NAT2TEST Tool: from Natural Language Requirements to Test Cases based on CSP

Gustavo Carvalho[1], Flávia Barros[1], Ana Carvalho[2]
Ana Cavalcanti[3], Alexandre Mota[1], and Augusto Sampaio[1]

[1] Universidade Federal de Pernambuco - Centro de Informática, 50740-560, Brazil
[2] Universidade Federal de Pernambuco - NTI, 50670-901, Brazil
[3] University of York - Department of Computer Science, YO10 5GH, UK
[1] {ghpc,fab,acm,acas}@cin.ufpe.br,
[2] ana.alves@ufpe.br,
[3] ana.cavalcanti@york.ac.uk

Abstract. Formal models are increasingly being used as input for automated test-generation strategies. However, typically the requirements are captured as English text, and these formal models are not readily available. With this in mind, we have devised a strategy (NAT2TEST) to obtain formal models from natural language requirements automatically, particularly to generate sound test cases. Our strategy is extensible, since we consider an intermediate and hidden formal characterisation of the system behaviour from which other formal notations can be derived. Here, we present the NAT2TEST tool, which implements our strategy.

Keywords: natural-language requirements, test-case generation, tool

1 Introduction

In 2009, the Federal Aviation Administration (FAA) published a report [7] that discusses current practices concerning requirements engineering management. It states that at the very beginning of system development, typically only natural-language (NL) requirements are documented.

In this light, we have investigated automatic strategies to obtain formal models from NL requirements aiming to generate sound test cases. Automation is essential for this task, since we cannot expect that practitioners will always have formal modelling knowledge. To accomplish our goal, we have devised a strategy (NATural language requirements to TEST cases – NAT2TEST) that generates test cases from NL requirements based on different internal and hidden formalisms: *Software Cost Reduction* – SCR (NAT2TEST$_{SCR}$ [3]), *Internal Model Representation* – IMR (NAT2TEST$_{IMR}$ [1]), and *Communicating Sequential Processes* – CSP (NAT2TEST$_{CSP}$ [4]).

Each instance of the NAT2TEST strategy has its own benefits and limitations. NAT2TEST$_{SCR}$ encodes the system behaviour as SCR specifications and, thus, one can use SCR-based tools, such as T-VEC[4], to generate test cases and

[4] http://www.t-vec.com/

test drivers. Although time can be manually encoded, it is not a native element of SCR specifications on T-VEC. Differently, NAT2TEST$_{IMR}$ translates requirements into the RT-Tester[5] internal notation, which natively considers discrete and continuous time representations. NAT2TEST$_{CSP}$ distinguishes itself by using refinement checking, instead of specific algorithms, for generating test cases. In such case, the test-generation approach can be proved sound. However, its performance might be worse than the one of specific algorithms.

Differently from previous works, where technical aspects of the NAT2TEST strategy are discussed, our focus here is on the NAT2TEST tool[6] that automates generation of test cases, particularly when using CSP as an internal and hidden formalism. Therefore, besides discussing implementation aspects, we provide here an overview of the functionalities supported by this tool. Section 2 presents an overview of our strategy. Section 3 details the NAT2TEST tool, including its user interface, functionalities and overall architecture. Section 4 addresses related work. Section 5 presents our conclusions and future work.

2 The NAT2TEST Strategy

Our strategy is tailored to generate tests for *Data-Flow Reactive Systems* (DFRS): a class of embedded systems whose inputs and outputs are always available as digital signals. The input signals can be seen as data provided by sensors, whereas the output data are provided to system actuators. These systems can also have timed-based behaviour, which may be discrete or continuous.

NAT2TEST receives as input system requirements written using the *SysReq-CNL*, a *Controlled Natural Language* (CNL) specially tailored for editing unambiguous requirements of data-flow reactive systems. As output, it produces test cases. Our test-generation strategy comprises a number of phases. The three initial phases are fixed: (1) syntactic analysis, (2) semantic analysis, and (3) DFRS generation; the remaining phases depend on the internal formalism.

The syntactic analysis phase receives as input the system requirements, and performs two tasks: it verifies whether these requirements are in accordance with the SysReq-CNL grammar, besides generating syntactic trees for each correctly edited requirement. The second phase maps these syntax trees into an informal NL semantic representation. Afterwards, the third phase derives an intermediate formal characterization of the system behaviour from which other formal notations can be derived (currently, SCR, IMR and CSP). The possibility of exploring different formal notations allows analyses from several perspectives, using different languages and tools, besides making our strategy extensible.

Here, we focus on the use of CSP to generate test cases. In this context, we have two additional phases. First, the DFRS model is encoded as CSP processes. Then, with the aid of the FDR[7] and Z3 tools[8], test cases are generated.

[5] `https://www.verified.de/products/rt-tester/`
[6] **Available for download at**: `http://www.cin.ufpe.br/~ghpc/`
[7] FDR tool – `http://www.cs.ox.ac.uk/projects/fdr/`
[8] Z3 tool – `http://z3.codeplex.com/`

3 The NAT2TEST Tool

The tool is written in Java (it is multi-platform), and its Graphical User Interface (GUI) is built using the Eclipse RCP[9]. Figure 1 shows the tool interface.

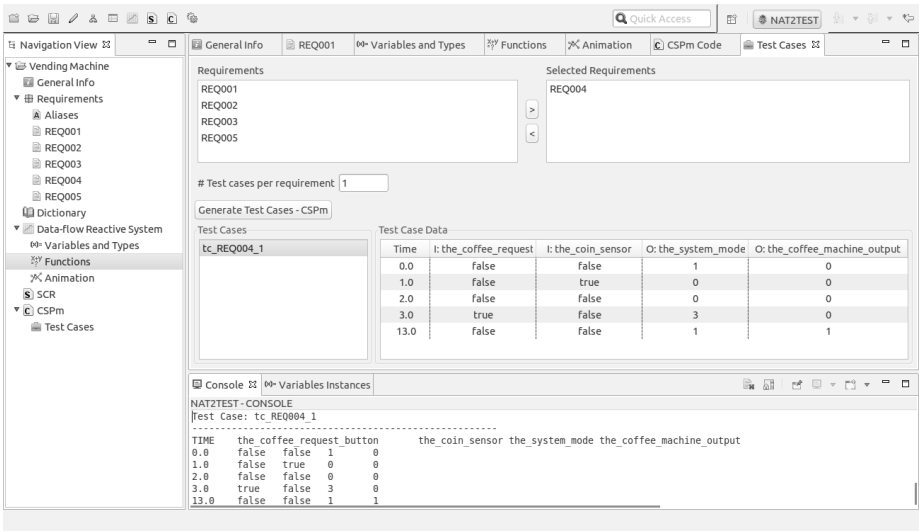

Fig. 1: The NAT2TEST tool

Each phase of the strategy is realised by a different component. Figure 2 shows a diagram of the tool architecture, which follows a traditional layered structure: presentation, business, and data layers. The first one comprises editors that interact with the business layer via the LocalFacade. The business layer has a set of controllers that are responsible for interacting with the components that realise each phase of the strategy. Besides that, it also persists data (i.e., requirements and dictionaries) via Business Objects (BO) and Data Access Objects (DAO). We do not persist other elements (e.g., the DFRS model), as they can be automatically derived from the requirements very efficiently.

An explanation on how to use the tool is available on its help. In the following sections we describe each component in terms of implementation details and functionalities provided. To illustrate the tool, we consider a Vending Machine (VM) (adapted from [9]). Initially, the VM is in an *idle* state. When it receives a coin, it goes to the *choice* state. After inserting a coin, when the coffee option is selected, the system goes to the *weak* or *strong* coffee state. If coffee is selected within 30 seconds after inserting the coin, the system goes to the *weak coffee* state. Otherwise, it goes to the *strong coffee* state. The time required to produce a weak coffee is also different from that of a strong coffee.

[9] `http://wiki.eclipse.org/index.php/Rich_Client_Platform`

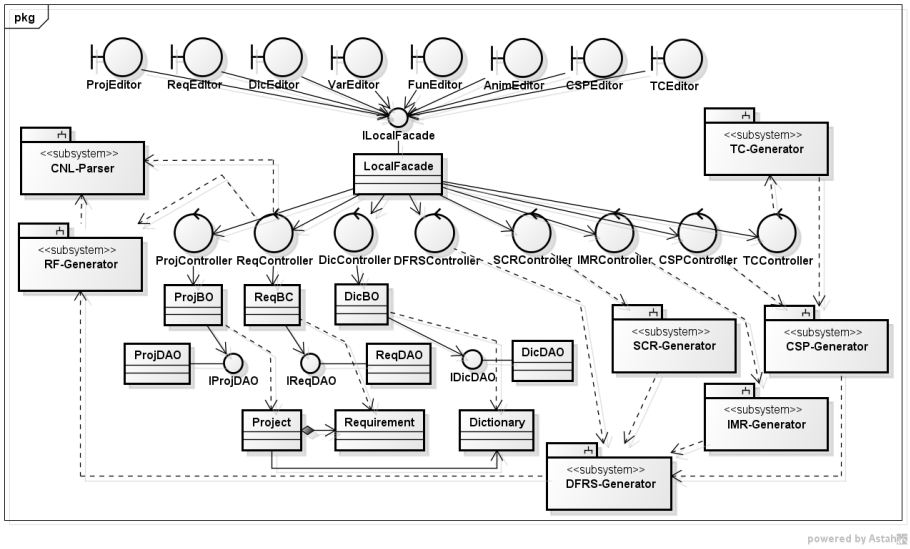

Fig. 2: The NAT2TEST tool architecture.

3.1 CNL-Parser Component

The CNL-Parser analyses the system requirements according to the SysReq-CNL grammar, yielding the corresponding syntax trees. This CNL allows writing requirements that have the form of action statements guarded by conditions [3]. For a concrete example, consider the following valid requirement for the VM: "*When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode*".

First, each word is classified into its corresponding lexical class by a POS-Tagger (Parts-Of-Speech Tagger), based on a domain-specific dictionary. In NL the same lexeme may bear more than one classification (e.g., "change" may be a noun or a verb). In our work, we implemented a customized POS-Tagger that searches all possible classifications of each lexeme. For parsing we implemented a version of the Generalized LR (GLR) algorithm [12]. It generalizes the traditional LALR (Look-Ahead LR parser) algorithm to handle non-deterministic and ambiguous grammars. When the parser identifies more than one possible syntax tree, the user needs to remove the ambiguity before proceeding.

The tool provides other functionalities, such as editing the domain-specific dictionary, besides using aliases to promote text reuse (in Figure 1, by clicking on *Dictionary* and *Aliases*, respectively). It is also capable of assisting the user while writing the requirements by informing the next expected grammatical classes.

3.2 RF-Generator Component

The second processing phase receives as input the generated syntax tree, and delivers the requirement semantic representation. In this work, we adopt the

Case Grammar theory [8] to represent meaning. In this theory, a sentence is analysed in terms of the semantic (Thematic) Roles (TR) played by each word, or group of words in the sentence. The verb is the main element of the sentence, and it determines its possible semantic relations with the other words in the sentences, that is, the role that each word plays with respect to the action or state described by the verb.

The verb's associated TRs are aggregated into a structure named as Case Frame (CF). Each verb in a requirement NL specification gives rise to a different CF. All derived CFs are joined afterwards to compose what we call a *Requirement Frame* (RF). In this work, we consider nine thematic roles [3], for instance, agent: entity who performs the action; patient: entity who is affected by the action; and to-value: the patient value after action completion.

This component is implemented using the visitor pattern to analyse the syntax trees, considering the inference rules defined in [3], which associate words with the corresponding TRs. In Figure 1, one can see the inferred TRs for a given requirement by clicking on the respective requirement identifier (e.g., *REQ0001*).

3.3 DFRS-Generator Component

The DFRS model [2] provides a formal representation of the requirements semantics, which has a symbolic and an expanded representation. Briefly, the symbolic version is a 6-tuple: (I, O, T, *gcvar*, s_0, F). Inputs (I) and outputs (O) are system variables, whereas timers (T) are used to model temporal behaviour. The global clock is *gcvar*, a variable whose values are non-negative numbers representing a discrete or a dense (continuous) time. The element s_0 is the initial state. The last element (F) represents a set of functions, each one describing the behaviour of one system component. The expanded DFRS comprises a (possibly infinite) set of states, and a transition relation between states. This expanded representation is built by applying the elements of F to the initial state to define *function transitions* and letting the time evolve to define *delay transitions*.

The symbolic DFRS is automatically generated by the DFRS-Generator, which identifies its constituent components from the RFs. First, variables (inputs, outputs and timers) are obtained from the contents of the thematic role *patient*. Their types are inferred considering the values mentioned by roles such as the *to-value*. Then, we create an initial state considering initial default values (like 0 for *integers*, and *false* for *booleans*, for instance). Nevertheless, the tool allows the user to edit the initial values.

Afterwards, we encode the conditions and actions described by the requirements as functions. The tool keeps traceability information between the requirements and the function entries. The requirement shown in Section 3.1 is encoded as the guard: $\neg\ (prev(the\_coin\_sensor) = true) \wedge the\_coin\_sensor = true \wedge the\_system\_mode = 1$, where 1 represents the *idle state*, and *prev* denotes the value in the previous state), and the following assignments $the\_request\_timer := gc, the\_system\_mode := 0$, where *gc* refers to the system global clock, and 0 to the *choice* state. The tool also supports validation of the requirements by animating DFRS models (in Figure 1, by clicking on *Animation*).

3.4 CSP$_M$-Generator Component

This component encodes DFRSs as CSP processes. It describes in CSP how the expanded DFRS is obtained from the symbolic one. First, processes are created to represent a shared (global) memory, which comprises the values of the DFRS inputs and outputs. Time is modelled symbolically to prevent state explosion when compiling the CSP specification and generating the corresponding LTS. When some behaviour depends on the amount of time elapsed, we just assume that the delay occurred satisfies the temporal constraints, and we perform a specific event to represent this assumption. Later, we use Z3 to find concrete values for delays that satisfy these constraints (see Section 3.5).

The tool creates a CSP process for each function of the symbolic DFRS. We also keep traceability with the original requirements by means of events named after their identifier. When these events occur, it implicitly states that the system is presenting the behaviour described by the corresponding requirement. Besides being one of our alternatives for generating test cases, the CSP model allows the automatic verification of important properties concerning the requirements, and thus providing more confidence in the system specification, namely: completeness, consistency, and reachability. More information is available in [4]. In Figure 1, one can see the obtained CSP specification by clicking on *CSPm*.

3.5 TC-Generator Component

This component accomplishes the ultimate goal of the NAT2TEST strategy: the generation of test cases. It is done in two steps: (1) the enumeration of *symbolic* test cases via FDR, and (2) the instantiation of time-related events via Z3. The enumeration of test cases is performed with the aid of a TCL[10] script, which is based on the traces enumeration technique presented in [10].

Due to the potential large (possibly infinite) number of test cases, we consider coverage criteria (e.g., maximum number of test cases, coverage of nodes or transitions of the LTS, requirement coverage) to guide the test-generation process. Here, we consider requirement coverage: one can select which requirements should be covered by the generated test cases. To meet this criterion the tool searches for traces that have the event named after the requirement identifier.

Using FDR, the NAT2TEST tool enumerates traces that meet the coverage criteria. Basically, we can split the events of these traces into three distinct groups: input, output, and time-related events (delays and resets). From the first two, the tool infers the stimuli provided to the system, as well as the expected response. In this way, we obtain a symbolic test case as it still lacks time information. The proper test case is obtained with the aid of Z3. From the reset and delay events we automatically generate a satisfiability problem. More specifically, there is a mapping from each time-related event that appears in the trace to a time constraint that needs to be fulfilled. Z3 is then used to find solutions (delays) that satisfy these constraints.

[10] `http://www.tcl.tk/`

Figure 1, presented in Section 3, shows the screen where the user can select which requirements the test cases are going to cover, as well as inspect the generated test cases, which are presented in a tabular form. The test case depicted in Figure 1 tests the following scenario: first, the coin sensor becomes true (1.0s), leading the system to the *choice* state (*the_system_mode* = 0). Later (3.0s), the user presses the coffee request button (*the_coffee_request* = true); after 10 seconds, the machine produces weak coffee (*the_coffee_machine_output* = 1).

4 Related Work

In the related literature, other approaches generate test cases from NL specifications. In [5], requirements are written in the quasi-natural language Gherkin. Tests are generated with the aid of a model-based testing tool. In order to obtain executable test cases, clauses from the specification are manually associated with code, which is not required by us. Nevertheless, we generate executable test cases, since they represent data to be sent and monitored from sensors and actuators. Furthermore, we also consider time aspects when generating tests. While [5] addresses test generation for web applications, we focus on embedded systems.

In [11], after defining a dictionary, test cases are generated from plain text, with no need of an underlying CNL, which brings flexibility, but also more user intervention. It is necessary to identify and partition system inputs and outputs manually. In our work, they are automatically identified from thematic roles. Similarly to our approach, time is considered as an element of testing in [11].

Some works impose a more standardised writing form and, thus, rely on less user intervention. In [6] requirements need to be written according to a strict if-then template, which, however, can be used to represent time properties, besides generating tests. In our work, the SysReq-CNL provides a more flexible writing structure. In [10] a similar sentence structure is also considered. However, it generates non-executable test cases, besides not considering time aspects.

The absence of user intervention in our strategy is due to the compromise reached by the SysReq-CNL. As we focus on the domain of embedded systems, whose behaviour can be described as actions guarded by conditions, we can impose some restrictions, while allowing the requirements to be expressed as a textual specification. However, these restrictions make our approach not suitable for writing requirements that do not adhere to this format of actions and guards.

5 Conclusions

We presented the NAT2TEST tool, which supports the automatic generation of test cases from natural-language requirements, which might consider discrete or continuous temporal properties. This is achieved possibly using commercial tools (like T-VEC and RT-Tester) or based on a formal conformance relation using tools like FDR and Z3, in which case the test generation is proved sound. As future work, we envisage the following tasks: (1) apply compression and

optimisation techniques to enhance the performance of our strategy, and (2) extend our approach to consider NL descriptions of hybrid systems.

Acknowledgments. This work was carried out with the support of the CNPq (Brazil), INES[11], and the grants: FACEPE 573964/2008-4, APQ-1037-1.03/08, CNPq 573964/2008-4 and 476821/2011-8.

References

1. Carvalho, G., Barros, F., Lapschies, F., Schulze, U., Peleska, J.: Model-Based Testing from Controlled Natural Language Requirements. In: Artho, C., lveczky, P.C. (eds.) Formal Techniques for Safety-Critical Systems, Communications in Computer and Information Science, vol. 419, pp. 19–35. Springer International Publishing (2014)
2. Carvalho, G., Carvalho, A., Rocha, E., Cavalcanti, A., Sampaio, A.: A Formal Model for Natural-Language Timed Requirements of Reactive Systems. In: Merz, S., Pang, J. (eds.) Formal Methods and Software Engineering, International Conference on Formal Engineering Methods ICFEM, Lecture Notes in Computer Science, vol. 8829, pp. 43–58. Springer International Publishing (2014)
3. Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., Blackburn, M.: NAT2TEST$_{SCR}$: Test case generation from natural language requirements based on SCR specifications. Science of Computer Programming 95, Part 3(0), 275 – 297 (2014)
4. Carvalho, G., Sampaio, A., Mota, A.: A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. In: Formal Methods and Software Engineering, LNCS, vol. 8144, pp. 148–164. Springer Berlin Heidelberg (2013)
5. Colombo, C., Micallef, M., Scerri, M.: Verifying Web Applications: From Business Level Specifications to Automated Model-Based Testing. In: Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014. pp. 14–28 (2014)
6. Esser, M., Struss, P.: Obtaining Models for Test Generation from Natural-Language like Functional Specifications. In: International Workshop on Principles of Diagnosis. pp. 75–82 (2007)
7. FAA: Requirements Engineering Management Findings Report. Tech. rep., U.S. Department of Transportation - Federal Aviation Administration (2009)
8. Fillmore, C.J.: The Case for Case. In: Bach, Harms (eds.) Universals in Linguistic Theory, pp. 1–88. New York: Holt, Rinehart, and Winston (1968)
9. Larsen, K., Mikucionis, M., Nielsen, B.: Online Testing of Real-time Systems using Uppaal: Status and Future Work. In: Perspectives of Model-Based Testing - Dagstuhl Seminar. vol. 04371 (2004)
10. Nogueira, S., Sampaio, A., Mota, A.: Test generation from state based use case models. Formal Aspects of Computing 26(3), 441–490 (2014)
11. Santiago Junior, V., Vijaykumar, N.L.: Generating Model-based Test Cases from Natural Language Requirements for Space Application Software. Software Quality Journal 20, 77–143 (2012)
12. Tomita, M.: Efficient Parsing for Natural Language. Kluwer Academic Publishers (1986)

[11] `www.ines.org.br`