A Case Study on Automated Generation of Integration Tests

Giuseppe Della Penna, Alberto Tofani
Università di L'Aquila
Coppito, L'Aquila, Italy

Marcello Pecorari,Orazio Raparelli
Technolab s.r.l.
L'Aquila, Italy

Igor Melatti, Enrico Tronci
Università di Roma "La Sapienza"
Rome, Italy

Benedetto Intrigila
Università di Roma "Tor Vergata"
Roma, Italy

Abstract

We present a case study on the integration in a real industrial design flow of *Automatic Integration Test Generation methods*.

We achieved our goal by devising a testing plan specification language to easily define sets of test cases. The key idea was to base our language on formal standard-based system specification languages already used by the testing engineers in the company. We then have developed the software to integrate our approach in the design flow. From our testing plan specifications we can then generate all the test cases. We believe this approach can be reused also in different contexts.

Our framework meets the initial requirements, i.e. no additional skills to the integration testing engineers are required, and only open source software is used. Moreover, during our preliminary experiments we were able to catch errors which went previously undetected.

1 Introduction

We present a case study on the integration in a real industrial design flow of *Automatic Integration Test Generation methods*.

Our goal is to automatically generate integration tests for a class of telecommunication embedded systems, namely network hardware devices with their controlling software. The goal of the integration testing phase [Mar03, Som01] is to make sure that the hardware and the software properly work together.

In the following, we will refer to the whole system, composed by the hardware and the controlling software, as *Network Element* (NE for short). NE is a "real world" system, actually designed at TechnoLab, a company operating in the telecommunication field.

To carry out the integration testing phase two steps are needed: 1. define a set of *interesting* tests cases 2. send the test cases to the NE.

The integration testing phase, as implemented in the company design flow, is performed *manually* and without a formal definition of the set of test cases to be send to the NE. This causes delay in the deployment of the NE, and thus increases the time-to-market. Moreover, interesting *test cases* might be skipped, forcing to expensive late corrections on the NE.

In order to solve the above mentioned problems *automatic integration test generation* has been investigated. As it turns out, the introduction in the company design flow of an automatic approach must meet the following constraints:

- The actual design flow should not be altered too much;
- No heavy training should be needed for the test engineers to use the automatic approach;
- Open source software must be used.

To achieve our goal, we take advantage of the software development process adopted in TechnoLab. In fact, the NE is developed according to the standardized Telecommunication Management Network (TMN [TMN06, Bla94, Sid98]) framework.

In particular, for each NE submodule S the company design flow makes available a formal description of S based on standard formal languages (namely GDMO and ASN.1, see Section 2). Moreover testing engineers are familiar with such formal languages.

Exploiting the available information (GDMO, ASN.1 definitions of NE submodules) and testing engineers expertise on GDMO and ASN.1 languages we were able to devise the following approach.

First, define a *testing plan specification language* which is as close as possible to the GDMO, ASN.1 languages already known to the company test engineers. This avoids retraining.

Second, exploits GDMO, ASN.1 specifications for NE submodules to generate tests cases. We can do this by developing a suitable software that from the test plan definition and GDMO, ASN.1 definitions of NE generates all test cases. In this way we use open source software (our software) and avoid changing the actual design flow.

Note that a similar result could also be obtained by using tools based on the Testing and Test Control Notation (TTCN-3 [TTC06, GHR+03, Wil01]) standard. However, in our context, this approach has at least two drawbacks.

First, the deployment of any TTCN-based test system require a considerable initial effort in term of human resources in order to learn how to use and deploy such systems. This is incompatible with our requirement asking for (almost) no retraining.

Second, many vendors provide complete (compiler, parser and run time environment) TTCN-3 test systems, however, to the best of our knowledge, no open source complete distributions exists. This is incompatible with our requirement asking for open source software.

Summing up, following our approach we have obtained a light weighted tool which has a limited cost, and is quite easy to integrate with the existing design flow. Moreover our approach, may be used also by other companies, which adopt the TMN framework (see Section 2).

We implemented a prototype of our framework, and tested it on a submodule of an important NE at TechnoLab. We were able to discover errors which went previously undetected.

2 The GDMO and ASN.1 Standards

Generally speaking, GDMO and ASN.1 are substandards of the TMN framework [TMN06, Bla94, Sid98]. TMN is used for achieving interconnectivity and communication across heterogeneous operating systems and telecommunication networks. The TMN framework relies on various Open Systems Interconnection (OSI [OSI06]) standards: the Common Management Information Protocol (CMIP [ICM06]) that defines management services exchanged between peer entities, the Guidelines for the Definition of Managed Objects (GDMO [ITU06, Heb95, Udu99]) that provides templates for classifying and describing managed resources, the Abstract Syntax Notation number One (ASN.1 [ASN06, Udu99, Dub00]) that provides the syntax rules for data types and, finally, the open systems reference model (the seven-layer OSI reference model).

The communication between the NE and the external environment is organized in triggers. Essentially a *trigger* is an assignment of variables to an NE submodule. Variables are also called *attributes* in the NE parlance.

For the above reasons a test case is just a trigger, i.e. a set of assignments to variables of an NE component (*submodule*).

Each trigger is generated by the TelecommunicationManagement NetworkManager (TMNMfor short) which is a software module external to the NE and running on a management server.

The NE receives a trigger t via a software module internal to the NE, the Telecommunication Management Network Agent (TMNA for short). TMNA duty is that of routing the assignments in t to the interested submodule of NE.

We can obtain the triggers structure from GDMO and ASN.1 descriptions for the NE.

In fact, GDMO is used to describe the internal structure of the NE, by specifying the NE submodules and the *attributes* which characterize each submodule.

Since the triggers which stimulate the NE set values for (or get values from) these attributes, we have that the attributes structure, which is described using ASN.1, gives also the triggers structure. Thus, from a GDMO and ASN.1 description, we can obtain a formal description of the triggers structure.

Our approach will take advantage of the GDMO and ASN.1 specifications which have been already written by the engineers. As a matter of fact, we will not directly use these specifications, but the header files which are generated by GDMO and ASN.1 parsers starting from them.

Finally, note that we also could have performed the integration testing in a different way, by using existing languages that support constrained random testbenches within an object-oriented programming methodology (e.g., OpenVera[Ope06], SystemC[Sys06a], and SystemVerilog[Sys06b]). This solution has not been considered since it would have required additional skills to the integration testing engineers, and would have not exploited the technologies which are already used for the integration testing phase.

3 The Manual Integration Testing Phase

In this section we sketch the relevant parts of the integration testing procedure as it was performed before our automatic approach was completed.

First, a *test plan* is informally stated, to define a set of interesting triggers to be sent to the NE.

Second, each of the triggers defined in the previous step is manually submitted to the NE, by using a graphical interface provided by a proprietary simulator tool, the *IST-Simulator* (which essentially simulates the TMNM, Section 2).

In order to send the proper triggers to a generic NE, the IST-Simulator takes advantage of a description of the NE itself, which is given by means of GDMO and ASN.1 standards (Section 2).

This whole integration testing phase is summarized in Figure 1 where also the C header files generated from the GDMO and ASN.1 files are shown.

The above approach has two main drawbacks.

First, the set of interesting triggers to send to the NE is not clearly defined.

Second, to generate all the triggers is a manual, long, tedious and error prone task.

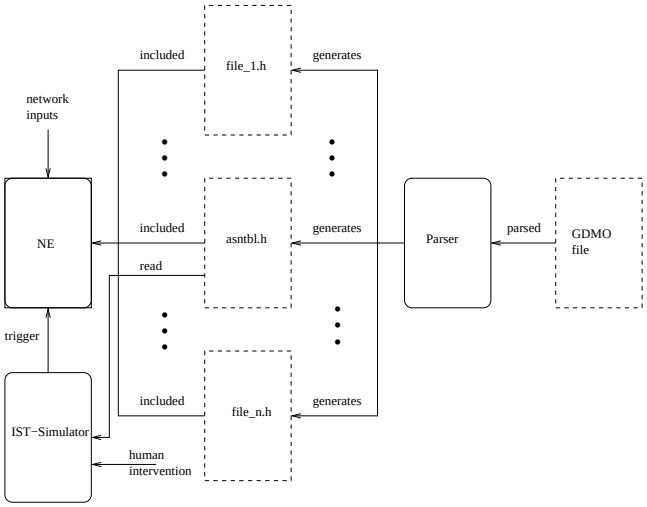

Figure 1: The integration test environment previously adopted

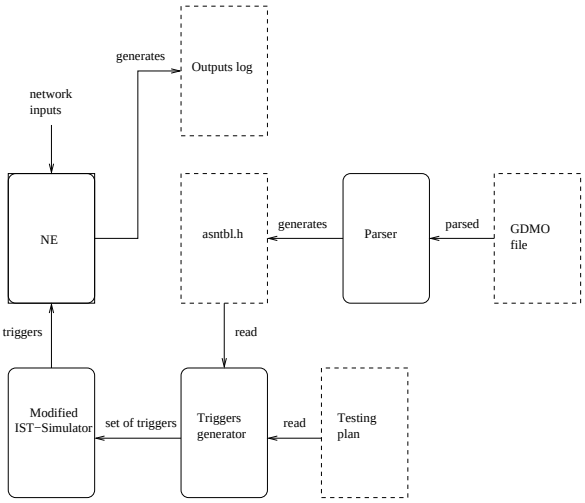

Figure 2: The proposed integration test environment

4 Automatic Generation of Integration Tests

In this section we present our automatic integration testing approach. Our goal is to address the drawbacks of the manual approach listed in Section 3.

To this end we proceed as follows.

First, we define a language to easily specify testing plans, i.e. the set of interesting triggers to be send to the NE. In this way the set of test cases is unambiguously defined. Our testing plan specification language uses a syntax that is very close to the one the company test engineers already know, namely GDMO and ASN.1 formats. This makes it easy for them to use our *test plan specification language*.

Second, from the above definition of testing plan and from GDMO, ASN.1 descriptions of NE we automatically generate the triggers to be sent to the NE. To this end, among other things, we modified the IST–Simulator so that triggers can be sent without going trough the graphical interface.

Figure 2 shows our automatic integration testing framework. In the following we give details about our approach.

4.1 Sequence Choice Trees and Attributes

A *Sequence Choice Tree* (SCT for short) is a labelled tree $T = (V, E, \lambda, \alpha)$, where:

- V is a finite set of nodes;
- $E \subseteq (V \times V)$ is the set of edges;
- λ is a labeling function (λ-label) s.t. each non-leaf node v of T has a label $\lambda(v) \in \{s, c\}$, is undefined ($\uparrow$) on leaf nodes;
- α is a labeling function (α-label) s.t. for each node v, $\alpha(v)$ is an identifier.

A node v s.t. $\lambda(v) = s$ is called a *sequence* node. A node v s.t.$\lambda(v) = c$ is called a *choice* node. Figure 3 shows an SCT for a submodule of a sample NE, which we call `MSI-FP` (see Section 4.1.1).

The α-labels of the leaf nodes of an SCT T (notation $Basic(T)$) are called basic attributes (of T). Thus, $Basic(T) = \{\alpha(v) | v \text{ is a leaf node in } T\}$. For example, from the SCT T in Figure 3 we have that the basic attributes of `MSI-FP` are: $Basic(T) = \{\texttt{sN}, \texttt{inst}\}$.

The α-label of a node which is not a leaf of the SCT T is called a *complex attribute* (of T). An *attribute* is a basic or complex attribute.

To each basic attribute a is associated a finite type denoted with $Type(a)$. For example $Type(\texttt{sN}) = [1..5]$, $Type(\texttt{inst}) = [1..2]$, mean that the types given to the basic attributes `sN`, `inst` are, respectively, the finite integer subranges $\{1, 2, 3, 4, 5\}, \{1, 2\}$.

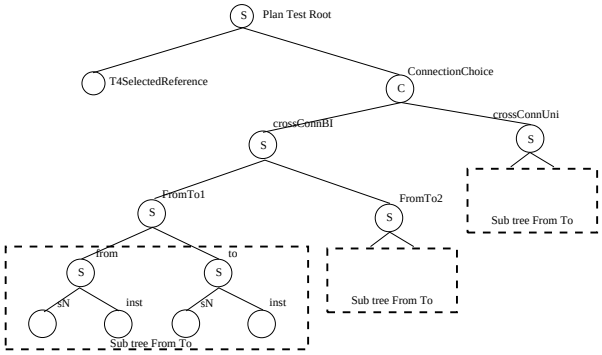

Figure 3: The SCT for the test plan of the MSI-FP

4.1.1 An Example of SCT

We will use a running example to clarify our approach.

As a running example we will use a NE called `MSI-FP` (Multi-Service Integrator – Flat Pack). We assume we want to test two `MSI-FP` submodules, namely the $SETS$ and the CCM submodules.

We focus here on 2 attributes: `T4SelectedReference`, which is a SETS basic attribute, and `ConnectionChoice`, which is a CCM complex attribute. The internal structure of these attributes is shown in Fig. 3, where nodes labeled by s are sequence nodes, nodes labeled by c are choice nodes.

Indeed, `T4SelectedReference` is a basic attribute, and its type consists in just 3 possible values. On the other hand, `ConnectionChoice` is a complex attribute, and its top structure is a choice between two possible subattributes (`crossConnBI` and `crossConnUni`), which are built upon sequences of other subattributes.

The basic attributes for `ConnectionChoice` (i.e. the leaves of the sub-SCT rooted in this attribute) are named `sN` and `inst`, which are both integer subranges.

4.2 Triggers

A *trigger schema* for the complex attribute a is a subtree t of the SCT T s.t. $v \in T$ is the root of t, $\alpha(v) = a$ and all nodes in t that are not leafs are sequence nodes. We call attribute a the *top* attribute of t.

Figure 4 shows a trigger schema for attribute `crossConnUni` of the NE `MSI-FP` described in Section 4.1.1.

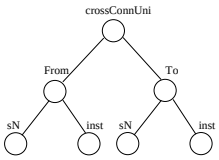

Figure 4: Trigger schema for attribute `crossConnUni`

A `trigger instance` (or just a *trigger*) for the complex attribute a is a trigger schema t for a in which each leaf node v of t is labelled with a value in $Type(v)$. We denote with $t(v)$ such value.

Figures 5, 6 show two trigger instances for `MSI-FP`.

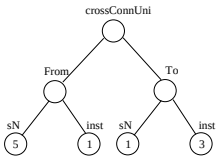

Figure 5: Trigger instance 1 for `MSI-FP`

Remark 4.1 *Note that given a trigger schema t with leafs $v_1, \ldots, v_k$ we have $|Type(\alpha(v_1))| \ldots |Type(\alpha(v_k))|$ possible trigger instances for t.*

4.3 Test Cases

The goal of our approach is to automatically generate, for all NE submodules all *interesting* trigger instances.

We are now ready to present how we define sets of test cases (*testing plan*). The idea is to allow the testing engineers to specify sets of triggers by providing their top attribute names and an optional limitation on the ranges of the basic attributes contained in them. All the triggers respecting the so-specified constraints will be generated and sent to the modified IST–Simulator. The details follow.

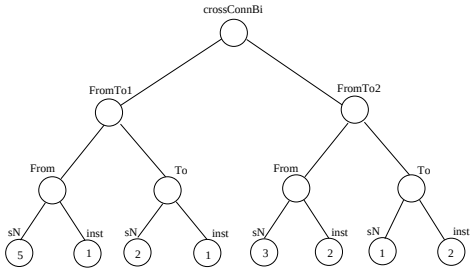

Figure 6: Trigger instance 2 for `MSI-FP`

An *attribute path* in the SCT T is a path $a_1, \ldots, a_k$ of attributes in T s.t. there is a path $v_1, \ldots v_k$ in T s.t. for each $i = 1, \ldots k$ we have $a_i = \alpha(v_i)$.

A *spec-line* is a constraint of form:
`<A>[(<BA> = <ISR>)(, (<BA> = <ISR>))*]`
where:

- `<A>` is an attribute name (the *head* of the specline),
- `<BA>` is a a basic attribute b or an attribute path in the SCT T from the head of the spec-line to a basic attribute b,
- `<ISR>` is an integer subrange or an enumerative type, which has to be a subset of the type of b, i.e. of $Type(b)$.

As usual, strings in square brackets [] are optional, while strings in starred parentheses ()$*$ can be repeated for zero or more times.

Example 4.1 *The following is a spec-line with head a basic attribute for the SCT in Figure 3:*
`T4SelectedReference`
The following is a spec-line with head a complex attribute for the SCT in Figure 3:

```
ConnectionChoice
    (crossConnBI.FromTo1.from.sN = [1..5]),
    (crossConnBI.FromTo2.from.inst = [1..4])
```

Let v be a leaf node of an SCT T and let $v_1, \ldots v_n$ be the path from the root of T to v (i.e. $v_n = v$). We say that v is *compatible* with the attribute path $a_1 a_2 \ldots a_k$ iff for each $i = 1, \ldots k$ we have: $\alpha(v_{n-k+i}) = a_i$.

Intuitively, a trigger instance t *satisfies* a spec-line iff the value assigned by the trigger instance to the basic type satisfies the constraints in the spec-line. More formally, a trigger instance t *satisfies* the spec-line $a(\theta_1 = R_1) \ldots (\theta_k = R_k)$ iff t is a trigger instance of the sub-SCT rooted at a and one of the following conditions holds:

- for each leaf v in t there exists $i = 1, \ldots k$ s.t. v is compatible with θ_i and $t(v) \in R_i$
- there is no $i \in \{1, \ldots k\}$ s.t. v is compatible with θ_i.

Example 4.2 *The trigger instance in Figure 5 satisfies the spec-line*

```
ConnectionChoice
    (crossConnBI.FromTo1.from.sN = [1..5]),
    (sN = [1..3]),
    (crossConnBI.FromTo2.from.inst = [1..4]),
    (inst = [1..2])
```

Note that, in general, we cannot use just the basic attribute b instead of the path to b in a spec-line. This is because there may be more than one leaf node labeled with the basic attribute b.

A sequence $t_1, \ldots t_k$ of trigger instances is *compatible* with a testing plan specification $<A_1(r_1), \ldots, A_k(r_k)>$ iff for each $i = 1, \ldots k, t_i$ is compatible with $A_i(r_i)$.

Given a testing plan specification $\mathcal{L}_{spec} =< A_1(r_1), \ldots, A_k(r_k)>$, where each A_i is an attribute name and r_i is the relative list of optional restrictions, our goal is to generate all sequences of trigger instances compatible with $\mathcal{L}_{spec}$.

Of course one may as well generate *all* trigger instances without posing any constraint. However, from remark 4.1, we known that the number of such instances (test cases) can be prohibitively large.

Note that the constraints in the spec-lines comes from test engineers, since they typically know quite well what are the meaningful subset of values for each basic attribute.

The simple language used to define $\mathcal{L}_{spec}$ allows test engineers to easily focus on the interesting test cases.

In fact its syntax is quite familiar to them being quite close to GDMO, ASN.1 formats.

Finally, some basic attributes have an exceedingly large type imposed by conformance to the standards. Using spec-lines we can test our NE only on the values for these attributes that are meaningful for the case at hand.

4.3.1 An Example of test plan specification

A test plan specification example is shown in Fig. 7. This example states that all the possible trigger combinations for `T4SelectedReference` and for `ConnectionChoice` respecting the given restrictions have to be generated and sent. More in detail, `T4SelectedReference` has no restrictions, thus all the 3 test cases have to be considered, while for `ConnectionChoice` we have that, e.g., the sN basic attribute may take values in [1..3] everywhere it is found, but if it is the one in the leftmost path of the SCT subtree of `ConnectionChoice` shown in Figure 3 then it can take values in [1..5]. Note that the overall number of these test cases is $3 \times 5 \times 2 \times 3 \times 2 \times 3 \times 4 \times 3 \times 2 \times 3 \times 2 \times 3 \times 2 = 466560$. This is indeed an affordable number of test cases. On the other hand, without the restrictions imposed by the specification in Figure 7, the number of test cases would have been exceedingly large.

```
T4SelectedReference
ConnectionChoice
 (crossConnBI.FromTo1.from.sN =
   [1..5]),
 (sN = [1..3]),
 (crossConnBI.FromTo2.from.inst =
   [1..4]),
 (inst = [1..2])
```

Figure 7: A test plan specification example

4.4 Generation of Test Cases

Given a testing plan specification $\mathcal{L}_{spec} =< A_1(r_1), \ldots, A_k(r_k)>$, the triggers generator uses the information in the `asntbl.h` header file (see Figures 1, 2) to build k SCTs $S_1, \ldots, S_k$ representing the structure of the triggers with top attributes $A_1, \ldots, A_k$. Finally, a unique SCT S is built having a sequence root node with k successors, which are set to $S_1, \ldots, S_k$. This latter construction stems from the fact that indeed the k triggers have to be sent in the given order.

We then use a procedure `GenTriggers` (Figure 8) that takes the SCT S and generates the sequences of k triggers that the TMNM has to send to the NE.

Let v be a leaf node of T. Then the number of *choices* in v is $|Type(b)|$ and we need $\lceil \log |Type(b)| \rceil$ bits to represent them. Let v be a choice node of T with k successors.Then the number of choices at node v is k and we need $\lceil \log k \rceil$ bits to represent them. We denote with $numchoicebit(v)$ the number of bits needed to represent the number of choices of (the non-sequence) node v.

The idea is to browse S with the guidance of a counter `cnt`, which is done by the auxiliary function `BrowseByCnt` of Figure 8. Namely, the binary representation of `cnt` is used in order to univocally make a choice when needed, i.e.: 1. when a choice node is reached, thus one of its successor nodes has to be selected; 2. when a leaf node is reached, thus a value in its basic attribute domain has to be picked.

In fact, choice and leaf nodes in S comes labeled with an incremental integer code. This code allows us to single out, in the binary representation of `cnt`, groups of bits whose value is used to make choices. Thus, if g is the value for the group of bits of node n, then procedure `BrowseByCnt` will choose the g-th successor node of n (if n is a choice node) or the g-th value in basic attribute domain of n (if n is a leaf node).

Given this, it is sufficient to loop on all possible values of `cnt` (which is done in function `GenTriggers` in Figure 8) to have all the possible trigger sequences, without repetitions.

```
GenTriggers(SCT S) {
 m = number of non-sequence nodes
     of S;
 CN = set of non-sequence nodes of
      S;
 tot_bits = ∑_{v∈CN} numchoicebit(v);
 foreach cnt in [0..2^tot_bits − 1] {
  if (cnt is a valid value) {
   trggrSet = ∅;
   BrowseByCnt(trggrSet, cnt, S);
   foreach trigger in trggrSet
    send trigger;
} } }

BrowseByCnt(list ret, int cnt,
            SCT S) {
 if (S is a sequence) {
  foreach node in succ(S)
   BrowseByCnt(ret, cnt, node);
 } else if (S is a choice) {
  tmp_cnt = ReadRelValue(cnt, S);
  node = tmp_cnt-th element of
         succ(S);
  BrowseByCnt(ret, cnt, node);
 } else { /* S is a leaf */
  tmp_cnt = ReadRelValue(cnt, S);
  BA = restricted interval related
       to S;
  choice=tmp_cnt-th element of BA;
  append(choice, ret);
} }

ReadRelVal(int cnt, SCT S) {
 return the value of the group of
  bits in cnt related to S;
}
```

Figure 8: Procedure `GenTriggers`

Note that we have to discard some values of `cnt`. In fact, if the number of successor of some choice node (or the number of values of some leaf node) is not a power of 2, then the related group of bits in `cnt` may have a invalid value.

5 Evaluation and Conclusions

We presented a case study on the integration in a real industrial design flow of an *Automatic Integration Test Generation method*.

Our approach uses a GDMO-like language to define *testing plans*. This avoids retraining for testing engineers. Moreover we developed software to easily integrate our approach in a TMN based design flow. Thus our approach can also be used by other companies that use the TMN framework.

Using our tool we were able to detect errors which previously went undetected.

References

[ASN06] ASN.1: http://asn1.elibel.tm.fr/en/, 2006.

[Bla94] Uyless Black. *Network Management Standards: SNMP, CMIP, TMN, MIBs and Object Libraries*. Mcgraw-Hill, 1994.

[Dub00] Olivier Dubuisson. *ASN.1 - Communication between heterogeneous systems*. Morgan Kaufmann Publishers, 2000.

[GHR+03] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (ttcn-3). *Computer Networks*, 42(3):375–403, 2003.

[Heb95] Baha Hebrawi. *GDMO: Object Modelling & Definition for Network Management*. Technology Appraisals, 1995.

[ICM06] CMIP: http://www.sei.cmu.edu/str/descriptions/cmip.html, 2006.

[ITU06] GDMO: http://www.itu.int/ITU−T/formal−language/gdmo/database/itu−t/, 2006.

[Mar03] Peter Marwedel. *Embedded System Design*. Kluwer Academics Publishers Inc., 2003.

[Ope06] http://www.open-vera.com, 2006.

[OSI06] OSI: http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/osi_prot.htm, 2006.

[Sid98] David J. Sidor. TMN standards: Satisfying today's needs while preparing for tomorrow. *IEEE Communications Magazine*, 36(3):54–64, Mar 1998.

[Som01] Ian Sommerville. *Software Engineering*. International Computer Sciences Series. Addison-Wesley, Harlow, UK, 6th edition, 2001.

[Sys06a] www.systemc.org, 2006.

[Sys06b] www.systemverilog.org, 2006.

[TMN06] TMN: http://www.iec.org/online/tutorials/tmn/, 2006.

[TTC06] TTCN-3: http://www.ttcn-3.org/, 2006.

[Udu99] Divakara K. Udupa. *TMN: Telecommunications Management Network*. McGraw-Hill, 1999.

[Wil01] A. Wiles. ETSI testing activities and the use of TTCN-3. *Lecture Notes In Computer Science*, 2078:123–??, 2001.