Automation of Avionic Systems Testing *

David Cebrián, Valentín Valero and Fernando Cuartero

Albacete Computer Science Research Institute
University of Castilla-La Mancha, Avda. España s/n, Albacete (Spain)
http://www.dsi.uclm.es

Abstract. In this paper we present an automatic testing process to validate Avionic Systems. To do that, we have developed a tool that interprets scripts written in Automated Test Language and translate them to user codes written in C language. To carry out this work, the syntax of scripts has been defined by a context free grammar. This testing process is based on the execution of a pre-defined set of test cases. Currently, these test sets are obtained from Test Description Document and they are introduced in the system in C code manually. Therefore, automation of this process would reduce the time used for the testing, as a great quantity of tests are realized and a great quantity of errors are made when tests are made by hand.

Key words: Testing, Avionics systems, Real time systems, grammar testing

1 Introduction

In the development of Avionic Systems, testing and validation, have become a very important part into the software development process, due to the critical real environment where they are going to work. For this reason, testing is usually a very time consuming task. Test automation facilities are desirable in order to reduce the time required for this task.

Actually, for the development of Avionic Systems Software, and in general for the development of complex critical systems, the use of formal techniques for evaluating the capabilities provided by the system and the expected ones becomes very important. Depending on the system to be designed, very different specification formalisms can be used [2, 4, 12, 15].

Real-time and embedded systems are nowadays so complex that to completely specify their behavior is a very difficult task. In particular, these systems are very heterogeneous and include a big amount of components with different natures (sensors, busses, displays, keyboards, storage devices, etc.).

For this reason, software testing, and mainly in this kind of systems, has become a very important part into the software development process [15], as

* Supported by the Spanish government (cofinanced by FEDER founds) with the project TIN2006-15578-C02-02, and the JCCLM regional project PAC06-0008-6995

systems are more and more complex and not detected failures can have fatal consequences.

A failure in a software system can occur for several reasons, now we just mention some of them:

- Specification deficiencies:
  - Incomplete description of functionality.
  - Inconsistent description of functionality.
- Design errors:
  - Misinterpretation of specification.
  - Erroneous control logic.
  - Insufficient error handling.
- Coding errors:
  - Non-initialized data.
  - Usage of wrong variables.

In the development of Avionic systems, testing and validation is a decisive job due to the critical real environment where they are going to work. The effects of an avionic system malfunction would be catastrophic in many cases, so intense efforts to avoid failures are always taken and heavy tests are performed on the equipments. That is why testing is usually a very time consuming human effort and so much time is dedicated to achieve the necessary qualification.

In the literature about avionic software testing, we can see in [14] a description of the main aspects that must be considered to test this kind of systems. Another related work is [7], where model-based simulation testing is used to test avionic software systems.

Then, the main purpose in these works is to be able to qualify before getting the system totally operative in the Avionic systems in order to improve the whole process. For that purpose, test automation facilities are desirable in order to reduce the time required for qualification. Thus, in this paper our goal is to describe a tool that we have developed in order to automate a part of the testing process for Helicopter embedded software.

This tool interprets scripts written in Automated Test Language and translates them to user codes written in C. This Automated Test Language is closer to human language and it permits to describe test cases easily. This test automation tool has been applied to helicopter software testing, in a real corporation (Eurocopter company).

The particular language that we use has been defined by the software testing group of the helicopter company, and it is specific for this purpose. There are, of course, some standard notations that could be used to accomplish this task too, like TTCN-3 [3].

To interpret these scripts written in Automated Test Language, a context-free grammar has been used. Nowadays, grammars are omnipresent in software development. They are used for the definition of syntax, exchange formats and others. Several important kinds of software rely on grammars, e.g., compilers,

debuggers, profilers, slicing tools, pretty printers, (re-) documentation tools, language reference manuals, browsers, software analysis tools, code preprocessing tools, and software modification tools [6, 9, 16].

A grammar defines a formal language and provides a device for generating sentences. From a perspective of software engineering, a grammar may be considered as both a specification (defining a language) and a program (serving as a parser generators input). In practice, ensuring that a grammar specifies an intended language which can be considered as user requirements is indeed a validation problem [5].

Testing is a standard way to validate specifications or programs (formal analysis is another). Grammar testing covers various technical and pragmatic aspects such as coverage notions [10], test set generation [11], correctness and completeness claims for grammars or parsers, and integration of testing and grammar transformations.

In this paper we focus our attention on test set generation. Test data generation requires a variety of techniques [8], for example, to minimize test cases, to accomplish negative test cases, etc.

The rest of the paper is structured as follows. In the next Section a description of a system that accepts user codes generated by the tool will be described. Next, the tool operation is shown. In Section 4 a case study is presented. Finally, our conclusions and future work are presented.

2 System overview

In this section, we describe the specific system that we consider for our Avionic System Testing environment. This System accepts user codes generated by the implemented tool. The System is a combination of a real-time platform designed to execute Avionic Equipment Tests and a Unix workstation which is used as a user interface to drive the test.

The Avionic Equipment of the considered helicopter basically consists of a core computer that integrates, among others, the functions concerning control and display subsystem, navigation subsystem and communication subsystem. These subsystems are connected via redundant busses to improve reliability.

Then, the Real-Time System platform contains the simulated equipment of the helicopter and it is mainly composed by I/O cards for the different busses of the helicopter (MILBUS 1553, ARINC 429, RS485, etc.) and an avionic database containing specific information about the helicopter to be tested (Fig. 1).

And finally, the System Unix Workstation shall provide the capability to control the operation of the tests and run the simulations. By means of it, we can manage the simulations, specifying the concrete datas that are to be used, we can also inject some types of errors to test the system reactions, we can prepare the scripts for the tests, and of course, we can monitor the system, to view and record the results of the simulation.

Another feature of interest of this system is that of scenarios, which allow the testers to establish the context in which tests are to be made. Then, a

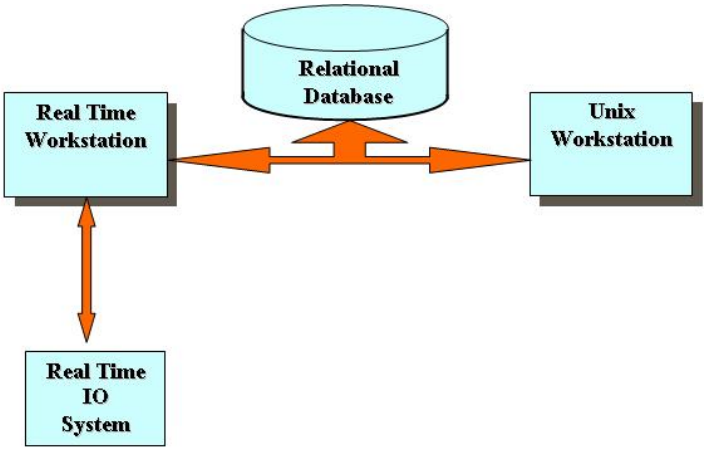

Fig. 1. Real-Time platform

scenario is a specific test directly linked to an upper context that defines the set of objects which should be operational during the test like codes, data items to be displayed in dashboards, data items to be modified in dashboards and configuration of simulated equipment. The descriptions of the system and of the scenarios are stored in the database.

2.1 Testing environment constraints

Due to the nature of the system environment there are important constraints related to how user code should be generated. User code will be always executed in the same way: there is a period indicated in the test information which serves as a basis to cyclic execution of pieces of code. Each cycle will have the duration of the specified period, and each piece of code is forced to be executed within one cycle. Generated user code has to fit this rule and special care has to be taken in controlling that no piece of code extends the cycle duration, since this would cause a general system failure.

Thus user code must be divided according to this restriction, so a control code is introduced to select the concrete piece of code that must be executed on each cycle. The easiest solution to fulfil this as a set of switches, each one including a persistent counter that will indicate in each cycle what case clause to execute:

```
static int Counter = 1;

int TestRun ( )
{
  switch ( Counter )
```

```
  {
    case 1 :
      <execution piece 0>
      Counter = 2;
      break ;

    case 2 :
      <execution piece 1>
      Counter = 3;
    ...
  }
}
```

Thus, the modification of the value of the persistent counter will allow navigating between different execution pieces.

This hard coding constraint makes the most challenging task of the code generator to establish a set of mechanisms that will allow translating a sequence of instructions to an equivalent code made of a set of switch clauses connected and controlled by auxiliary variables.

Then, once we have described the testing environment, let us see the format that system user codes have (each test). They are defined by the following items:

- A name - (32 bytes length).
- A period - integer expressed in milliseconds.
- An Interface: a text file description describing the exchange of information between the user code and the system.
- A specific main module which will be automatically called by the system according to the period.
- Some user modules.

Furthermore, a test is composed of three hierarchical levels: procedures, tasks and steps. This division obeys the grammar definition that we are considering, in which each test is divided in this way. Then, each of these three levels may be run in a kind of control loop for some specific set of values, which may be defined directly in the test description or in a text file which can be modified from an execution to another.

3 The tool

The system accepts C code to specify the tests. With the purpose of making easier the specification of these tests, a tool called Code Generator, has been built. This tool accepts as input user scripts that make easier the tests specification. The function of this software is to interpret these scripts written in Automated Test Language and translate them to system user codes written in C.

These user scripts (written in Automated Test Language) are generated from a document called, test description document. This document is written by expert testers and it is written in natural language. So to carry out these tests, they

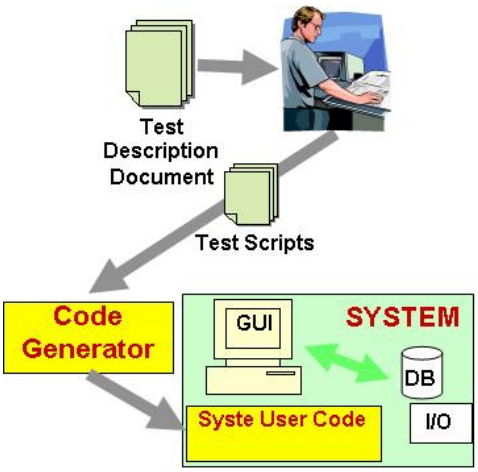

Fig. 2. Testing process

have to translate the test description document to Automated Test Language (Fig. 2). This task is made manually but in the future the test description document will be written in a high level language and this task will be automated. An example of Automated Test Language is shown in Fig. 3.

The tool must read the scripts specified by the user and translate them into C code. C functions will be created and grouped into different files in order to increase modularity. The file structure will follow the scheme shown in Fig. 4:

Some execution levels are considered for system user code execution control. Each level will call the level immediately below. In the example script 1000 (Fig. 3), the resulting user code will be executed in different nesting levels:

- **Level 0**: code in INIT, RUN (not nested switch) and END phase in Main User Code file.
- **Level 1**: Example_Script 1000 related code: nested switch in RUN phase in Main User Code file.
- **Level 2**: P1 and P2 related code. This is the code in *ProcedureName*_Run.cc files.
- **Level 3**: T1 and T2 related code.This code is located in *ProcedureName_TaskName*.cc files.
- **Level 4**: Step 1 and Step 1 related code (but not the nested code they contain: IF/ THEN/ ELSE/ ENDIF). This code is also located in *ProcedureName_TaskName*.cc files.
- **Level 5 and below**: code inside IF and ELSE clauses. This code is located in *ProcedureName_TaskName*.cc files.

```
Example_Script 1000
Procedure P1
    Task T1
        Step 1
            GET Variable INSERT_BOTH
            IF WAIT_VALUE VALUE_AVAILABLE TRUE 0
            THEN
                ADDOUTPUT "IRS1 available"
            ELSE
                ADDOUTPUT "IRS1 not available"
            ENDIF
Procedure P2
    Task T2
        Step 1
            GET Variable INSERT_BOTH
```

Fig. 3. Example script 1000

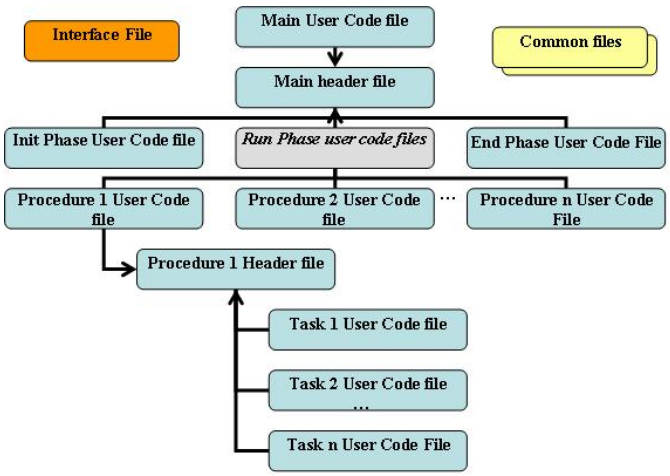

Fig. 4. Generated file structure

3.1 General strategies

This tool has been implemented in a platform independent way. This approach allows the system to be compiled for several platforms with no changes in the source code.

The formal syntax of the scripts has been defined by a non-ambiguous context-free grammar, so each valid input will have only one possible derivation tree. The complete grammar is omitted, because it is very large, it has 72 production rules, and it is unimportant for our purposes.

The tool must read the scripts and translate them into C code. For this reason, a LALR(1) interpreter has been built, which can deal with many context-free grammars by using small parsing tables. The parsing process of these interpreters is divided into several levels:

- **Lexical level:** This is the simplest and lowest level of the interpreter. In this level the parser reads the input character by character and translates these sets of characters into words.
- **Syntactic Level:** Once the input has been divided into tokens, the processing is much easier. This level checks the correctness of a given input according to the specified grammar. Then, once the words of the input have been identified (which is done by the lexical level), this level just tests if there is a derivation tree in the grammar, which leads to the given input. As this level generates the derivation trees of the given grammar, it is very important that this grammar has no ambiguity. Each given input will have, if it has any, only one possible derivation tree.
- **Semantic Level:** A grammar must define all the correct sequences of the language. But there are conditions which might be really difficult to represent in the definition of a language. To avoid this, grammars which define a superset of the correct sequences accepted are used and some tests to check that the accepted sequences fulfil these constraints are added. In this step we check that some features (like the declaration of variables) are consistent. These features have to be checked over the whole test. For this reason, a structure capturing a logical representation of the scripts has been introduced. Specifically, we have used ASTs (Abstract Syntax Trees). Figure 5 shows an example of AST, in this case that one associated with the Example Script 1000 (Fig. 3). In this example, we can see how the defined tree has a structure according to the level division in the scripts. These nodes which keep the structure of the test (TestDescription, Procedure, Task, Step) could be called structural trees. For each of the available instructions in the test (If-then-else, GET, ADDOUTPUT) there is an appropriate tree too.

For the development of the project, lex and yacc tools have been used. Particularly, flex 2.5 implementation of lex and bison 2.1 implementation of yacc have been chosen.

- **Flex**[13]: with this tool programs whose control flow is directed by instances of regular expressions in the input stream can be written. It is well suited for

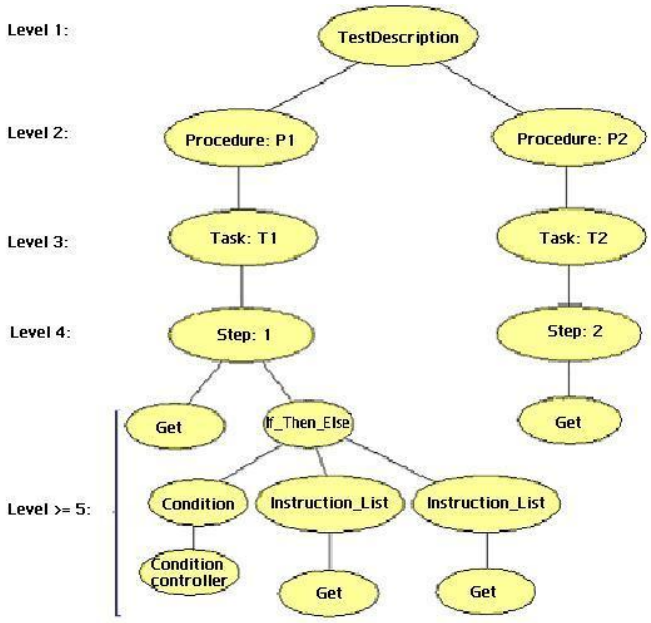

Fig. 5. AST associated with Example Script 1000

editor-script type transformations and for segmenting input in preparation for a parsing routine.

– **Bison**[1]: Yacc provides a general tool for imposing structure on the input to a computer program. The yacc user must prepare a specification for the syntax that the input must fulfil: this specification includes rules describing the input structure (productions from our context free grammar), but also code to be invoked when these rules are recognized (semantic actions).

4 Case study

We now use another example (Antenna_Selection) to describe the tool operation (Fig. 6). In practice, the scripts are obtained manually by using the test description document. Notice that we do not need to know how these scripts are obtained (which technique is used in particular to generate them), neither to fully understand their mission because these tasks are carried out by testing engineers of helicopter company.

Code Generator takes this script as input and translates it to system user code. The system user code thus obtained consists of some files, the structure of which is shown in Fig. 7.

The contents of these files are:

– **Antenna_Selection.x:** This is the interface file. It contains imports and exports of variables from the database.

```
Antenna_Selection 200
PROCEDURE P1_Antena
  TASK T1
    STEP 0
      SET External 5
      GET External1 Internal1
      SET External1c 6.2
      ADDOUTPUT "HELLO"
    STEP 1
      GET External2 Internal2
      FREEZE Proof1
      UNFREEZE Proof2
      FREEZE Proof3
    STEP 2
      GET External2 Internal2
      ADDOUTPUT hey
      INSERT_LABEL hello label1
      REMOVE_LABEL hello label1
      IF (6==6) THEN
        INSERT_LABEL hello label1
      ELSE
        ADDOUTPUT hey
        INFORMATION "proof"
        ENDIF
    TASK T2
      STEP 0
        GET External Internal
    TASK T3
      STEP 0
        GET External1 Internal1
        SLEEP 10
        GET External2 Internal2
PROCEDURE P2_Antena
  TASK T1
    STEP 0
      SLEEP 10
      SLEEP 12
      GET External3b Internal3b
      SLEEP 15
      GET External3c Internal3c
      GET External3d Internal3d
PROCEDURE P3_Antena
  TASK T1
    STEP 0
      ADDOUTPUT "hello"
```

Fig. 6. Example script

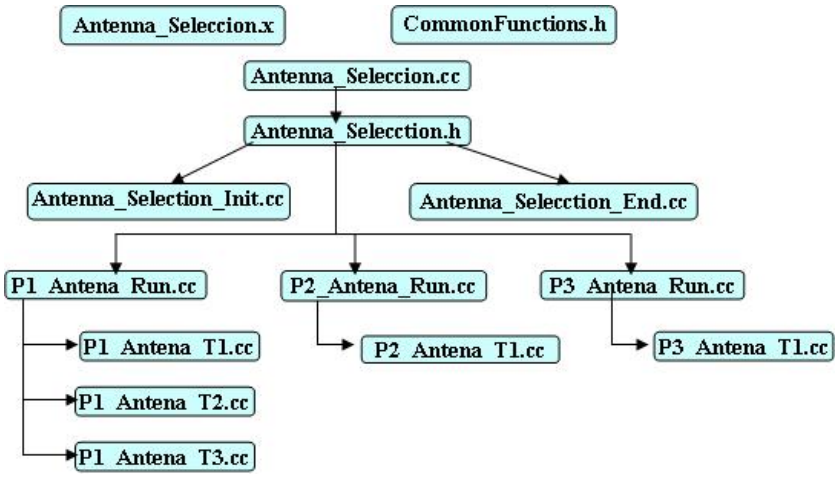

Fig. 7. Generated files

– **CommonFunctions.h:** This file contains a set of common functions used by other files.
– **Antenna_Selection.cc:** This is the main file and controls the test operation. Below, its source code is shown in Fig. 8:

```
#include Antenna_Selection.h"

void CODE (unsigned long inCodeState)
{
  switch(inCodeState)
   {
    case CO_INIT:
      Level = 1;
      for(int i=0; i<4; i++)
        Step[i]=0;
      Antenna_Selection_Init();
      AnaisEndOfPhase ();
      break;
    case CO_RUN:
      switch ( Step[1] )
      {
        case 0:
          P1_Antena_Run();
          if ( Level==1 )
            Step[Level]
                 =Step[Level]+1;
        case 1:
          P2_Antena_Run();
          if ( Level==1 )
            Step[Level]
                 =Step[Level]+1;
        case 2:
          P3_Antena_Run();
          if ( Level==1 )
            Step[Level]
                 =Step[Level]+1;
          break;
      }
      break;
      case CO_END:
        Antenna_Selection_End();
        fclose(Fout);
        AnaisEndOfPhase ();
        break;
                                  }
                              }
```

Fig. 8. Antenna_Selection.cc

– **Antenna_Selection_Init.cc:** It contains the code for the inicialization of the test.
– **Antenna_Selection_End.cc:** It contains the code for the conclusion of the test.
– **P1_Antena_Run.cc, P2_Antena_Run.cc and P3_Antena_Run.cc:** They contain the related code with procedures P1, P2 and P3.
– **P1_Antena_T1.cc, P2_Antena_T1.cc, P3_Antena_T1.cc, P1_Antena_T2.cc, P1_Antena_T3.cc:** These files contain the related code with the diferent tasks.

These files are accepted by the test system as input and they are used to drive the test. We can observe, in Fig. 8, that the generated code obeys the constraints imposed by the testing environment as it is divided into some pieces of code by means of a set of switchs. Each piece of code will require a time that will not be greater than the cycle duration (this is controlled by the *inCodeState* variable).

5 Conclusions and Future Work

In this paper, a first step for helicopter software automated testing has been shown. We cannot provide real experimental results because the tests are carried out by the helicopter company testing group, and we have no information comparing the time required for the testing process when it was made manually and currently, by using our tool.

As future work our intention is to extend the automation of testing to other aspects of this process. The whole concept of Automated Test will allow generating a complete test set based on the test description document. It will concern a complete environment including:

1. Automatic database frame import.
2. Automatic scenario definition.
3. Automatic code generation.

The principle is to define a high-level language with a friendly syntax which will be used by test teams to describe their tests. A simple way may be to directly use this high-level language as part of the test description document and to be able to extract it automatically. The main advantage that we can obtain with this automated testing is to save a lot of time in testing execution and humman effort to achieve the necessary qualification.

For this purpose, some constraints must be fulfilled:

1. A friendly syntax and format easily generated from the specifications. This description must be accessible to a test team guy even if he is not an expert in programming languages.
2. Common format agreed by all test teams. The idea is to use a simple support which may be provided by test guys using standard editors like: text editors, Microsoft Office editors, etc.
3. Comprehensive language, not directly mapped on a specific software compiler. The idea is that test team guys must be able to describe a test procedure, even if they are not experts in software programming language.
4. The test description language must cover at least all the functionalities actually covered by the specifics ones already existing.

Our intention for the immediate future is to increase the automation level of testing environment, by including scenarios in the Code Generation tool, and even it would be important to replace the Test Description Document by another document which can be automatically interpreted by a tool.

Acknowledgement: We would like to thank to the anonymous referees for their suggestions and corrections that have contributed to improve this paper significantly.

References

1. R. Stallman C. Donnelly. *Bison. The YACC-compatible Parser Generator*, 1995.
2. Ana R. Cavalli, Jean Philippe Favreau, and Marc Phalippou. Formal methods for conformance testing: Results and perspectives. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 3–17, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.
3. C.Willcock. *Introduction to TTCN-3*, 2002.
4. Marc Constantijn Willem Geilen. *Formal Techniques for Verification of Complex Real-Time Systems*. M.C.W. Geilen, 2002.
5. Chao Liu Hu Li, Maozhong Jin and Zhongyi Gao. Test criteria for context-free grammars. *Computer Software and Applications Conference. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 300–305, 2004.
6. Ralf Lämmel Jan Kort and Chris Verhoef. The grammar deployment kit. *Electronic Notes in Theoretical Computer Science*, 65(3):7, 2002.
7. Gunnar Jonas Jan Peleska, Klemens Brumm and Tobias Hartmann. Advancement in automated simulation and testing technology for safety-critical avionic systems. *Aerospace Testing 2006*, 2006.
8. Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.
9. Ralf Lämmel. Grammar testing. *Fundamental Approaches to Software Engineering : 4th International Conference, FASE 2001 : Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, pages 201–216.
10. Ralf Lämmel and Wolfran Schulte. Controllable combinatorial coverage in grammar-based testing. *TestCom 2006*, pages 19–38, 2006.
11. Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.
12. M. Núñez, F.L. Pelayo, and I. Rodríguez. A formal methodology to test complex embedded systems: Application to interactive driving system. In *IFIP TC10 Working Conf.: International Embedded Systems Symposium, IESS'05*, pages 125–136. Springer, 2005.
13. V. Paxson. *Flex, version 2.5. A fast scanner generator.*, 1995.
14. Jan Peleska. Test automation for avionic systems and space technology (extended abstract). 1996.
15. Jan Peleska. Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family. *Integrated Design and Process technology, IDPT-2002*, 2002.
16. Hui Wu. Grammar-driven generation of domain-specific language tools. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 772–773, New York, NY, USA, 2006. ACM Press.