Automatic Code Generation for Testing

David Cebrián, Llanos Tobarra, Valentín Valero, Fernando Cuartero
University of Castilla-La Mancha, Dpto. Sistemas Informaticos
Avda. España s/n, Albacete (Spain)
fernando@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.

Keywords: 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, it becomes very important the use of formal techniques in order to be able to evaluate the capabilities provided by the system and the expected ones. Depending on the system to be designed, very different specification formalisms can be used [1, 3, 10, 13].

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 [13], as 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:

- 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 [12]. 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.

The main purpose is to be able to qualify before getting the system totally operative in the Avionic systems in order to improve the whole process. Test automation facilities are desirable in order to reduce the time required for qualification. Avionic systems development and integration is also need to be tested and validated in a mix of real and simulated environment, so this coexistence between simulation and reality should be possible.

Then, to facilitate test automation we have developed a tool that interprets scripts written in Automated Test Language and translate them to user codes written in C. This Automated Test Language is a formal language closer to human language and it permits to the engineer to describe test cases easily. It is a compromise between the conflicting goals of a totally exhaustive test and a carefully controlled scenario for validating a subsystem. The use of a model-based testing strategy allows the test engineer to easily create new scenarios by writing scripts, which will be automatically translated to a a very wide range of tests. This text automation tool has been successfully applied to helicopter software testing.

To interpret these scripts, 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 [5, 7, 14].
Any grammar defines a 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 [4].

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 [8], test set generation [9], 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 [6], 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 specific data 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 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 System 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.

So 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:

```c
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 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. Below, an example of Automated Test Language is shown:

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

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 next scheme(Fig. 3):

Some execution levels are considered for system user code execution control. Each level will call the level immediately below. In the example script (see above), 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.

---

1\(^{\text{The complete grammar is omitted, because it is very large and unimportant for our purposes.}}\)
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 can be defined by a context-free grammar. An 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.

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**[11]: 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 editor-script type transformations and for segmenting input in preparation for a parsing routine.
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
ENDIF

Figure 4: Example script

- **Bison**[2]: Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc’s 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. 4). In practice, this script would have been obtained manually by using the test description document:

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. 5:

The contents of these files are:

- **Antenna_Selection.x**: This is the interface file. It contains imports and exports of variables from the database.
- **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. 6:
- **Antenna_Selection_Init.cc**: It contains the code for the initialization 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 different tasks.

These files are accepted by the test system as input and they are used to drive the test.
#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;
            break;
        case 1:
            P2_Antena_Run();
            if ( Level==1 )
                Step[Level] =
                Step[Level] + 1;
            break;
        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;
    }
}
5 Conclusions and Future Work

In this paper, the first step for automated testing has been shown. 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.

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.

As future work, our intention 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.

References


