Automated Functional Test Case Synthesis from THALES industrial Requirements

David Lugato
CEA/LIST/DTSI
CEA Saclay – Bâtiment 451, 91191 Gif-sur-Yvette Cedex France
+33 169 08 91 33
David.Lugato@cea.fr

Frédéric Maraux
THALES Airborne Systems
2, avenue Gay-Lussac 78851 Elancourt Cedex France
+33 134 59 66 25
Frederic.Maraux@fr.thalesgroup.com

Yves Le Traon
Clémentine Nebut
INRIA/IRISA
Campus de Beaulieu, 35042 Rennes Cedex France
+33 299 842 568
yletraon@irisa.fr
cnebut@irisa.fr

Véronique Normand
THALES Research & Technology
Domaine de Corbeville 91401 Orsay Cedex France
+33 1 69 33 07 06
Veronique.Normand@thalesgroup.com

Hubert Dubois,
Jean-Yves Pierron,
Jean-Pierre Gallois
CEA/LIST/DTSI
(Hubert.Dubois,
Jean-Yves.Pierron,
Jean-Pierre.Gallois)@cea.fr

Abstract

Test case generation and specification validation are essential concerns for the software industry in its continuous search for productivity improvement and quality mastering. This paper presents the approach developed in the MUTATION project for functional test case synthesis in the THALES industrial context.

A two-step approach is proposed, which automates most of the process in a continuous way.

The first step consists in expressing the requirements in the Requirements Description Language (RDL) textual formalism and completing, disambiguating and validating them through simulation. Test objectives are then automatically derived from the RDL formalisation.

The second step synthesises functional test cases through combining the test objectives produced at step 1 and symbolic path computation on a UML detailed design model of the application, using the academic tool AGATHA (French acronym for toolset dedicated to automatic test generation for automata based specification). Test cases are produced through selecting symbolic behaviour paths that match the test objectives, applying a constraint solver to generate numeric tests.

This approach is based on three innovations that are the core of this work: 1) definition of an appropriate language and a related methodology to define the test requirements; 2) symbolic execution on a UML action language; 3) combination of test objectives and symbolic execution to synthesize functional test cases.

1. Introduction

The economic impact of test generation is very important for software industry. Formal methods allow system analysis and test generation from specifications. This provides an early feedback on a system's behavior. The benefit of this specification analysis step is considerable, as it simultaneously reduces cost and time of validation, while increasing system reliability. However, formal techniques are generally complex to use in practice: this is why such techniques have not, at this time, penetrated the industrial domain. Therefore, it is crucial to provide tools in which these techniques are automated.

Several techniques exist to validate systems specifications: proof [1], model-checking [5], [3], timed automata [24] and formal testing [8]. Both formal validation of specification and software testing are not yet very mature. Indeed, they require high skills, time and staff, leading many organizations to consider its financial impact too heavy compared to the entire project cost.

Test case synthesis and specification validation are core concerns for THALES in its continuous search for development cost reduction and system quality mastery. These are the subject of the MUTATION project presented in this paper. MUTATION is part of the CARROLL research program1, launched by THALES and two French public research laboratories: CEA (Commissariat à l’Energie Atomique) and INRIA (Institut National de Recherche en Informatique et en Automatique), addressing core software development technologies for large-scale and embedded systems.

This paper presents the approach to test case synthesis currently developed and used experimentally.

1 See www.carroll-research.org for more details.
in the MUTATION project. A two-step methodology (see Figure 1) is proposed to generate functional test cases from textual requirements and a UML design model, in a continuous and mostly automated process. This approach is being applied on two real weapon systems of last generation combat aircrafts, in THALES Airborne Systems (TAS).

The two-step methodology exploits UML models satisfying well-defined modelling rules.

Roadmap: this paper first introduces the THALES Airborne Systems software development context and the current validation process and its limitations. Section 3 presents the first step of the approach that aims at deriving test objectives from textual requirements. Section 4 provides an overview of the different formal methods used in the AGATHA toolset. Section 5 then focuses on the second step of the approach and presents the solution developed by using the AGATHA toolset for a symbolic execution of THALES specifications, in order to synthesize test cases from test objectives. Conclusions and perspectives then follow.

2. THALES context

2.1 Combat aircraft weapon navigation systems

To allow aircraft crews to carry out their missions in an ever more complicated and stressing operational context, the combat aircraft systems assume more and more functions, using the processors and embedded software capabilities to a maximum.

The paper deals with last generation combat aircraft (Mirage 2000-9 and Rafale) Weapon Navigation Systems (WNS). The main processor of these fighters (MDPU: Modular Data Processing Unit) and the application software are developed by TAS. This software addresses mission management (e.g. flight management, navigation, human computer interaction) and weapon systems management (radar, self protection systems, ...).

TAS has been developing this software family since the seventies, and uses an object process based on the UML notation. The process is a result of studies that were conducted at the beginning of the nineties [2].

To ensure the quality level required by the customer, more than 50% of this process is dedicated to the test phase. Therefore, test productivity enhancement is a major improvement axis for TAS.

2.2 Process

The whole development process (analysis, design, and coding) follows an object oriented method based on UML. Its objective is to master the product and to simplify its testing strategy. It imposes restrictions on design (e.g. limiting coupling, control on data exchange, no synchronous call), and coding (no dynamic allocation, no recursion).

Mainly two testing techniques are used: unit/integration testing (at the method or set of methods level) and functional/validation testing (the application software being executed in the real processor with environment simulation software
allowing replacing the software operational interface). The requirements are the reference during the testing phases. Each test is built in order to validate some specific requirement with the test stopping criteria:

- 100% of requirements are checked,
- 100% of code statements are executed.

2.3 Possible and expected improvements
It seems to us that the organization of the testing steps is optimized in the current process. It is now necessary to take a leap to enhance significantly testing productivity. This change requires the adoption of new technologies such as test case synthesis, formal validation, proof, simulation, ...

The MUTATION project aims at improving two aspects of the testing strategy: requirements validation and synthesis of test cases.

3. Step 1 - from textual requirement to test objectives
The input of the problem we seek to solve is a set of textual requirements written in the RDL Requirement Description Language. This controlled natural language [7] reduces the possibility of introducing ambiguous, incomplete or incorrect sentences. The goal is thus to build requirement rules, that will prevent writing ambiguous sentences. The initial problem is to unify a large set of separated requirements (hundreds) into a small number of use cases corresponding to the system services, to reject ambiguous or incomplete textual requirements, and to identify both actors of the system and requirements that are not of the right level of abstraction.

3.1 Translation into an AST
The difficulty comes in the translation of a semi-formal description to a non ambiguous and compact formal representation. The main constraint on the problem is to maintain a requirement language close to the natural language (e.g. some of them are often from the client’s company). We admit that a limited number of domain experts are able to define “interpretation patterns” for the requirement language.

The technique to treat textual requirements produces an Abstract Syntax Tree (AST), independently from its interpretation(s). It is quite easy to write significant sentences in natural language, but it is difficult for 50 requirement writers to formally describe the expected behavior of a complex system. The requirement writers have to be guided through an incremental improvement of the textual requirements so that they reach a preciseness level equal to the one of a formal approach. This result is obtained as follows:

- If a requirement cannot be parsed, it means that it is not syntactically correct w.r.t. the RDL language. This requirement must be rewritten.
- If a requirement is parsed into the AST but cannot be interpreted by any of the “interpretation patterns” defined by the experts of the domain, it means either that the requirement has no meaning and must be rewritten or that an interpretation pattern must be introduced. It must be added to the set of interpretation patterns. It is also possible that the requirement is not from the right level of abstraction, e.g. describing “how” the system works rather than “what” it does.

At the end of the requirement improvement process, the AST can be derived into another format (e.g. “pretty-printing” of textual requirements). In the “use cases and contracts” format, requirements are automatically unified in terms of use cases (corresponding to the system services) enhanced with contracts, i.e. pre/post conditions of activation (see [19] and [20] for details).

3.2 Interpretation patterns
An interpretation pattern is a template structure in the AST of the parsed RDL requirements. A pattern has a crucial impact on the process since it both removes ambiguity and associates semantics to a sentence. This semantic part may be specific to the company. It is linked to a production, e.g. a use case with contracts. The set of couples [pattern,production] includes means to express:

- pre and post conditions of an action on the system,
- changes of objects property when occurs,
- activation of an action after another,
- reaction at property change.

For example, a pattern can describe the fact that a property of an object changes upon the activation of a use case. The top line of Table 1 describes the general form of the pattern and the bottom part describes some additional constraints on the involved items (i.e. constraints on lexical items coming from the parsing of the RDL statements).

| IF S1=Action THEN O1=ObservableProperty |
| O1.type | BECOMES |
| O1.reference | THE |
| S1.type | DOES |

Table 1: a Pattern structure

Once a pattern is defined, a production has to be linked to the pattern. In the example, the production is a use case indicating that a property must change its value, due to the execution of an action (Table 2).
Currently, requirements that cannot be taken into account concern data aspects (validity domains of numeric variables) or non-functional properties (performances, real-time constraints).

The number of interpretation patterns needed to interpret the 70% of requirements is 16. In fact, 8 patterns were defined in the common interpretation core, common to any company, and 8 were added to fulfill the TAS requirements. So, with a small number of interpretation patterns a significant part of the system requirements have been translated. It constitutes the core of the requirement asset that will be used as a basis for further projects.

If we only consider the treatment of textual requirement, this process has the following advantages:

- Requirement rules are built, that allow the requirement writers to specify unambiguously the system. Indeed, the interpretation patterns may constitute mandatory rules for writing requirements correctly, and are thus a requirement asset.
- The method is adapted to any application context and also new interpretation patterns to be added when an existing system evolves (migration, reuse …).
- The reuse of an existing requirements set is easier.
- The automated identification of the various levels of abstraction of requirements.
- The unification of distributed requirements into a set of services expressed with use cases and contracts.
- Identification of requirements ambiguities and incompleteness, if interpretation patterns are not mandatory as writing rules.
- Requirement debugging and validation through use cases simulation.

### 3.4 Test objectives generation

When requirements are validated using simulation, other outputs are offered such as the generation of test objectives.

A test objective is a valid sequence of use cases detailing the instances of actors and needed parameters. It is presented under the form of a sequence diagram such as the one of Figure 4.

A test synthesis tool must be then applied to transform these test objectives into executable test cases. Results of [20] show that most of the functional code and half of the robustness code can be covered by such test cases derived from test objectives. There are two benefits: guaranteeing that the code fulfills the requirements, and defining and reusing high-level tests.

In MUTATION, a translation is needed to allow the usage of the generated test objectives by the AGATHA toolset, as described hereafter in section 5. The RDL-
level test objectives are translated into the terms of the UML design model through exploiting a traceability model provided by the design team. RDL services are mapped to UML operation calls; RDL property conditions are mapped to design model state conditions. The resulting test objectives are defined as sequences of operation calls with guards, and represented as UML sequence diagrams.

4. The AGATHA toolset

As a preliminary to describing step 2 of the methodology, this section provides an overview of the theoretical foundations of the AGATHA toolset; for more details see [9], [16], [17] and [22]. The aim of AGATHA is to help engineers to debug behavioral oriented specifications. Our approach is based on symbolic execution techniques. The main idea is to translate specifications into Extended Input Output Labeled Transition System (EIOLTS) and to unfold this translation in order to exhibit in an explicit way all behaviors that can be deduced. Some reduction procedures are applied in order to cope with the state explosion problem.

Figure 3 shows the overall architecture of the AGATHA toolset.

4.1 Native language: AEIOLTS

Natively, AGATHA handles specifications formulated in an automation language called AEIOLTS (for AGATHA Extended Input Output Labeled Transition System) that is based on a simplified version of Estelle [12]. Usually Formal Description Techniques (FDT): SDL [13], Statecharts [11] and of course UML specifications can be translated into this simplified version of the basic formalism.

4.2 Symbolic execution

Symbolic execution has been studied in details for programming, see [6], [14] for details. Symbols are used instead of numerical data as input values. Since an input is by nature a constant it follows that these symbolic inputs are new symbols of constants. Symbolic execution allows one to describe all possible computations in an abstract way.

Thus, a peculiar behavior no longer depends on the result of a calculus being completely performed but on an expression representing constraints on the variables being denoted by the symbols of entries. We call Path Condition the conjunction of the encountered guards (constraints) to reach a symbolic node. Each path of the symbolic execution graph denotes several (or even an infinity of) numerical paths characterized by some constraints deduced from the specification.

4.3 Reduction procedures

We introduce on-the-fly procedures to avoid useless computations without loss of information. In order to compute inclusion procedures, we actually work with the Omega Library\(^2\) for specifications based on Presburger Arithmetic [23]. Extensions to more general theories admitting quantifier elimination property are under consideration.

4.4 Simplification procedures

During the computation of AGATHA, symbolic expression of variables and path conditions may rapidly grow. To solve this problem a simplification procedure must be applied “on the fly” in order to shorten expressions and detect useless paths.

As of today we use a simplifier based on rewriting techniques. The rewriting engine is Brute that is part of the CafeOBJ toolset\(^3\). This allows both to maintain symbolic expressions within a reasonable size range and to obtain normal forms for the expressions, making easier the comparison between expressions needed in algorithms such as reduction procedures.

4.5 Composition

The symbolic execution process is performed on one module, but the global application is generally composed of many, so they have to be merged.

There are two possible ways to merge modules. The first solution is to use the composition introduced by Milner [18]. The global module is made of the transitions of its components except those that are synchronized by a rendezvous and are replaced by an

---


\(^3\) Brute rewriting engine, GAIST, http://www.theta.theta.ro/cafeobj
equivalent transition obtained by eliminating the exchanged parameter.

The other solution is to compute the symbolic execution on each module (or subset of modules) first and then merge the results to obtain the global application behavior. The major benefit of this approach is the parallelization of the calculus; each module can be computed separately. But the major drawback is the risk of state explosion problem for a module (or a subset of modules) that will not occur if it was restricted by the behaviors of the others modules of the global application.

4.6 Constraint solvers
Once the execution tree is computed, the whole behaviour of the system is exhibited. A constraint solver may then be used to get the appropriate values for symbolic variables satisfying path conditions and generate numerical test input sequences. AGATHA actually uses Omega as constraint solver. We elect to generate one numeric test for each symbolic test. Each symbolic test represents an equivalence class of numeric tests; the constraints solver computes only one solution for each path condition.

4.7 Industrial applications
AGATHA has been successfully used to analyze large industrial specifications in various industrial contexts: industry of energy distribution [9], automotive industry [22] and aerospace industry [16].

Our team also took part in a European project, AIT-WOODDES ⁴ [10]. In that context, we worked with the industrial Peugeot-Citroën Automobile on an embedded navigation system for cars to automatically generate a set of tests for this UML specification with the AGATHA toolset. The execution model of UML for this project [17] was completely different than the one used here in the MUTATION project context.

5. Step 2 – Generating Test Cases
In this section, we present the second step of the approach and focus on the connexion between the test objectives (see section 3) and AGATHA presented in section 4, to obtain test cases. AGATHA combines the test objectives derived from requirements, and expressed with sequence diagrams, and the detailed design (expressed with class diagrams and activity diagrams) to generate test cases. We first present how UML is adapted to this purpose in the TAS context.

5.1 UML behavioural models
The UML Profile defined by TAS for their design models (refer to section 2) was adjusted, in the respect of UML notations, to enable the modelling of the detailed behaviour information required by AGATHA. Any TAS application is structured into several components that are decomposed into several categories which are themselves divided into classes. These classes are composed of methods the bodies of which hold the basic behaviour of the application. Two categories of diagrams are used to model behaviour in a TAS application design:

- diagrams for the specifying the cyclic activation of components, categories and classes;
- diagrams for specifying method bodies.

Sequence diagrams as presented in Figure 4 are used to model the activation of components, categories and classes; they may also correspond to a test objective as stated in Section 3.

4.6 Constraint solvers
Figure 4 - sequence diagrams for a component

In Figure 4, we represent one main operation called “activateC1” of a given component. This operation consists in a sequence of operation calls on categories, represented by the arrows. The possibility for representing guards was added to these diagrams so as to allow the full modeling of activation conditions, as required by AGATHA. These sequence diagrams are exploited by AGATHA to schedule the different operations calls when generating test cases.

Method bodies were not modelled in the initial TAS design model. Means for representing method bodies were added to the UML Profile so as to allow the provision of method behaviour information with sufficient details for an efficient exploitation by AGATHA to generate relevant tests. A double, textual and graphical representation was proposed: an action language, close to the code, each language construction being associated to a UML graphical pattern, using activity diagrams.

Applying the lessons of the ACCORD/UML [10] methodology, we define method bodies based on the “Action Language Notations”, which are used to define a mapping between a notation and the elements of the abstract syntax. An action language is proposed with composite actions, conditional notations, loop facilities, synchronous and asynchronous calls, etc….

We show an example of a conditional notation on the left side of the Figure 5 and its related graphical view on the right side.

```
Action 0;
    if (condition)
        Action 1;
        Action 2;
    endif
    Action 3;
```

Figure 5 - textual and graphical notation for a condition

When the user describes the method bodies by following the textual notations given to him, we can automatically derive the corresponding UML activity diagram that can thereafter be exploited by AGATHA to generate test cases.

Furthermore, the possibility for describing method bodies at different levels of detail is being investigated. In this article, we consider that the level of description of method bodies is as close as possible to the code.

We are using the Objecteering UML case tool. The UML model is exploited by AGATHA through a translation into the AEIOLTS format. A first benefit is that AGATHA can detect several types of deadlocks, livelocks and design errors in the design.

5.2 Functional test case synthesis

Tools such as AGATHA generate test sets allowing the validation of the conformance between the software implementation and its specification (black box testing).

A first exploitation of AGATHA addresses unit testing. AGATHA is fed models of methods or classes; test cases are synthesized, which allow conformance testing between the code and the design specification.

Functional testing is addressed through the exploitation of the test objectives produced at the first step of the approach (refer to section 3).

As mentioned in section 2, functional testing requires the test engineer to validate all the requirements for the implementation under test. Our approach is to provide the test engineer with a tool that allows him to select scenarios that validate a given requirement, from the symbolic execution graph generated by AGATHA.

Previous works with Peugeot-Citroën Automobile resulted in a methodology for the definition of criteria for functional test selecting [21]. This methodology proposes a property expression language which uses basic operators, logical and temporal connectors on the basic operators and operators on the selected scenarios sets. The basic operators have been built according to the experience of the test users.

The expressed property allows automatically selecting scenarios from the symbolic execution graph generated by the AGATHA toolset. In order to obtain a minimum number of scenarios, the exploration graph algorithm only goes through paths that contain at most the same symbolic node twice.

So, the result holds all the symbolic behaviours of the system. These behaviours satisfy the property specified by the user and contain at most twice the same symbolic node.

In the MUTATION project, the aim is to extend this technology to accommodate the test objectives derived from the textual RDL system requirements (refer to section 3), within the frame of the test strategy described in section 2.

We are adapting the tests objectives technology [8] to AGATHA. Test selection will be done “on the fly” and will be interlaced with the symbolic execution algorithm and the reduction procedures.

A difficulty of building functional tests is to avoid testing the same functionality several times and to choose the best moment for the best test. Firstly, it is essential that the generated test cases are mutually behaviourally separated; we can then guarantee to avoid the execution of two equivalent behaviours. Secondly, it is essential to optimise the execution sequence of scenarios. This optimisation is computed with initial and final states, and also with similar state sequences in the scenarios. Indeed, we want to reduce the number of times we go through a given state.

6. Conclusion and perspectives

Confronting efficient testing tools to realistic problems is always a fruitful challenging experience. In this paper, we have described a two-step approach that automates most of the process for deriving functional test cases for an industrial application.

The first step consists in expressing the requirements in the RDL formalism and completing, disambiguating and validating them through simulation. Test objectives are automatically derived from the RDL formalisation.

The second step synthesises functional test cases through combining the test objectives produced at step 1 and symbolic path computation on a detailed design model of the application. Test cases are produced through selecting symbolic behaviour paths that match the test objectives and applying a constraint solver to generate numeric tests.

Thanks to the complete automation of formal techniques, test engineers will be able to generate test cases in a transparent way, staying in the UML case tool used for modelling. The underlying complexity of formal methods is largely hidden.

This approach is based on three innovations at the core of our work: 1) definition of an appropriate language and a related methodology to define the test requirements; 2) symbolic execution on a UML action language; 3) combination of test objectives and symbolic execution to synthesize functional test cases.

The implementation of this approach is currently being completed. Future work involves the consolidation of the technology developed in MUTATION for RDL interpretation and test case synthesis so as to fully accommodate the industrial needs of the THALES context. In particular, we are considering the integration of new automatic simplification procedures in AGATHA, not working on generated expressions, but on the model itself, and based on abstraction principles.

Experiments on the TAS weapon navigation systems are under way; an industrial evaluation will be conducted to fully assess the actual scope and impact of this approach and to prepare decisions for a technical change in the TAS operational processes.

In particular, experiments will provide insight to prepare the redefinition of test strategies when introducing the MUTATION technology, and the reconsideration of practices of the test engineer.

Validation of UML specifications is also one of our perspectives. Several subjects are under consideration:
- Theorem proving, integrating property formalization and proof techniques.
- Compositional approaches.
- Scheduling analysis, considering the schedulability problem as a reachability problem.

7. References