML*
SUT	*System Under Test*
TRex	*TTCN-3 Refactoring and Metrics Tool*

TTCN-3	*Testing and Test Control Notation Version 3*
UML	*Unified Modeling Language*
XML	*eXtensible Markup Language*
XP	*eXtreme Programming*
XSD	*XML Schema Definition*

Contents

1 Introduction **6**

2 Foundations **8**
2.1 Inres System 8
2.1.1 Inres Service 8
2.1.2 Medium Service 11
2.1.3 Inres Protocol 11
2.1.3.1 Connection Establishment Phase 12
2.1.3.2 Data Transmission Phase 12
2.1.3.3 Disconnection Phase 13
2.2 Learning of Finite Automata Using Queries 13
2.3 Conformance Test of Protocols 16

3 Adaptation of the Learning Algorithm **18**
3.1 Realisation of Queries 18
3.2 Coding 19

4 Implementation **23**
4.1 XML Overview 23
4.1.1 Streaming API 23
4.1.2 Tree-based (*Document Object Model*s) 24
4.1.2.1 DOM 24
4.1.2.2 JDOM 25
4.2 Description of the XML Format 26
4.2.1 Input File Format 26
4.2.2 Output File Format 27
4.3 Simple Traces Format 27
4.4 Implementation of the Learning Algorithm 28
4.5 Coder and Decoder 28

5 Experiments **33**
5.1 Adaptation of Inres Protocol 33
5.2 Test Suite 34

6 Conclusion **41**

List of Figures **42**

List of Tables **43**

Listings **44**

7 Appendix **45**

Bibliography **50**

1 Introduction

The standard process of software engineering has four basic phases: analysis, design, implementation, and testing. In this classical model the testing of the implemented system is based on the specifications developed in the first two phases. However in practice there are several situations, in which the specification is not up-to-date or even missing completely. For example the software engineering method *eXtreme Programming* (XP) works without any specifications. This method sees a specification as an ever changing and thus very time intensive and therefore unnecessary phase for an effective development process. Another real-life example is an attempt of the managers to keep the development costs as low as possible. One measure of economy is to cut the budget for the first and second phases of software engineering. This approach is heavily criticised for the reason of higher error-rate of the software and often even higher costs for the development. Software products developed in this way have either fragmentary specification or no formal specification at all. (Somm 06)

Further, there is older software, which needs to be updated. Sometimes it had no specification initially; in other cases it might be lost over the years. But even if older software had a proper specification initially, it can differ significantly in case the software was updated over the years without updates of the specification. The test cases describe software for the most part better as this kind of specification.

The goal of this thesis is to create a formal specification of *System Under Test* (SUT) from a test suite. For that purpose the learning algorithm of Dana Angluin (Angl 87) will be modified in such a way that it is able to accept a test suite as input. The expected result of the work of this algorithm is a finite automaton, which describes the SUT and therefore is a formal specification of the SUT.

The results of the thesis can also be useful in a standard process of software engineering as an indicator of the quality of the test suite. We could generate a formal description of the system based on the test suite and then compare it with an already existing specification. If the specifications are equivalent, the quality of initial test suite is sufficient and does not need to be improved.

The thesis has four main parts, introduction and conclusion. The introduction gives a short overview of the goals of this thesis and the current problems it may help to solve. The Chapter 2 introduces some basic theoretical backgrounds including the description of the learning algorithm and the *Initiator- Responder* (Inres) protocol for the case study. The Chapter 3 is devoted to the modification of the algorithm. The Chapter 4 focuses on the implementation. The Chapter 5 contains our experiments and the results of the work. The Chapter 6 gives a short summary of the results and an overview of opportunities for a further research.

2 Foundations

This chapter describes the theoretical background of the thesis explaining the Inres system, learning of finite automata using queries and the grounds of conformance test of protocols.

2.1 Inres System

The following section describes the Inres system, which was developed by Dieter Hogrefe (Hogr 89; Hogr 91). The fact that Inres contains many basic concepts of the real life protocol for communication systems (Neuk 04), is easy to understand and not too big, makes it very interesting for research and educational purposes.

Figure 2.1 illustrates the structure of the Inres system.

2.1.1 Inres Service

Every user who wants to communicate with another user by the means of this service, has to make a connection in order to exchange data. The service is therefore connection-oriented. To keep the system simple, the service is asymmetric. Only Initiator-user can initiate a connection and send data. The Responder-user can either accept or reject the connection. If he accepts the connection, he can receive data from the Initiator-user.

The Inres service can be accessed on two *Service Access Point*s (SAPs), ISAPini and ISAPresp. Table 2.1 shows the *Service Primitive*s (SPs) used for the communication in Inres system.

The goal of an exchange of the first four SPs (ICONreq, ICONind, ICONresp and ICONconf) is the connection establishment. The next two SPs (IDATreq , IDATind) deal with data transmission and the last two SPs (IDISreq, IDISind) with connection release.

During this process several problems may occur (see Figure 2.2): The loss of the message can interrupt the connection establishment. That can happen, when Initiator sends the message CR (a) or when Responder acknowledges the connection establishment with a message CC (b). In both cases the Initiator continues waiting for an acknowledgment message from Responder. If the message is not received,

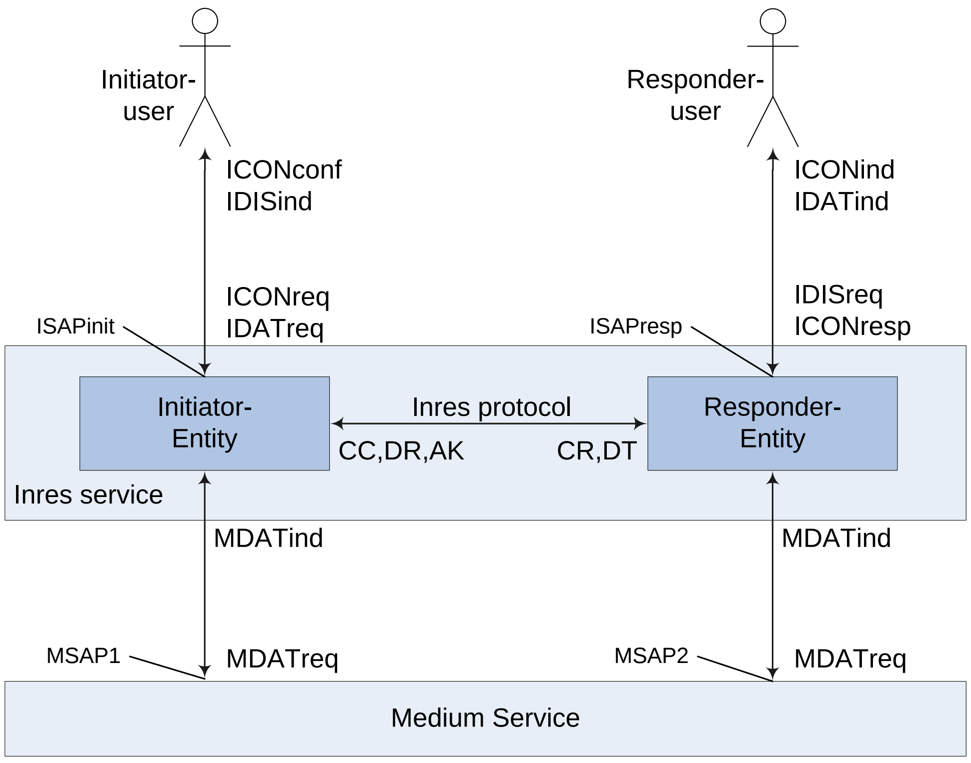


Figure 2.1: Inres System

SP	Parameter	Description
ICONreq		Request of connection by Initiator-user
ICONind		Indication of connection by provider
ICONresp		Response to connection attempt by Responder-user
ICONconf		Confirmation of connection by provider
IDATreq	ISDU	Data transfer from the Initiator-user to the provider, ISDU is user data
IDATind	ISDU	Data transfer from the provider to the Responder-user,
		ISDU is user data
IDISreq		Request of disconnection by the Responder-user
IDISind		Indication of disconnection by the provider

Table 2.1: Service Primitives *of the Inres Protocol*

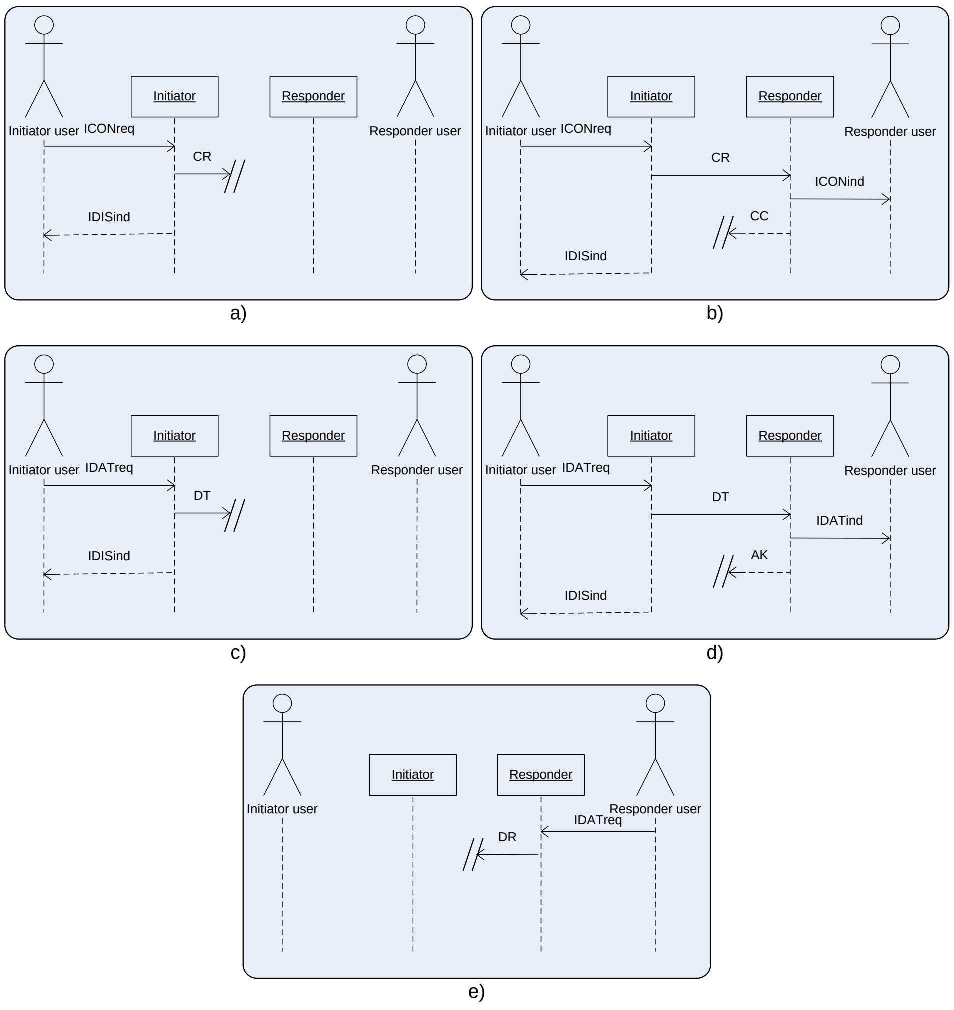


Figure 2.2: Unsuccessful Transmissions

the Initiator sends on expiry of timer an indicator IDISind. The parts (c) and (d) of the Figure 2.2 illustrate possible errors in the data transmission phase. The errors can happen during the data transmission from Initiator to Responder or while the Responder acknowledges correctly received data. On the analogy of cases (a) and (b) the Initiatorsends an indicator IDISind on expiry of timer after waiting vainly for an acknowledgment message. Furthermore the message can be lost, when Responder disconnects (e). As the Responderdisconnects, the Initiator is still connected. The Responder ignores the following data transmission. The Initiator does not get an acknowledgment and goes on expiry of timer into a disconnected state.

2.1.2 Medium Service

The Medium service is symmetrical and not connection-oriented. The data transmission is unreliable, which in this case means that the data can be lost and disordered, but can't be corrupted or duplicated. The Medium service can be accessed at two SAPs: MSAP1 and MSAP2. The SPs MDATreq and MDATind are used for data transmission from one SAP to another.

2.1.3 Inres Protocol

This connection-oriented protocol operates between two protocol entities Initiator and Responder. The communication happens by exchange of the *Protocol Data Unit*s (PDUs). Table 2.2 shows the specification of the respective PDUs.

PDU	Parameter	Description
CR		Connection establishment
CC		Connection confirmation
DT	seq, ISDU	Data transfer
AK	seq, ISDU	Acknowledgement
DR		Disconnection

Table 2.2: Protocol Data Units *of the Inres Protocol*

The three communication phases are the connection establishment phase, the data transmission phase and the disconnection phase. Each phase uses different PDUs and SPs; unexpected PDUs and SPs are ignored by Initiator and Responder.

2.1.3.1 Connection Establishment Phase

In the first phase the Initiator-user initiates a connection at the entity Initiator with an ICONreq. Next, a CR is sent from the entity Initiator to the entity Responder. The Medium service delivers it.

Three basic scenarios are possible:

- If Responder answers with CC, the Initiator issues an ICONconf to its user. That means the data phase can be entered.
- If Responder answers with DR, the disconnection phase is entered.
- The third option is that the Responder does not answer at all. In this case the Initiator waits for five minutes and sends CR again. After four attempts it enters the disconnection phase.

After getting a CR, Responder issues an ICONind to the Responder-user. If the Responder-user accepts the connection, he will send ICONresp. The Responder sends then a CC to Initiator and the data transmission phase is entered. Otherwise the Responder-user sends the IDISreq and enters the disconnection phase.

2.1.3.2 Data Transmission Phase

In the data transmission phase the Initiator-user first issues an IDATreq. Then the Initiator sends a DT to the Responder. After that the Initiator can receive another IDATreq. IDATreq has as parameter an ISDU. ISDU is a *Service Data Unit* (SDU) for the transmission of information to the peer user. The Initiator sends this user data as a parameter of DT to the Responder. After waiting for 5 seconds for an acknowledgment AK, he sends the DT again. There are four attempts possible. Then the Initiator enters the disconnection phase.

Both DT and AK have a sequence number as a parameter (0 or 1). Initiator sends a DT with a sequence number 1. The acknowledgment is correct, if the sequence numbers of DT and AK are identical. Are the numbers different, the last DT is sent again. The same happens, if AK does not arrive in 5 seconds. There are four attempts possible, before the Initiator enters the disconnection phase. If Responder receives a DT with the expected sequence number (here 1), he issues the ISDU as a parameter of an IDATind to the Responder-user and sends an AK with the sequence number to the Initiator. If the number is unexpected, the Responder sends an AK with the number of the last correctly received DT. In this case the user data (ISDU) must be ignored. The consequence of the receipt of a CR or an IDISreq is that the Responder enters the connection establishment or disconnection phase respectively.

2.1.3.3 Disconnection Phase

After receiving IDISreq from the Responder-user, the Responder sends a DR to the Initiator. The Initiator sends then IDISind to the Initiator-user. The participants are disconnected; a new connection can be established (See Part 2.1.3.1).

2.2 Learning of Finite Automata Using Queries

The method of learning of finite automata using queries was developed by Dana Angluin (Angl 87). The algorithm, also called the learner, uses a *Minimal Adequate Teacher* (MAT), also called the oracle, for an effective learning of a target concept.

Throughout this thesis, the following notation is used: A *Deterministic Finite Automaton* (DFA) $\mathcal{A}$ is a tuple ($Q, \Sigma, \delta, q_0, F$), where

- Q is a finite non-empty set of states in the automaton $\mathcal{A}$
- Σ is a binary input alphabet
- δ is a transition function, which maps $Q \times \Sigma$ into Q
- $q_0 \in Q$ is the initial state
- $F \in Q$ is a set of accepting states

For any word $w \in \Sigma^*$, $\mathcal{A}[w]$ is the state attained by $\mathcal{A}$ when started in state q_0 with w as input. The concatenation of two words w_0, w_1 can be written as $w_0 + w_1$.

Let T be the target automaton. The algorithm discovers in each phase new states of T with two types of queries:

- Membership query $\text{mq}(w)$: The algorithm asks the oracle, whether a given input word w is accepted by the target automaton T. The oracle answers with yes or no.
- Equivalence query $\text{eq}(H)$: The algorithm submits a hypothetical automaton H to the oracle. If the hypothetical automaton is not equivalent to the target automaton, then the oracle answers with a counterexample c. In case the automata are equivalent, the counterexample is empty.

The counterexample allows the algorithm to discover a new state of $\mathcal{A}$ using membership queries. The hypothetical automaton $H = T$ is equivalent with automaton, when all states of T have been discovered.

In order to save the already learned information the algorithm uses a binary classification tree τ. Each node of the tree contains strings, which can be accessed by the function n.label, where n is a node of the tree. The strings saved in leaves are referred to as *access strings* and those saved in internal nodes are called *distinguishing strings*. Each string $s \in S$ a set of access strings leads from an initial state q_0 to a unique state of the automaton.

A string $d \in D$ is distinguishing, when $\forall(s_0, s_1) \in S | s_0 \neq s_1$ the following condition is true:

$$(\mathcal{A}[s_0 + d] \in F \Leftrightarrow \mathcal{A}[s_1 + d] \notin F) \vee (\mathcal{A}[s_0 + d] \in F \Leftrightarrow \mathcal{A}[s_1 + d] \notin F)$$

The root of the classification tree is a node with an empty distinguishing string. The tree has a recursive structure. Let d be a distinguishing string in a current node. The access strings s is saved on the right part of the tree, if $s + d$ leads to an accepted state of the automaton. Otherwise s is saved on the left side of the tree.

If two strings s_0 and s_1 have the same path by sifting in a classification tree, then the states of automaton $q_1 = T[s_0]$ and $q_2 = H[s_1]$ belong to the same equivalence class. The Listing 2.1 describes the sifting function. This function sift(s) takes a string s as a parameter and starts its work at the root of the classification tree. If the current node n is an internal node, then the next step is a membership query mq$(s(n.\text{label}))$. If the answer to the membership query is yes, the sifting continues in the right subtree, otherwise in the left subtree. This is repeated until the leaf is reached. The acess string stored in the leaf is a representative element for the equivalence class. The partition of the target automaton in equivalence classes can be saved in the classification tree. In each iteration of the algorithm one of the equivalence classes splits until every equivalence class contains only one state.

```
String Sift(String s) {
  CurrentNode=treeRoot;
  String d="";
  while (! CurrentNode.isLeaf){
    if (mq(s+d))
      CurrentNode = CurrentNode.getRight;
        else
          CurrentNode = CurrentNode.getLeft;
    d = CurrentNode.Label;
  }
  return CurrentNode.Label
}
```

Listing 2.1: Sifting the Classification Tree

The hypothetical automaton is constructed from the classification tree by means of sifting as shown in Listing 2.2. Each known equivalence class of the target automaton corresponds to a state of the hypothetical automaton, which is marked with an acess string to a representative element of this equivalence class. The transitions of the hypothetical automaton lead from one equivalence class to another. Then the sift$(s + b)|b \in \Sigma$ returns a destination state of the δ transition out of the state s.

```
Tentative_Hypothesis(H, ClassificationTree) {
  H.crear();
  treeIterator= ClassificationTree.vertexSet().iterator();
  while (treeIterator.hasNext()){
    treeNode=treeIterator.next();
    if (treeNode.isLeaf)
      H.add(treeNode);
    }
  }

  HIterator= H.vertexSet().iterator();
  while (HIterator.hasNext()){
    from=HIterator.next();
    for (int b=0;b<2;b++){
      to=Sift(from.Label+b);
      G.addEdge(from, to).setName(b);
    }
  }
}
```

Listing 2.2: Tentative Hypothesis

After the construction of the hypothetical automaton H the algorithm makes an equivalence query eq(H). The query returns a counterexample c in case $T \neq H$, i.e. the target automaton and the hypothetical automaton differ. This counterexample is used in order to split one of the known equivalence states. For that purpose the work of both automaton is simulated on the counterexample c.

The algorithm ensures that the root node of the tree is always labelled with an empty acess string. That is why the automata have the same initial state and are synchronised at the start. At the end of the processing of c the automata are in the different equivalence classes. The reason for that is the fact that one of them accepts the counterexample and the other rejects it.

The function updateTree(c) (see Listing 2.3) searches for the smallest prefix of the counterexample c that leads the hypothetical automaton and the target automaton to two different equivalence classes. At that point the transitions of the automata must start from two different states of the same equivalence class. Therefore the previous equivalence class can be split in two different equivalence classes. The new equivalence class (leaf of the classification tree) is labelled with a previous prefix as acess string.

```
private  void Update_Tree(String y){
  for(i=1; i<=y.length; i++){
    yi=y.substring(0, i);
    sj = Sift(yi);
    smj = M[yi].Label;
    if (sj != smj) break;
  }

  yi = y.substring(0, j-1);
  si = Sift(yi);

  replace_node = ClassificationTree.getNodeWithAccessString(si);
  d = FindDistinguishingString(sj, smj);

  ClassificationTree.Add(newVertex);
  newVertex.Label = y.substring(j-1,j)+d;
  ClassificationTree.Add(newLeaf);
  newLeaf.Label = y.substring(0, j-1);

  //Connect new vertices
  ClassificationTree.exchange(replace_node, new_vertex);
  if (mqu(newLeaf.Label+newVertex.Label)) {
    T.addRightEdge(newVertex, newLeaf);
    T.addLeftEdge(newVertex, replace_node);
  } else{
    T.addLeftEdge(new_vertex, new_leaf);
    T.addRightEdge(new_vertex, replace_node);
  }
}
```

Listing 2.3: Update the Classification Tree

2.3 Conformance Test of Protocols

A conformance test is performed in order to verify if the external behaviour of a given implementation of a protocol is equivalent to its formal specification (Holz 91). In case the specification has a design error, the implementation passes the test only if it contains the same design error as its formal specification. It fails the test, if the specification and implementation are not equivalent.

A black box approach is widely used for conformance testing. The black box is provided with sequences of input signals. After that the output signals are compared with those prescribed by the formal specification. If they match, the *Im-*

plementation Under Test (IUT) passes the test. All the sequences of input signals together make a conformance test suite.

It is difficult to find a generally applicable, efficient procedure for generating test suites for the protocol implementation.

The first attempts to solve this problem aimed to establish that a given implementation realizes all functions of the specification over the full range of parameter values, and can correctly reject invalid inputs in a way that is consistent with the initial specification. But the problems with the complexity of conducting the tests and their standartisation as well as an uncertainty of their value, made clear that an alternative approach is needed.

The new goal is to establish that the control structure conforms to the specification structure. These structures are conform if they model the same sets of states and allow for the same state transitions.

The structural testing is focused on the control structure of the protocol; no data or parameter values are considered. The following assumptions are made (Holz 91):

1. The protocol represents a deterministic finite state machine; both input and output signals and a maximum number of states are known.

2. The response time to an input signal is a known, finite amount of time

3. The states and transitions of the protocol form a strongly connected graph.

These assumptions are required for testing protocols. They are also essential for this thesis since they allow to consider protocols as finite automata.

3 Adaptation of the Learning Algorithm

In this chapter the algorithm of Angluin is modified in order to answer the purposes of the thesis. The membership and equivalence queries were redefined. The automatic coding and decoding of the test suite was added.

3.1 Realisation of Queries

The algorithm of Angluin assumes the existence of an oracle that answers both membership queries and equivalence queries. In this work the test suite undertakes the functions of oracle.

First, the realisation of membership queries is described. Each proper test case t contains:

1. t.input - inputs, which are needed to conduct the test,
2. and t.reaction - an expected reaction of the test object. t.reaction can be either pass or fail. Pass is equivalent to an accepted state in the automaton theory and fail is equivalent to a rejected state accordingly.

The parameter of the membership query - a sequence of incoming signals - corresponds to a first component of the test case. The answer to the query is a real reaction of the automaton, which corresponds to the second component of the test case.

Based on this information the membership query was implemented as follows: Let assume that a parameter of the membership query is a sequence q. The answer to the query membership query is t.reaction , if t.input$=q$. See Listing 3.1.

But what if the test case t is not contained in the test suite. This question cannot be answered within this work and is a field for further research. The number of accepted sequences in the real-life systems is lower than a number of rejected sequences. Because of the higher probability of getting a rejected sequence an assumption is made that the target automaton rejects those sequences of input signals.

The realisation of equivalence queries is comparatively simple. In the test suite we look for a test case t, in which t.reaction$\neq H[t$.input$]$, where H is a hypothesis automaton learned until the moment, when the query is made. If we find such a

```
boolean membershipQuery(query){
  for (iterator = TestSuite.iterator(); iterator.hasNext();) {
    TestCase = iterator.next();
    if (TestCase.input == query){
      return TestCase.reaction;
    }
  }
  return false;
}
```

Listing 3.1: Membership Query

test case, then equivalence query returns t.input as a counterexample, otherwise we assume that the target automaton and the hypothesis automaton are equivalent. See Listing 3.2.

```
String equivalenceQuery(){
  for (iterator = TestSuite.iterator(); iterator.hasNext();) {
    TestCase = iterator.next();
    if (!traceInHypothesis(TestCase.input) == TestCase.reaction)
      return TestCase.input;
  }
  return "Stop";
}
```

Listing 3.2: Equivalence Query

3.2 Coding

Kearns describes the algorithm of Angluin, which can only learn automata with a binary input alphabet Σ. (Kear 94) The real systems use generally input alphabets with a volume larger than this of the binary alphabet. In order to solve this problem is the transformation of the target automaton T into an equivalent binary automaton T_b needed. For that purpose the incoming signals of the target automation are coded binary. After that the number of states of the automaton increases. The factor of increase depends on the selected code. In order to learn each new state, the algorithm has to make a new iteration. That leads to an increased number of membership queries. Therefore a code that allows us to keep the number of states low is the best choice for the task. The coding theory names this type of code the entropy encoding. The main idea of the entropy encoding is that frequent messages get a shorter code than the less frequent ones. For that purpose coding

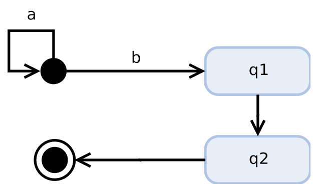


Figure 3.1: Sample Graph 1

algorithms use probabilities of messages p_m, where $\mathsf{m} \in \Sigma$.

In our case the target automaton is unknown. The only information available are the test runs. Therefore, the next step is to try to determine the probabilities p_m based on the information from the test runs. An assumption is made that the probability of getting a message m in the test run is the same as this in the target automation.

Let $\mathcal{A}$ be a known automation, which accepts all sequences of messages starting with an arbitrary amount of messages a and ending with a sequence of messages bcd. Figure 3.2 depicts the automaton $\mathcal{A}$. Then one of the possible test suites can be as follows:

Test Case	Expected result
ab	accept
aac	accept
aaaad	accept

Table 3.1: One of the Possible Test Suites the Automaton $\mathcal{A}$

The probabilities calculated on the base of this test suite are shown in Table 3.2

Message	Probability
a	0.7
b	0.1
c	0.1
d	0.1

Table 3.2: Probabilities of the Messages for the Automaton $\mathcal{A}$

There is an equal probability to get a message b, c or d. This probability is lower than a probability to get a message a. Knowing the probabilities makes the ap-

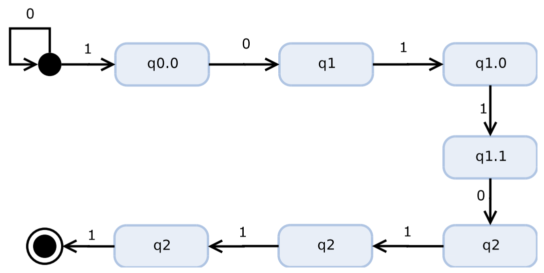


Figure 3.2: Sample Graph 1 Encodecd with Hufman Code

plication of one of the entropy coding techniques a method of Hufman (Huff 08) possible. The results are shown in Table 3.3.

Message	Code
a	0
b	10
c	110
d	111

Table 3.3: Hufman Code for the Messages of the Automaton $\mathcal{A}$

After the coding is applied to the automation $\mathcal{A}$, the number of all states in a binary automaton $\mathcal{A}_b$ will be nine (see Figure 3.2).

Even so this code leads to a minimal length of test runs; the size of the binary automaton is not minimal. So the code shown in Table 3.4 leads to a binary automaton with seven states (see Figure 3.3).

Message	Code
a	00
b	01
c	10
d	11

Table 3.4: Simple Code for the Messages of the Automaton $\mathcal{A}$

The reason for this result is the fact that a set of all messages *a* from the test runs corresponds to just one self-looped transaction in the automaton $\mathcal{A}$. Therefore this

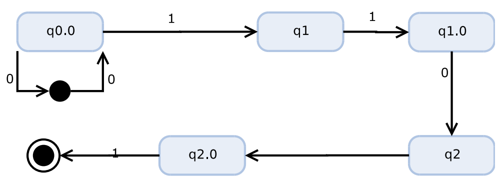


Figure 3.3: Sample Graph 1 Encoded with Simple Code

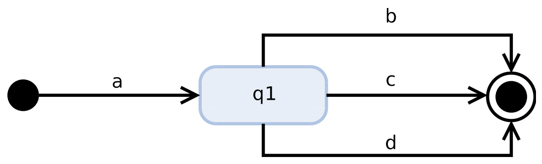


Figure 3.4: Sample Graph 2

approach does not work correctly, if the automation contains any loops.

Next, a performance of this coding in an average case could be explored. But even if the results would be positive that is not the only problem of this approach. Let the automaton $\mathcal{A}'$ accept sequences of messages ab, ac, and ad. At the same time these sequences are test cases of the automation. The automation is depicted in Figure 3.4

Each test run has a mutual path in the automation (here the transition a). That is why a set of all messages a from the test suite is represented in automation by just one transition. The coding on the base of the test suite leads to the same code as in the previous case. Therefore this type of coding is not suitable for cases, where a structure of the automaton is unknown.

After these experiments the decision was made to use a simple binary coding with a constant length $l = \log_2(\Sigma)$ of the words to be coded. The automaton grows in l times. The messages are enumerated with a binary numeral system, which starts with null. A binary number of the message is its code. If a length of the sequence can't be divided by l, it can't be decoded and consequently does not lead to an accepted state. Therefore this type of coding allows us to answer most membership queries effectively.

4 Implementation

The learning algorithm was implemented in Java 5.0. This chapter describes the technical details of the implementation, the file format and the choice of additional Java libraries. Two different file formats are supported. One of them was developed for this application and another one is an *eXtensible Markup Language* (XML) format used in the *TTCN-3 Refactoring and Metrics Tool* (TRex).

TRex is an open-source Eclipse plug-in, which provides *Integrated Development Environment* (IDE) functionality for the *Testing and Test Control Notation Version 3* (TTCN-3) core notation and supports the assessment and automatic restructuring of TTCN-3 test suites by providing suitable metrics and refactoring (TRex 08). For the XML Schema please see Appendix 7.1.

4.1 XML Overview

The language Java has a great variety of *Application Programming Interface*s (APIs) and parsers for the work with XML. Therefore the choice was not easy. There are two basic types of APIs: Streaming and Tree-based APIs.

4.1.1 Streaming API

The XML are transmitted and parsed serially at an application runtime; often in real time and from dynamic sources with contents unknown in advance. The parsers are able to start the generation of output immediately and XML can be discarded right after the use.

The main advantages of a streaming API are a smaller memory footprint, comparatively low processor requirements, and higher performance under certain circumstances.

The main disadvantage is a so called 'cardboard tube' view of a document, which means that the state can be seen only at one location at a time in the document. Therefore a streaming API is a good choice, if the application has strict memory limitations, for example a cell phone running *Java 2 Micro Edition* (J2ME), or if it needs to simultaneously process several requests.

There are two types of streaming APIs: pull and push streaming APIs.

During a pull streaming a client application calls methods on an XML parsing library, when it needs to interact with an XML. In other words the client application only gets (pulls) XML data, if it directly asks for it. On the contrary in a push streaming the XML parser sends (pushes) the XML data to the client as the parser encounters elements in an XML, regardless whether or not the client is ready to use the data at that particular time. A brief overview of the representatives of these two types of streaming APIs follows.

The *Streaming API for XML* (StAX) is a representative of the pull streaming API. StAX combines the two types of pull APIs. The first mode StAX Cursor stands for pure streaming with fast cursor and the second one StAX Iterator for creating XML event objects. It has two major advantages. StAX enables an XML-view of non-XML data. StAX allows also to filter XML documents for information relevant to the client.

A representative of the push streaming API is the *Simple API for XML* (SAX). It models the parser and not the document itself. The parser uses a callback interface in order to feed contents to the client application. Because of that, SAX is very fast and memory efficient. That means that it does not need to store the entire document in the memory and therefore can work efficiently with big XML documents. The developer of a SAX program uses additional data structures to keep the contents and the document separate. This additional work needed can be a disadvantage. Even so SAX is a good choice, if the information you process is close together in the document and you process for instance one element at a time, it is not suitable in case the access to the entire document at once is needed. For that purpose are tree-based APIs a better choice.

4.1.2 Tree-based (*Document Object Model*s)

APIs of this type create in-memory objects representing the XML document as a tree. After that the trees can be navigated and parsed arbitrarily. This leads to more flexibility in the work of developers on one side and to increased processor and memory requirements on the other side.

4.1.2.1 DOM

The *Document Object Model* (DOM) is a representative of the tree-oriented APIs. XML documents are represented as Document objects. The information is managed by invoking methods on the Document object and other objects it contains. It is an advantage, when a random access to widely separated parts of the original XML document is needed. A high memory requirement is a price for this advantage.

DOM is also a read-write API. That means it can not only parse XML documents, but also create them. Well-formed documents are the result.

4.1.2.2 JDOM

Another representative of the tree-based APIs is the Java-native tree-based API JDOM.

There are several similarities between JDOM and DOM, e.g. JDOM reads the XML document into the memory before it starts working. They also have similar program outlines.

However there are some differences in favour of JDOM. The reason for the complexity of the DOM and difficulties in its use is the fact that it is not actually a Java API. Instead it is defined in the *Interface Definition Language* (IDL) and then compiled to Java. Because of that DOM uses interfaces and factory methods instead of classes and constructors.

On the contrary JDOM was designed just for Java. It uses some standard Java coding methods, like `equals()` and `hashCode()`, and classes as `List` and `String`. All of that makes JDOM more intuitive for Java developers and its code less tricky compared to that of the DOM.

JDOM performs best on simple documents with no recursion, limited mixed content, and a well-known vocabulary. The XML side of the JDOM is not optimal. It doesn't have some important features like a common node interface or superclass for navigation. That is why JDOM has some problems by processing arbitrary XML.

The Table 4.1 shows the overview of APIs and their features.

The choice of APIs was based on the following criteria. In the current program XML is used for both input and output of the data. The streaming APIs can only generally read XML. Even though StAX has a feature of saving XML documents, it does not provide an interface for creating nodes. Of course different APIs could be used for saving and reading data, but this way is unnecessary complicated. That is why the idea of using the streaming APIs was rejected. The advantage of DOM is that it has more functions for working with XML documents as the other APIs. The XML files used in the program have a relative simple structure. So that the functions the JDOM provides are sufficient for our purposes. Using JDOM instead of DOM brings additional advantages, such as intuitive development and moderate resource requirements. Therefore JDOM was chosen as an XML API.

Feature	StAX	SAX	DOM	JDOM
API Type	Pull, streaming	Push, streaming	In memory tree	In memory tree
Ease of Use	High	Medium	High	Very High
XPath Capability	Not supported	Not supported	Supported	Supported
Efficiency	Very Good	Very Good	Medium	Good
Arbitrarily Navigation	Not supported	Not supported	Supported	Supported
Read XML	Supported	Supported	Supported	Supported
Write XML	Supported	Not supported	Supported	Supported
Create, Update, Delete Nodes	Not supported	Not supported	Supported	Supported

Table 4.1: XML API Feature Summary

4.2 Description of the XML Format

4.2.1 Input File Format

Test cases can be loaded into an application from an XML file, which is described by the *XML Schema Definition* (XSD) Schema *Labeled Transition System Markup Language* (LTSML). This format is pretty general and is designed to model labeled transition systems. That is why just a part of all available tags of this format will be used. Some restrictions were also made to the rules described in LTSML. The following text describes the details.

Test cases can be presented in form of a graph G_t. The initial state of the graph G_t is a starting point of each test case. Every transition is one test step of the test case. The test case ends, when the final state of the graph G_t is reached. The LTSML format does not allow to mark states as positive (accepted) or negative (rejected). To solve this problem test cases are stgored in two *Labeled Transition System* (LTS) structures. The positive test cases are stored in the first one and the second one contains negative tests. Both LTS have a similar structure. The example of Inres is used in order to describe this structure.

The tag action defines those input signals, which are used in test cases. The attribute id is a unique name of the signal. Even so this XML file has to contain the attribute type and the tag description, they are ignored during processing.

The states of the graph are used as connection points between two steps of the test case. That is why they need to be defined. For that purpose the tag state is used.

Next, each test case is defined by means of tag transition as a sequence of test steps. Every test case starts at a state marked with the tag startState. The end of a test case is marked with the tag endStates. The application analyses the document by searching for all paths between the start and the end state. It begins the search from a startState and proceeds until it reaches a state, where no output transitions are possible. Therefore it is important to develop an XML document without any loops.

The structure of the negative tests is almost the same except that it doesn't need a definition of actions. The definitions made for the positive tests can be used here.

The class TrexTracesReader enables the reading of this file format. For details see Figure 4.1.

4.2.2 Output File Format

A LTSML format is also used for saving the already learned automata. No restrictions described in Section 4.2.1 are necessary for saving of automata. First, the states Q of automaton are saved in an XML file. The attribute id and the tag description are set to a corresponding acess string. Consequently a state with id = "" (empty) is a startState and there are no endstates.

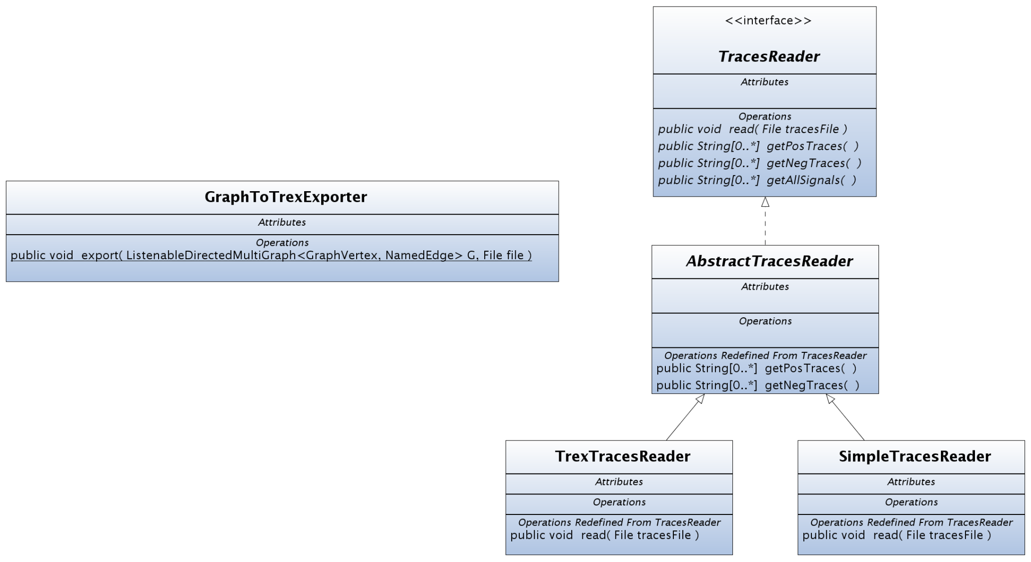
Every signal $\in \Sigma$ is saved in a separate tag action. The transition function is stored in a structure transitions. An attribute id of each transition is generated automatically.

The class GraphToTrexExporter was implemented for export of the learned automaton into LTSML format For details see Figure 4.1.

4.3 Simple Traces Format

The implemented system also supports the *Simple Traces Format*. This is a kind of a *Comma Separated Values* (CSV) format. Each line of the file contains one test case. In the first column stands a plus sign for a positive test case and a minus sign for a negative test case. The following columns contain input signals. The columns are separated with commas.

In comparison with an XML format this format has two main advantages. The files are more readable for the use and are also smaller as those in the XML format described in Section 4.2.1. The main disadvantage is the fact that the use of this format is possible only for this application. On the other hand one format can be easily mapped into another format The class SimpleTracesReader, which is shown in Figure 4.1, enables the support of the Simple Traces Format.



<<interface>>
TracesReader
Attributes
Operations
public void read(File tracesFile)
public String[0..*] getPosTraces( )
public String[0..*] getNegTraces( )
public String[0..*] getAllSignals( )
GraphToTrexExporter
Attributes
Operations
public void export(ListenableDirectedMultiGraph<GraphVertex, NamedEdge> G, File file)
AbstractTracesReader
Attributes
Operations
Operations Redefined From TracesReader
public String[0..*] getPosTraces( )
public String[0..*] getNegTraces( )
TrexTracesReader
Attributes
Operations
Operations Redefined From TracesReader
public void read(File tracesFile)
SimpleTracesReader
Attributes
Operations
Operations Redefined From TracesReader
public void read(File tracesFile)

Figure 4.1: Class Diagramm of Readers and Writer

4.4 Implementation of the Learning Algorithm

For the most part the implementation was done according to the description in the Chapter 2 and in the Chapter 3. The class diagram of the learning algorithm is shown in Figure 4.2. The exception was made for the implementation of the membership query. An additional information from decoder is used to answer the membership query.

A binary sequence can't be decoded, if there is no match in the decoding table between this sequence and one of the signals of the target automaton. That means the membership query contains signals that don't belong to a set of input signals of the target automaton. In this case the oracle returns false as an answer. The final structure of the membership query is depicted in Listing 4.1.

4.5 Coder and Decoder

As already mentioned in Section 3.2, the coding and decoding of signals is essential for a successful learning of protocols. For that purpose a class CoderDecoder was created. The class is depicted in *Unified Modeling Language* (UML) notation in Figure 4.3.

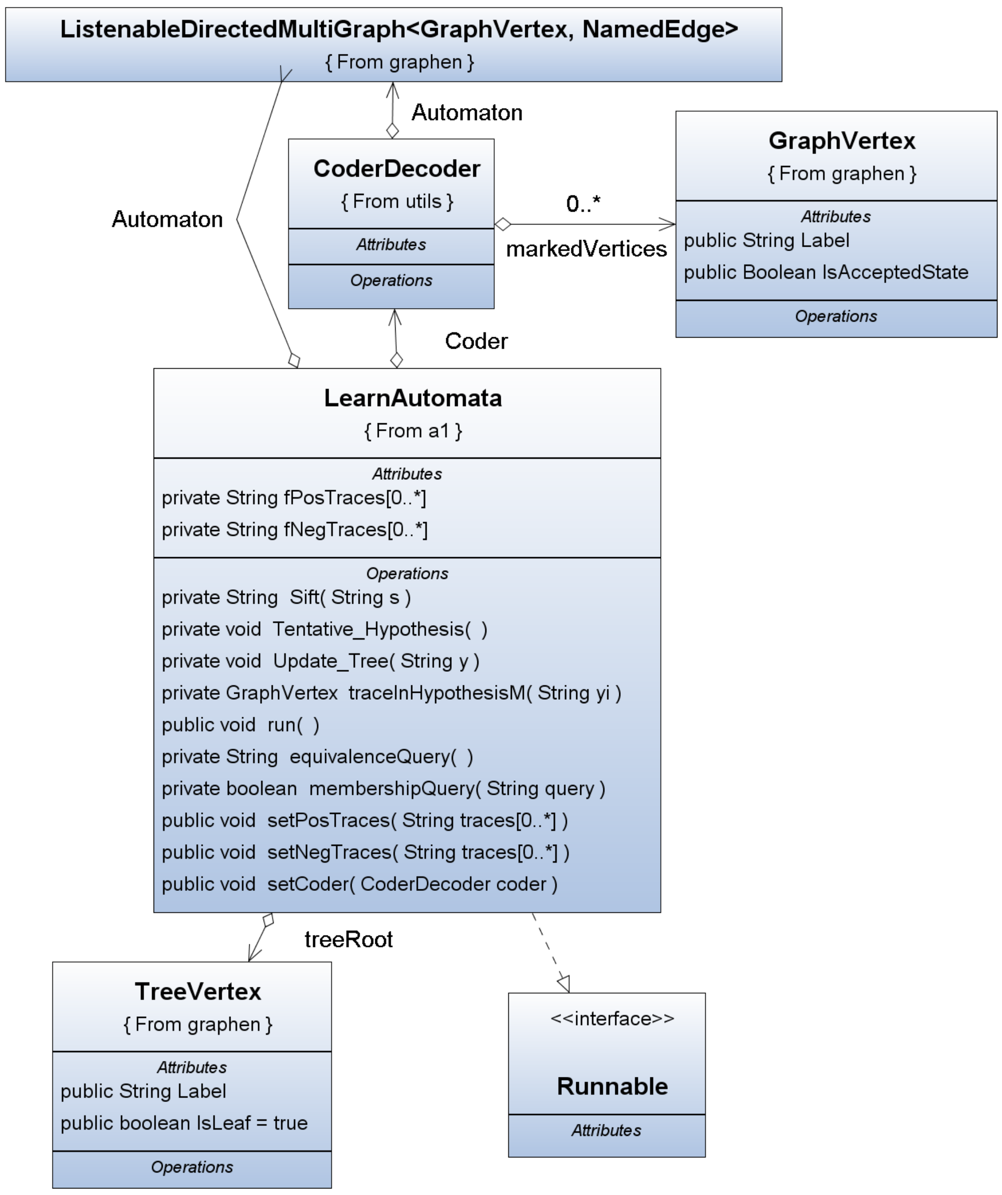

ListenableDirectedMultiGraph<GraphVertex, NamedEdge>
{ From graphen }
Automaton
Automaton
CoderDecoder
{ From utils }
Attributes
Operations
0..*
markedVertices
GraphVertex
{ From graphen }
Attributes
public String Label
public Boolean IsAcceptedState
Operations
Coder
LearnAutomata
{ From a1 }
Attributes
private String fPosTraces[0..*]
private String fNegTraces[0..*]
Operations
private String Sift(String s)
private void Tentative_Hypothesis( )
private void Update_Tree(String y)
private GraphVertex traceInHypothesisM(String yi)
public void run( )
private String equivalenceQuery( )
private boolean membershipQuery(String query)
public void setPosTraces(String traces[0..*])
public void setNegTraces(String traces[0..*])
public void setCoder(CoderDecoder coder)
treeRoot
TreeVertex
{ From graphen }
Attributes
public String Label
public boolean IsLeaf = true
Operations
<<interface>>
Runnable
Attributes

Figure 4.2: Class Diagramm of the Learning Algorithm

```
boolean membershipQuery(query){
      if (!fCoder.canDecode(query))
        return false;

  for (iterator = TestSuite.iterator(); iterator.hasNext();) {
    TestCase = iterator.next();
    if (TestCase.input == query){
      return TestCase.reaction;
    }
  }
  return false;
}
```

Listing 4.1: Implementation of Membership Query

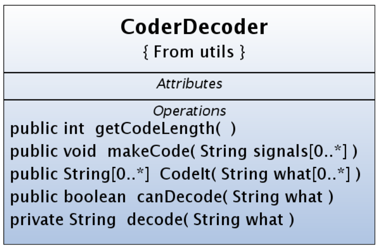

CoderDecoder
{ From utils }
Attributes
Operations
public int getCodeLength( )
public void makeCode(String signals[0..*])
public String[0..*] CodeIt(String what[0..*])
public boolean canDecode(String what)
private String decode(String what)

Figure 4.3: UML Notation of Coder Decoder

The public procedure makeCode and the public function codeIt are responsible for coding. First, a table is built to map a protocol signal with a corresponding code word. A set of all signals is passed as a parameter to the function makeCode, which iterates and numerates them. The binary representation of the number is a code word. Finally, each test case is coded by means of function codeIt.

In the learning process the algorithm makes membership queries. But not all of them can be decoded with CoderDecoder. They can not be decoded only, if a query contains an unknown code word. This kind of query can never lead to an accepted state of the automaton. That is why the function mq uses a function canDecode. This function returns true, if the query can be decoded, otherwise it answers with false.

After the algorithm has finished his work, the learned automaton needs to be decoded. For that purpose the function decodeGraph is used. A high level description of the decoding process is shown in the Listing 4.2. The function manages two lists:

```
decodedGraph decodeGraph(whatG, graphRoot) {
  //initialization
  markedVertices.empty;
  decodedGraph.empty;
  startQueue.empty;

  startQueue.add(graphRoot);

  while (startQueue.size() != 0){
    startFrom = startQueue.iterator().next();
    startQueue.remove(startFrom);
    connectionsLength = getCodeLength();
    findConnection(decodedGraph, startFrom, startFrom, 0, "", startQueue);
    markedVertices.add(startFrom);
  }
  return decodedG;
}
```

Listing 4.2: Function decodeGraph

- A startQueue - a list of vertices, which need to be processed.
- A markedVertices - a list of already processed vertices

The function starts to work with an initial state as a currentVertex. The procedure has to find all paths with a length codeLength starting at the currentVertex. If the vertex at the end of the path is not yet in the list markedVertices, it will be added to the startQueue.

Next, the nodes currentVertex and the vertex at the end of the path are added to the decodedGraph. After that the vertices are connected by the edge labeled with a signal =CoderDecoder.decode(path). Finally, the currentVertex is added to the markedVertices and startQueue.pop is assigned to a currentVertex. The decoding continues until the startQueue is empty.

5 Experiments

In this chapter the development of a test suite and the learning process are discussed and the results of this practical work are revised and evaluated.

5.1 Adaptation of Inres Protocol

As described in Section 2.1 of this Master Thesis, Inres is a complex system. This particular work is concentrated on the learning of the automaton I, which describes the behavior of the Initiator.

- The Input Alphabet of I.

  An input alphabet consists of all incoming messages on ISAPini and MSAP1 (see Figure 2.1) and a timeout message TIMER.

- Initial State

  Although the protocol has neither initial nor final states, the algorithm requires an initial state. That is why the disconnected state is set as an initial state. Because of this all test cases will start in a disconnected state.

- Accepted and Rejected States

  The requirements of the algorithm also include accepted and rejected states. Let the disconnected state be the accepted state and the other states be rejected states. Any test case stopped in a disconnected state is positive and any other test case is negative.

Before starting the experiments the Inres will be simplified. The goal of the modification is to make Inres automat in general and the encoded automat in particular more practical without losing important properties of the protocol. The following simplifications have been made:

In the Data transmission phase (see Section 2.1.3.2) the parameter ISDU is limited to a one constant message. Because of the constant ISDU it is omitted in the following work and for example IDATreq and DT is written instead of IDATreq(constMessage) and DT(constMessage) accordingly . This modification does not restrict the communication process between the Initiator-user and Responder-user in any significant

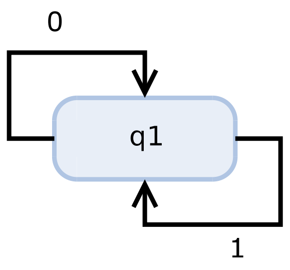


Figure 5.1: Special Vertex

way. The following example illustrates that. The data transmission can be carried out in blocks. Initiator opens a connection and sends n times DT to a Responder. Thereby a binary representation n is equivalent to a binary representation of the block. Responder counts the received messages and presents the results binary. After that the connection stops and a new connection can be established in order to transmit a new block of information. Therefore this simplification is congruent with the goal, not to lose the important functions of the protocol.

Unexpected PDUs and SPs are not ignored by the Initiator, they lead to a special vertex shown in Figure 5.1. The vertex itself has two outgoing edges, which are self-looped. The vertex is in a set of rejecting states of automaton.

In case of a transmission error, the sending is not repeated.

A simplified automaton is depicted in Figure 5.2.

5.2 Test Suite

The algorithm learns from membership and equivalence queries. Both memory and equivalence query are based on the test suite. Therefore the algorithm can learn only information, which is contained in the test cases. That is why, the minimal requirements of the test suite include:

1. All states of the automaton can be reached with the test suite.
2. Each transition of the automaton must be contained in the test suite at least one time

According to these rules, the first test suite was developed. The Table 5.1 shows this test suite.

If the algorithm runs on this test suite, it stops its work after the first equivalence query. The algorithm has created a simple automaton with one state. This state

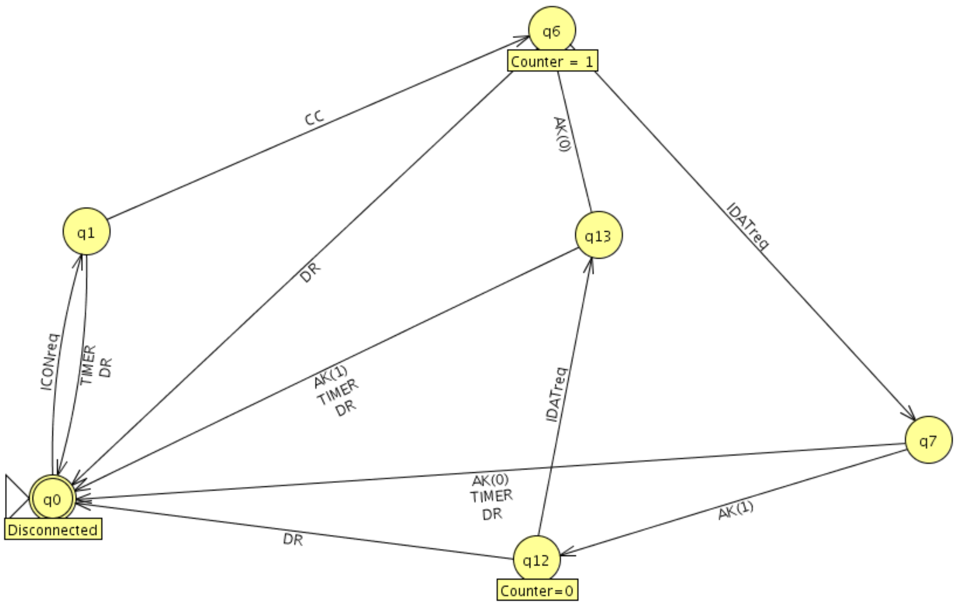


Figure 5.2: Simplified Inres Automaton

Test Case	Expected result
ICONreq, TIMER	accept
ICONreq, DR	accept
ICONreq, CC, DR	accept
ICONreq, CC, IDATreq, DR	accept
ICONreq, CC, IDATreq, TIMER	accept
ICONreq, CC, IDATreq, AK(0)	accept
ICONreq, CC, IDATreq, AK(1), DR	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, DR	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, TIMER	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(1)	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(0), DR	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(0), IDATreq, AK(0)	accept

Table 5.1: Positive Test Cases for the Inres Protocol

is an accepted state, which corresponds to an initial state of the target automaton. The state has two self-looped transitions. Because of that, the equivalence query can not find any difference between test cases and this automaton. Therefore our entire test suite was accepted. Figure 5.3 shows the hypothesis automaton of the first test suit.

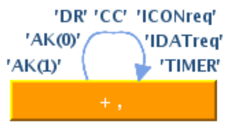


Figure 5.3: Hypothesis Automaton of the First Test Suit

The first test has shown that the algorithm can not learn successfully without negative tests. The next challenge is to determine, which negative test cases and how many of them are needed.

In the next experiment a special version of membership and equivalence queries was used. In this version the operator plays a role of oracle. The operator sees the already learned binary hypothesis automation and can enter as counterexample any sequence of binary signals. That means the operator can enter even a sequence of binary signals, which can not be decoded into a sequence of incoming signals of Inres. Such input is a counterexample, if it leads to an accepted state in the hypothesis automaton. A signal, which can't be decoded, is defined as an unknown signal. With the negative tests shown in Table 5.2 the learning of the automaton was successful. Figure 5.4 shows the binary coded learned graph.

Test Case	Expected result
ICONreq, CC, IDATreq, AK(1), IDATreq, *unknown signal*	reject
IDATreq, AK(1)	reject
ICONreq, CC, IDATreq, CC, DR	reject

Table 5.2: Negative Test Cases for the Inres Protocol

In the previous experiments test suites were in CSV format. In the following experiments an XML format is used. The reason for this change is the compatibility with TRex (details in Chapter 4). In this experiment the equivalence queries are based on the test suite.

Now the coder gets the signals in a different order as in the previous experiments. The algorithm gets other counterexamples as answers to the equivalence

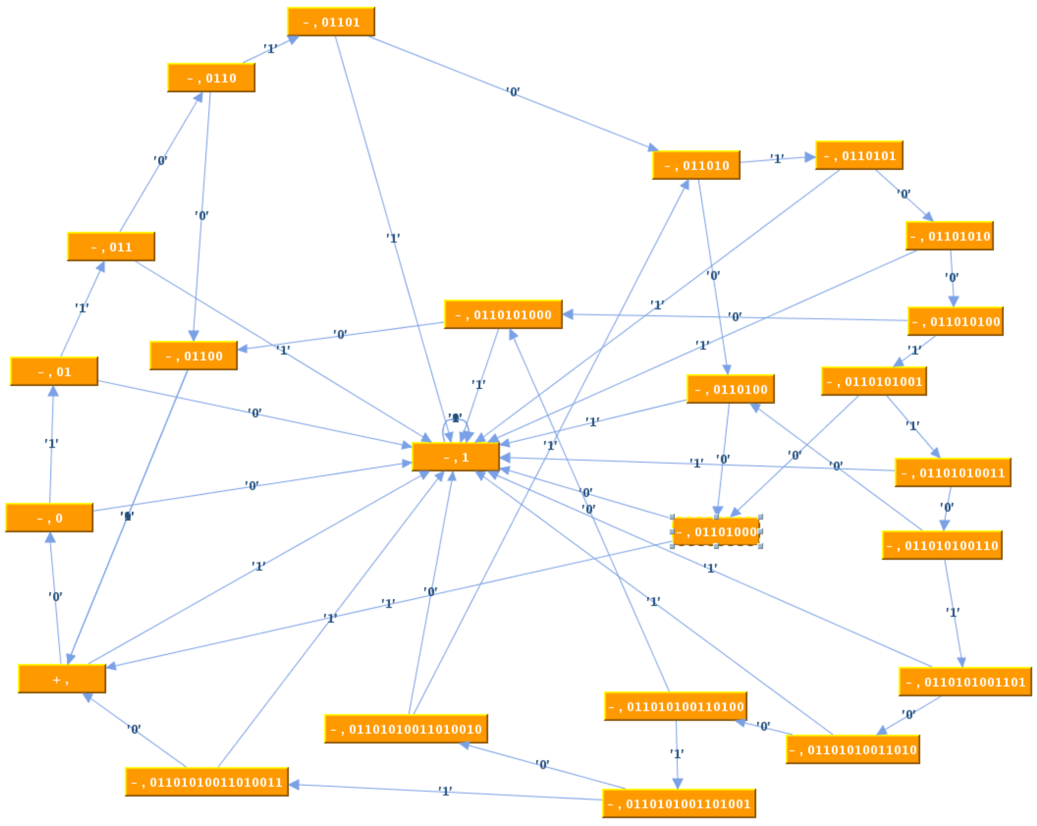


Figure 5.4: Binary Coded Inres Automaton

queries. Because of that the algorithm makes membership quieries, which differ from those in the previous experiments. This code leads to an increased number of test cases compared to the test suites of the previous experiments. Those additional test cases are necessary in order to answer membership queries. The length of the longest test case has also increased. It almost reaches the upper bound of the theoretically sufficient maximum length of the test case. This upper bound is less or equals $2n$, where n is a number of states of the automaton. A parameter of a membership query is a concatenation of a distinguishing string and acess string. The length of each string is bounded in the number of states of the automaton.

During the next experiments the program ran on a variety of combinations of the negative tests. A set of positive tests was constant. In this particular case one negative test was sufficient for the learning.[1] Each of the negative tests in Table 5.2 is suitable for this task. The best choice is a negative test with a minimal length, since in that case, the cumulative length of the generated access and distinguishing strings is smaller than this length in cases with a longer test case. That enables us to keep the length of the membership query small. The final test suite, which complies with all these rules, is shown in Table 5.3.

The result was an automaton equivalent to the reference automaton. Figure 5.5 shows the automaton before it is decoded.

During the tests we noticed one interesting feature of the algorithm: the machine never forgets the information once learned. This fact has both negative and positive effects for our work. On one side we must not search in learned test cases for a counter example in order to answer an equivalence query. On the other side it is not possible to correct the wrong answer. We have to start learning from scratch.

[1]The exact number of the tests should be determined in each case separately.

Test Case	Expected result
ICONreq, TIMER	accept
ICONreq, DR	accept
ICONreq, CC, DR	accept
ICONreq, CC, IDATreq, DR	accept
ICONreq, CC, IDATreq, TIMER	accept
ICONreq, CC, IDATreq, AK(0)	accept
ICONreq, CC, IDATreq, AK(1), DR	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, DR	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, TIMER	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(1)	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(0), DR	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(0), IDATreq, AK(0), IDATreq, TIMER	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(0), IDATreq, AK(0), IDATreq, DR	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(0), IDATreq, AK(0), IDATreq, AK(0), DR	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(0), IDATreq, AK(0), IDATreq, AK(0), IDATreq, DR	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(0), IDATreq, AK(0), IDATreq, AK(0), IDATreq, TIMER	accept
ICONreq, CC, IDATreq, AK(1), IDATreq, AK(0), IDATreq, AK(0), IDATreq, AK(0), IDATreq, AK(1)	accept
IDATreq, AK(1)	reject

Table 5.3: Second Test Suite for the Inres Protocol

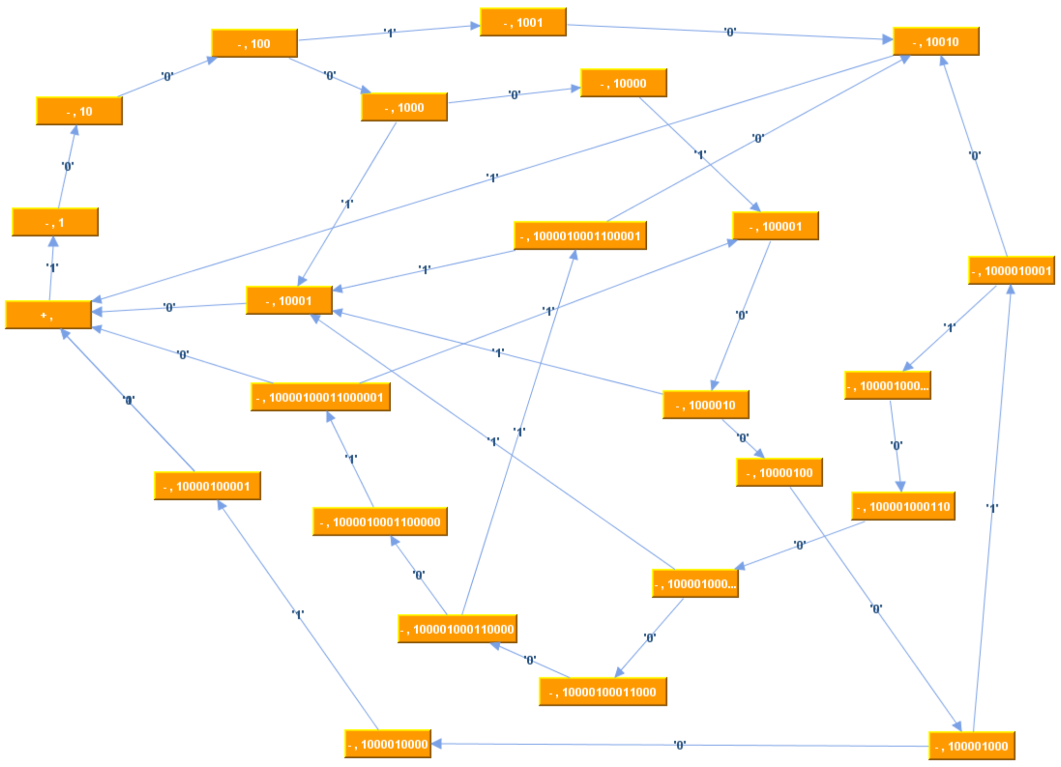


Figure 5.5: Second Binary Coded Inres Automaton

6 Conclusion

This thesis shows a method to create a formal specification from a test suite. The experiments were made on example of the automaton, which represents the initiator entity of a simplified Inres protocol. Both membership and equivalence queries were modified, which allowed the algorithm to learn from a test suite. Additionally the coding pre- and post processing were added to the algorithm so that the algorithm can now learn any protocols from a qualified test suite. The qualified test suite in terms of this thesis is a test suite, which contains unrolled test cases. The loops need to be unrolled until the length of the paths reaches the upper limit $2n$, where n is a number of states in the automaton.

In connection with this work several questions seem to be interesting areas for further research. What to do, when the answer to the membership query is missing in a test suite, is one of these questions. Throughout this thesis are the strategies pursued, which guarantee that this situation does not occur. The consequence of this is a big size of the required test suite. There are two possible solutions:

- Backtracking: The learning algorithm never forgets the already learned information. One wrong answer leads to contradictory results in the learning process. In this case the algorithm has to go back to the decision point, correct errors and continue learning with the right answer.

- Learning algorithm, which works with three types of answers (yes, no, I don't know) Both of these strategies are to be researched in regard of their ability to decrease a test suite.

The sufficient number of negative test cases and an application of the results of this work to an evaluation of the quality of the test suite by comparison of a formal description to an existing specification can also be a subject of further research.

List of Figures

2.1 Inres System 9
2.2 Unsuccessful Transmissions 10

3.1 Sample Graph 1 20
3.2 Sample Graph 1 Encodecd with Hufman Code 21
3.3 Sample Graph 1 Encoded with Simple Code 22
3.4 Sample Graph 2 22

4.1 Class Diagramm of Readers and Writer 28
4.2 Class Diagramm of the Learning Algorithm 29
4.3 UML Notation of Coder Decoder 30

5.1 Special Vertex 34
5.2 Simplified Inres Automaton 35
5.3 Hypothesis Automaton of the First Test Suit 36
5.4 Binary Coded Inres Automaton 37
5.5 Second Binary Coded Inres Automaton 40

List of Tables

2.1 *Service Primitive*s of the Inres Protocol 9
2.2 *Protocol Data Unit*s of the Inres Protocol 11

3.1 One of the Possible Test Suites the Automaton $\mathcal{A}$ 20
3.2 Probabilities of the Messages for the Automaton $\mathcal{A}$ 20
3.3 Hufman Code for the Messages of the Automaton $\mathcal{A}$ 21
3.4 Simple Code for the Messages of the Automaton $\mathcal{A}$ 21

4.1 XML API Feature Summary 26

5.1 Positive Test Cases for the Inres Protocol 35
5.2 Negative Test Cases for the Inres Protocol 36
5.3 Second Test Suite for the Inres Protocol 39

Listings

2.1 Sifting the Classification Tree . . . 14
2.2 Tentative Hypothesis . . . 15
2.3 Update the Classification Tree . . . 16
3.1 Membership Query . . . 19
3.2 Equivalence Query . . . 19
4.1 Implementation of Membership Query . . . 30
4.2 Function decodeGraph . . . 31
7.1 LTSML Shema . . . 45

7 Appendix

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.trex.informatik.uni-goettingen.de/ltsml"
  xmlns:lml="http://www.trex.informatik.uni-goettingen.de/ltsml"
          elementFormDefault="qualified">

  <xs:complexType name="State">
  <xs:all>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="propositions" type="lml:Propositions" minOccurs="0"/>
  </xs:all>
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="color" type="xs:string" use="optional"/>
  </xs:complexType>

  <xs:complexType name="States">
  <xs:sequence>
    <xs:element name="state" type="lml:State" maxOccurs="unbounded"/>
  </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Action">
  <xs:all>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="lineNumber" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="changeVariable" type="lml:ChangeVariable"
                minOccurs="0"/>
  </xs:all>
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="portRef" type="xs:string" use="optional"/>
  <xs:attribute name="type" use="optional">
    <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="input"/>
      <xs:enumeration value="output"/>
      <xs:enumeration value="tau"/>
      <xs:enumeration value="other"/>
    </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
```

```
<xs:attribute name="color" type="xs:string" use="optional"/>
</xs:complexType>

<xs:complexType name="Actions">
<xs:sequence>
  <xs:element name="action" type="lml:Action" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
  // force continuation on next page \clearpage
<xs:complexType name="Transition">
<xs:attribute name="id" type="xs:string" use="required"/>
<xs:attribute name="actionRef" type="xs:string" use="required"/>
<xs:attribute name="sourceRef" type="xs:string" use="required"/>
<xs:attribute name="targetRef" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Transitions">
<xs:sequence>
  <xs:element name="transition" type="lml:Transition" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="StartState">
<xs:attribute name="stateRef" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Lts">
<xs:all>
  <xs:element name="variables" type="lml:Variables" minOccurs="0"/>
  <xs:element name="states" type="lml:States"/>
  <xs:element name="actions" type="lml:Actions" minOccurs="0"/>
  <xs:element name="ports" type="lml:Ports" minOccurs="0"/>
  <xs:element name="transitions" type="lml:Transitions" minOccurs="0"/>
  <xs:element name="startState" type="lml:StartState"/>
  <xs:element name="endStates" type="lml:EndStates" minOccurs="0"/>
</xs:all>
<xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Ltsml">
<xs:sequence>
  <xs:element name="lts" type="lml:Lts" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:element name="ltsml" type="lml:Ltsml">

<xs:key name="StateKey">
  <xs:selector xpath=".//lml:state"/>
  <xs:field xpath="@id"/>
</xs:key>
<xs:keyref name="SourceKeyRef" refer="lml:StateKey">
  <xs:selector xpath=".//lml:transition"/>
  <xs:field xpath="@sourceRef"/>
</xs:keyref>
<xs:keyref name="TargetKeyRef" refer="lml:StateKey">
  <xs:selector xpath=".//lml:transition"/>
  <xs:field xpath="@targetRef"/>
</xs:keyref>
<xs:keyref name="StartStateKeyRef" refer="lml:StateKey">
  <xs:selector xpath=".//lml:startState"/>
  <xs:field xpath="@stateRef"/>
</xs:keyref>
<xs:keyref name="EndStateKeyRef" refer="lml:StateKey">
  <xs:selector xpath=".//lml:endState"/>
  <xs:field xpath="@stateRef"/>
</xs:keyref>

<xs:key name="ActionKey">
  <xs:selector xpath=".//lml:action"/>
  <xs:field xpath="@id"/>
</xs:key>
<xs:keyref name="ActionKeyRef" refer="lml:ActionKey">
  <xs:selector xpath=".//lml:transition"/>
  <xs:field xpath="@actionRef"/>
</xs:keyref>

<xs:key name="PortKey">
  <xs:selector xpath=".//lml:port"></xs:selector>
  <xs:field xpath="@id"></xs:field>
</xs:key>
<xs:keyref name="PortKeyRef" refer="lml:PortKey">
  <xs:selector xpath=".//lml:action"/>
  <xs:field xpath="@portRef"/>
</xs:keyref>

<xs:key name="VarKey">
  <xs:selector xpath=".//lml:variable"/>
  <xs:field xpath="@id"/>
</xs:key>
<xs:keyref refer="lml:VarKey" name="VarKeyRef">
  <xs:selector xpath=".//lml:changeVariable"/>
  <xs:field xpath="@varRef"/>
</xs:keyref>

</xs:element>
```

```
<!-- optional types/elements for Kripke structures -->

<xs:complexType name="Proposition">
<xs:simpleContent>
  <xs:extension base="xs:boolean">
  <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:extension>
</xs:simpleContent>
</xs:complexType>

<xs:complexType name="Propositions">
<xs:sequence>
  <xs:element name="proposition" type="lml:Proposition" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="EndState">
<xs:attribute name="stateRef" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="EndStates">
<xs:sequence>
  <xs:element name="endState" type="lml:EndState" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="Port">
<xs:sequence>
  <xs:element name="description" type="xs:string"/>
</xs:sequence>
<xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Ports">
<xs:sequence>
  <xs:element name="port" type="lml:Port" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="Variable">
<xs:attribute name="id" type="xs:string" use="required"/>
<xs:attribute name="name" type="xs:string" use="required"/>
<xs:attribute name="init" type="xs:boolean" default="false" use="optional"/>
</xs:complexType>

<xs:complexType name="Variables">
<xs:sequence>
  <xs:element name="variable" type="lml:Variable" maxOccurs="unbounded"/>
</xs:sequence>
```

```
  </xs:complexType>

  <xs:complexType name="ChangeVariable">
  <xs:attribute name="varRef" type="xs:string" use="required"/>
  <xs:attribute name="value" type="xs:boolean" use="required"/>
  </xs:complexType>

</xs:schema>
```

Listing 7.1: LTSML Shema

Bibliography

[Angl 87] D. Angluin. "Learning regular sets from queries and counterexamples". *Inf. Comput.*, Vol. 75, No. 2, pp. 87–106, 1987.

[Hogr 89] D. Hogrefe. *Estelle, Lotos und SDL*. Springer, 1989.

[Hogr 91] D. Hogrefe. "OSI Formal Specification Case Study: The INRES Protocol and Service". Tech. Rep. IAM-91-012, University of Berne, Institute for Informatics and Applied Mathematics, May 1991.

[Holz 91] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Huff 08] "Huffman coding". `http://en.wikipedia.org/wiki/Huffman_coding`, Apr. 2008.

[Kear 94] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994.

[Neuk 04] H. Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, Dissertation, Universität Göttingen, November 2004 (electronically published on http://webdoc.sub.gwdg.de/diss/2004/neukirchen/index.html and archived on http://deposit.ddb.de/cgi-bin/dokserv?idn=974026611 . Persistent Identifier: urn:nbn:de:gbv:7-webdoc-300-2), Nov. 2004.

[Somm 06] I. Sommerville. *Software Engineering:(Update) 8th Edition - Cased*. Addison-Wesley, Juni 2006.

[TRex 08] "TRex". `http://www.trex.informatik.uni-goettingen.de/`, Apr. 2008.