Test Case Generation from Natural Language Requirements Using CPN Simulation

Bruno Cesar F. Silva(✉), Gustavo Carvalho, and Augusto Sampaio

Centro de Informática, Universidade Federal de Pernambuco,
Recife 50740-560, Brazil
{bcfs,ghpc,acas}@cin.ufpe.br

Abstract. We propose a test generation strategy from natural language (NL) requirements via translation into Colored Petri Nets (CPN), an extension of Petri Nets that supports model structuring and provides a mature theory and powerful tool support. This strategy extends our previous work on the NAT2TEST framework, which involves syntactic and semantic analyses of NL requirements and the generation of Data Flow Reactive Systems (DFRS) as an intermediate representation, from which target formal models can be obtained for the purpose of test case generation. Our contributions include a systematic translation of DFRSs into CPN models, besides a strategy for test generation. We illustrate our overall approach with a running example.

Keywords: Test generation · CPN · Model simulation

1 Introduction

Since the so-called *software crisis* (in the Sixties), when the term *software engineering* originated, difficulty of understanding the user needs, ambiguous specifications, poorly specified and interpreted requirements are still common issues. In the field of requirements engineering, several studies have been conducted focusing on the use of (semi-)formal methods to specify, model and analyse requirements, besides automatically generating test cases, resulting in greater maturity and in the construction of underlying theories and supporting tools.

According to Myers [14], as formal methods can be used to find errors in requirements such as inconsistency and incompleteness, they play an important role in the earlier discovery of errors and, thus, reducing costs and correction effort. In this light, we have devised a strategy [2] that generates test cases from natural language requirements. In this strategy, the system behaviour is formally represented as a Data-flow Reactive System (DFRS). Later, this model is translated into a target formalism for generating test cases. The purpose of the NAT2TEST tool is to be easily extensible to various target formalisms.

Although the use of CSP [6], SCR [5] and IMR [3] have been explored, each one has advantages and limitations. For instance, test generation from SCR and IMR is based on commercial tools, which is a practical advantage, but, on the

M. Cornélio and B. Roscoe (Eds.): SBMF 2015, LNCS 9526, pp. 178–193, 2016.
DOI: 10.1007/978-3-319-29473-5_11

other hand, is not formal, as no explicit conformance relations are considered. Differently, the test strategy for CSP is formal and can be proved sound, with an explicit conformance notion. In this case, the FDR tool[1] is used to generate test cases as counterexamples of refinement verifications and, thus, it demands the complete expansion of the underlying label transition system, which can easily lead to state explosion problems.

Therefore, here we explore the use of Colored Petri Nets (CPN) as an alternative extension of the NAT2TEST strategy where simulation of CPNs is used for generating test cases. To simulate a CPN it is not necessary to explore its complete state space first and, thus, we minimise state explosion problems. In other words, we are capable of generating test cases for more complex systems since it is not necessary to enumerate all its states first, but they can be computed dynamically during simulation.

Also, we benefit from the diversity and maturity of CPN tools[2]. For instance, besides simulation we can also perform model checking. Another promising opportunity is to make use of the CPN ML language, a functional programming language based on Standard ML. Furthermore, a testing theory based on CPN can also be entirely formal, although our focus here is on its feasibility. The main contributions of this paper are the following:

- A systematic translation of DFRSs into CPN models;
- A strategy for test case generation from CPN models via simulation;
- An example to illustrate our overall approach.

Section 2 introduces the NAT2TEST strategy and CPNs. Section 3 explains how to generate CPN models from DFRSs. Section 4 describes how to generate test vectors from CPN models via simulation. Section 5 addresses related work, and presents our conclusions and future directions.

2 Background

Here, we provide a brief explanation of the NAT2TEST strategy and CPNs.

2.1 NAT2TEST Strategy

In this section we briefly describe the *NAT2TEST* (NATural language requirements to TEST cases) strategy for generating test cases from requirements written in natural language. It 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.

It 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.

[1] http://www.cs.ox.ac.uk/projects/fdr/.
[2] http://cpntools.org/.

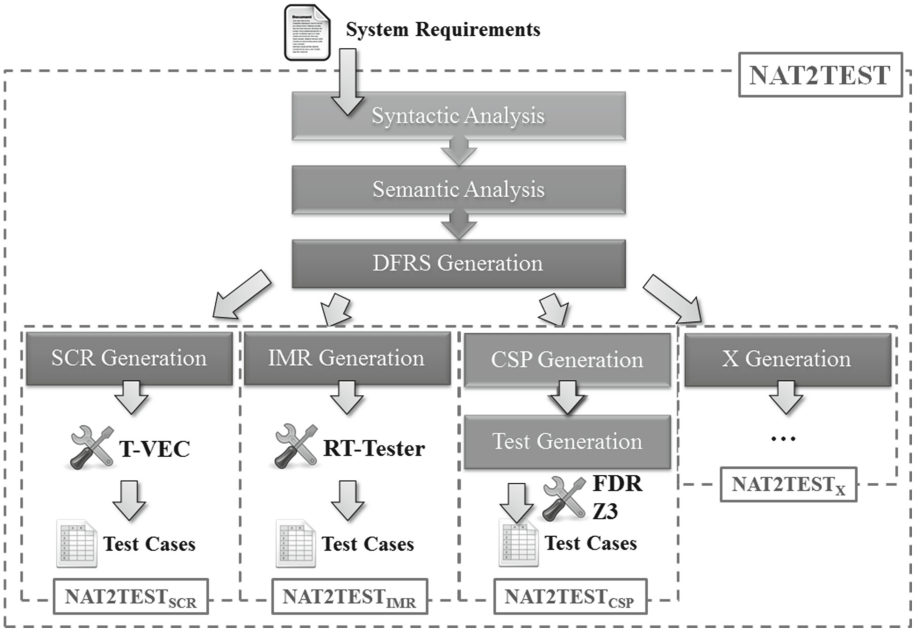

Fig. 1. NAT2TEST strategy

This test-generation strategy comprises a number of phases (see Fig. 1). 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 (DFRS) from which other formal notations can be derived. The possibility of exploring different formal notations allows analyses from several perspectives, using different languages and tools, besides making the strategy extensible, as discussed in the previous section.

Data-Flow Reactive Systems. A DFRS model [4] provides a formal representation of the requirements semantics. It 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. There is a global clock denoted $gcvar$. 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*, but also letting the time evolve to define *delay transitions*. This expanded representation can be seen as a semantics for symbolic DFRSs.

Figure 2 illustrates a DFRS for a simplified version of the control system for safety injection in a nuclear power plant (NPP) described in [11]. If the water pressure is too low (lower than 9 units), the system injects a coolant into the reactor. In Fig. 2, one can note that the corresponding DFRS has one input and one output variable (*the_water_pressure* and *the_pressure_mode*, respectively).

The system behaviour is captured by the function *the_safety_injection_system*. It has four entries. Each entry comprises a 3-tuple: a static guard, a timed guard (here, empty), and the expected system reaction (a list of assignments). The possible pressure modes are *high*, *low*, and *permitted*. While these values are represented in the DFRS as numbers ($high \mapsto 0, low \mapsto 1, permitted \mapsto 2$), they are represented in the corresponding CPN as strings for legibility purposes.

In Fig. 2, the first requirement (REQ001) captures the system reaction when the current pressure mode is low ($the\_pressure\_mode = 1$), and the water pressure becomes greater than or equal to 9. As in this example the system reaction does not depend upon the time elapsed, we only have static guards. When the situation previously described happens, the pressure mode changes to permitted ($the\_pressure\_mode := 2$). Similarly, the other requirements describe when the pressure mode changes to (from) low, permitted, and high.

Kind	Name	Type	Expected Values	Initial Value
INPUT	the_water_pressure	INTEGER	{0, 10, 9}	0
OUTPUT	the_pressure_mode	INTEGER	{high, low, permitted}	high
GLOBAL CLOCK	gc	FLOAT	-	

Static Guard	Timed Guard	Statements	Requirement Traceability
◢ **Function: the_safety_injection_system**			
the_pressure_mode = 1 AND ¬(prev(the_water_pressure) >= 9) AND the_water_pressure >= 9		the_pressure_mode := 2	REQ001
the_pressure_mode = 2 AND ¬(prev(the_water_pressure) >= 10) AND the_water_pressure >= 10		the_pressure_mode := 0	REQ002
the_pressure_mode = 2 AND ¬(prev(the_water_pressure) < 9) AND the_water_pressure < 9		the_pressure_mode := 1	REQ003
the_pressure_mode = 0 AND ¬(prev(the_water_pressure) < 10) AND the_water_pressure < 10		the_pressure_mode := 2	REQ004

Fig. 2. An example of DFRS (variables, types and functions)

2.2 Colored Petri Nets

Colored Petri Nets (CPN) are an extension of high-level Petri nets – a graphical modelling language with formally defined syntax and semantics. As described in [8], the formal definition of a CPN is given by a 9-tuple (Σ, P, T, A, N, C, G, E, I), satisfying the following properties:

– Σ is a finite set of non-empty *color sets* (types).

- P is the finite set of all *places*.
- T is the finite set of all *transitions*.
- A is the finite set of all *arcs* such that $P \cap T = P \cap A = T \cap A = \emptyset$
- N is the *node* function such that $N : A \rightarrow P \times T \cup T \times P$
- C is the *color* function such that $C : P \rightarrow \Sigma$
- G is the *guard* expression function such that $G : T \rightarrow Expr$, and $\forall t \in T : [Type(G(t)) = Bool \wedge Type(Var(G(t))) \subseteq \Sigma]$
- E is the *arc expression* function such that $E : A \rightarrow Expr$, and $\forall a \in A : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$, where $p(a)$ is the place of $N(a)$;
- I is the *initialization* function such that $I : P \rightarrow Expr$, and $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$

We denote by $Expr$ the set of expressions provided by the inscription language (CPN ML), and by $Type(e)$ the type of an expression $e \in Expr$, i.e., the type of $Var(e)$ (the values obtained when evaluating e). The subscript MS denotes the multiset of the associated place.

Informally, a CPN model expresses both the states and the events of a system. Events are represented by transitions (drawn as rectangles) and the states are represented by places (drawn as ellipses or circles). These two kinds of nodes are connected by arcs that indicate the direction in which tokens (data values) are moved. Arcs are used to connect places with transitions, and vice-versa, but never two places or two transitions. Places hold collections of tokens and, thus, represent local states (markings). The global state (or global marking) of a model is represented by the distribution of tokens throughout its places. The places also have an initial marking representing the initial local state. Figure 3 shows part of the CPN obtained for the simplified NPP (explained in Sect. 3).

As described in [15], a transition typically represents an event that occurs in the model. When this event occurs, the transition is said to fire. When a transition is enabled it means that it is ready to fire. When a transition fires, it is able to remove tokens from its input places and to produce new tokens on its output places. The inscriptions in the arcs leading to a transition define the quantity of tokens needed for enabling the transition. Moreover, qualitative constraints can be specified for the tokens from the input places. This can be done by the use of pattern matching or guards in the transition.

A guard is a predicate that must evaluate to true for enabling the transition. For a given transition, if it is possible to select a set of tokens from the input places while satisfying these constraints, this transition is enabled. If another transition also depends on some of the same tokens for being enabled, the transitions are said to be in conflict. This property is part of what makes (Colored) Petri Nets a very strong tool for modelling distributed and concurrent systems with shared resources, parallel processes and other characteristics that come with this type of systems.

The graphical definition of CPNs is supplemented by declarations of functions, operations, constants, variables, color sets and priorities; all of them in a functional programming language (CPN ML [9]): an extension of the more

commonly known Standard ML [13]. Therefore, CPN models differ from traditional Petri Nets by the fact that the tokens have types (*color sets*) and values.

The CPN modelling language also supports the specification of hierarchically structured models as a collection of connected modules (or pages). This makes it possible to work with different levels of abstraction. A module is itself a CPN model (possibly hierarchical too). Structuring is performed through two mechanisms: fusion places or substitution transitions. Here, we use the former. A fusion place is a set containing multiple places that may be found in different modules. This allows interaction to traverse boundaries of the modules in the model.

CPN also allows the specification of time-based behaviour [10]. This is accomplished adding time stamps to tokens. In this way, delays are expressed as timing annotations in arcs and transitions. Although the DFRS is able to deal with discrete and continuous time representation, we are not dealing with time in our CPN representation yet.

3 CPN Model Generation

Intuitively, the behaviour of a DFRS is encoded as transitions that modify the corresponding output variables, which are modelled as CPN places. Therefore, the target of these transitions are these output places, and their source are auxiliary places that provide to the transitions the system inputs. The generation of CPN models from DFRSs is accomplished in three steps.

In the first step, input variables and their types are translated to CPN variables and color sets, respectively. In second step, output variables are mapped to places along with the corresponding color set. In the last step, the DFRS functions are represented in the CPN model as transitions. In what follows, we detail these steps. To illustrate the relevant concepts, here and in the next section, we consider the simplified NPP briefly described in Sect. 2.1.

Representing Input Variables. For each input variable of the DFRS, a CPN variable is created along with the corresponding color set (type). Sometimes, the system reaction depends upon the current, but also the previous input value. In such cases, besides creating a variable to represent the system input, we also create another variable to store the previous input value. For instance, considering the simplified NPP, we create two variables to represent the water pressure: *wp* (*water pressure*) and *pwp* (*previous water pressure*), whose type (color set) is *INTWP* – an integer ranging from 0 to 11.

Representing Output Variables. In this case, they are represented by places to denote the system state, along with variable for transmitting the previous and current values, similarly to input variables. Color sets are also created to represent the type of the output variables. When transitions that modify an output variable are fired, it leads to the corresponding place changing its marking and, thus, the value of the corresponding CPN variable accordingly. A place has as initial marking the initial value of the corresponding output variable.

Considering the simplified NPP example, we have a single output variable that represents the pressure mode. There are three possible modes: *high*, *low*, and *permitted*. Therefore, we create a place (*Pressure Mode*), a variable (*pm* – *pressure mode*), and a color set (*PRESSURE* – an enumeration comprising these three values). This place is reached, for instance, when the transition *REQ001* is fired (see Fig. 3). As explained in what follows, CPN transitions are created to represent the system requirements. The arc leading to the place *Pressure Mode* has as inscription *permitted*, since the requirement REQ001 describes the situation when the pressure mode becomes permitted.

Representing Functions. For each entry of a DFRS function, a page is created to represent the behaviour of this entry. It simplifies visual inspection of the model, since each page details just one possible system reaction. For easier traceability of CPN pages to the requirements, the page is named after the requirement related to the considered entry.

Each page comprises some places and one transition. The transition (named after the same requirement) represents a possible system reaction – how the output variables should be updated. Therefore, this transition has outgoing arcs to one or more places (the ones that represent output variables) that it should modify when fired. Besides the places that represent output variables, there are four other places (*Input*, *Next Input*, *Next TO*, and *Test Oracles*) that are auxiliary elements used for generating test cases (later explained in Sect. 4.1).

To give a concrete example, consider the simplified NPP and the entry related to *REQ001*: $(the\_pressure\_mode = 1 \wedge \neg(prev(the\_water\_pressure) >= 9) \wedge the\_water\_pressure >= 9, \emptyset, the\_pressure\_mode := 2)$. It means that when the pressure mode is 1 (*low*), and the water pressure is greater than or equal to 9, but it was not greater than or equal to 9 in the previous state (*prev*), the system shall change the pressure mode to 2 (*permitted*). We note that, as previously said, string enumerations are represent in DFRSs as numbers, whereas we use the original strings in the CPN for legibility purposes.

For this entry, we create the page *REQ001*. In this page, there is a transition *REQ001*. To be enabled, the guard associated to this transition needs to evaluate to true. This guard $([not(pwp >= 9), (wp >= 9), (pm = low), (n = i)])$ is a direct translation of the guards of the corresponding function entry. The variable *pwp* stands for the previous value of the water pressure. The last element of this guard $(n = i)$ is required to ensure the order in which this transition processes the system inputs (later explained in Sect. 4.1).

Besides satisfying its guard, a transition may also need to know the current and previous values of the system inputs, along with the value of the system outputs. Note that the aforementioned guard also depends upon the value of a system output – the pressure mode (*pm*). These values (tokens) are sent to the transition *REQ001* by the arcs emanating from the places *Input* and *Pressure Mode*. When this guard evaluates to true, and the required tokens are available, the transition fires and assigns permitted to the pressure mode. As previously said, for every entry of each DFRS function, a page similar to this one is created.

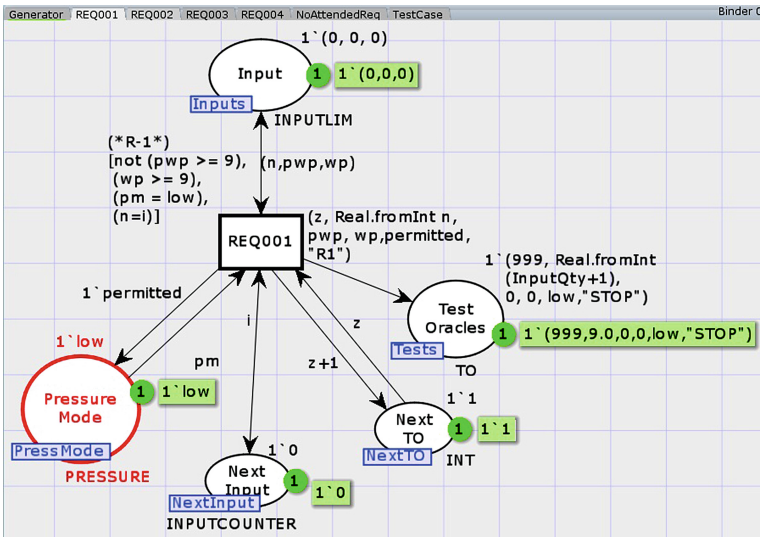

Fig. 3. Page created to represent the requirement REQ001

4 Generating Test Vectors Based on CPN Simulation

In order to generate test cases via CPN simulation, it is necessary to complement the model generated in the previous section with some auxiliary elements. This is the subject of Sect. 4.1. Section 4.2 presents how test vectors are generated via simulation of a CPN.

4.1 Auxiliary Components

After constructing a CPN to represent a DFRS (see Sect. 3), we now create auxiliary components, which are used to generate test cases. These components are always the same, but parametrized by the number of inputs and outputs. Considering the constant *InputQty*, which is defined beforehand to limit the number of generated inputs, the following auxiliary elements are created.

Generating Entries. A page (*Generator*) is created to generate input values randomly. When fired, the transition *Generate Inputs* generates the system inputs, and transmits them, along with an index i, to the place *Input* (see Fig. 4). This index is used to ensure the order the generated inputs are consumed by the transitions that represent the system behaviour. This index is initially 0, and is incremented with the aid of the place *Next*. We note that the generated inputs are also an input to the transition *Generate Inputs*. It is necessary because when generating the next inputs, the token sent to *Inputs* might comprise the newly generated values, but also the previous values of the inputs. When the number of generated inputs reaches *InputQty*, the transition *Generate Inputs* becomes

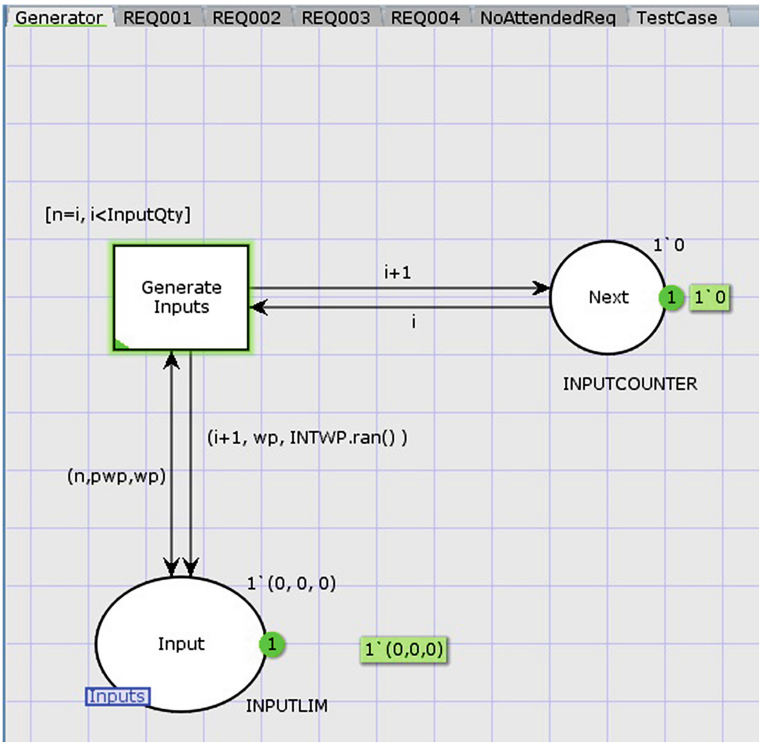

Fig. 4. Page created to generate the inputs randomly

disabled. At this moment, all generated inputs (tokens) are stored in *Input*, and ready to be consumed by the other transitions.

Dealing with No System Reaction. A page (*NoAttendedReq*) similar to those that represent the requirements (DFRS functions) is created. Unlike those, although it has a transition (*noReq*) that consumes the inputs (tokens in the place *Input*) and increments the input ordering index, this transition does not change the value of the places that represent output variables. This transition is enabled when no other transition is enabled, and it represents the idea that when no system reaction is expected for a given input, the system does not change its outputs (their values remain the same). To ensure that *noReq* is only enabled when no other transition is enabled, it has a lower priority (*P_LOW*).

Ordering the Generated Test Vectors. While the place *Next Input* is used to define the order in which the inputs are processed, the place *Next TO* is used to order the obtained test vectors. When a transition related to a requirement (or the *noReq*) fires, it produces a token that represents a test vector: the received inputs, along with the expected results. As one can see in Fig. 5, this token begins with an index z. This index, which is controlled by *Next TO*, establishes

an order for the tokens stored in *Test Oracles*. Another element of this tuple is a label to track which requirement produced the output. This label allows us to relate generated test vectors with requirements and, thus, extract coverage information with respect to the system requirements.

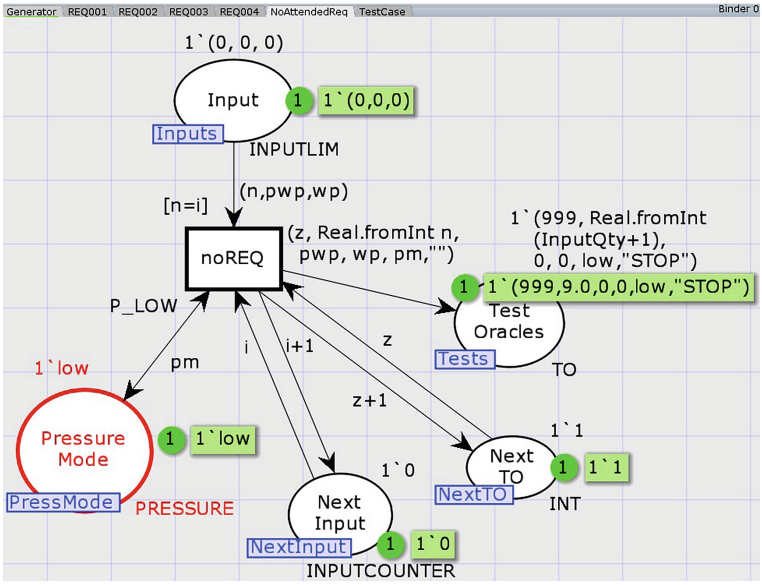

Fig. 5. Page created to deal with no system reaction

Creating Test Case File. The page *TestCase* stores the steps (test vectors) of the generated test cases in a csv file. This page has two transitions to manipulate the csv files: *Write File* and *Close File*. The transition *Write File* is enabled when all inputs are processed, that is, when the index i (received from *Next Input*) is greater than the defined number of inputs of the test case. The created test vectors, which are stored in the place *Test Oracles* and ordered by the index z, are sent to this transition when it fires.

When fired, this transition executes the associated code segment (shown below) that creates a csv file. First, it creates a file (*TC.csv*), and adds to it a header – the name of the columns, which are named after the system input and output variables. Afterwards, it appends to this file a test vector every time the transition *Write File* is fired. The other transition (*Close File*) has lower priority and is enabled when all outputs are processed. When it fires, it closes the file previously created. These transitions are shown in Fig. 6.

```
if r=0.0 then outfile := TextIO.openOut("TC.csv") else ();
if r=0.0 then TextIO.output(!outfile,Head) else ();
if r<=(Real.fromInt (InputQty)) then
   TextIO.output(!outfile, saida (r,wp,pm,req)) else ();
```

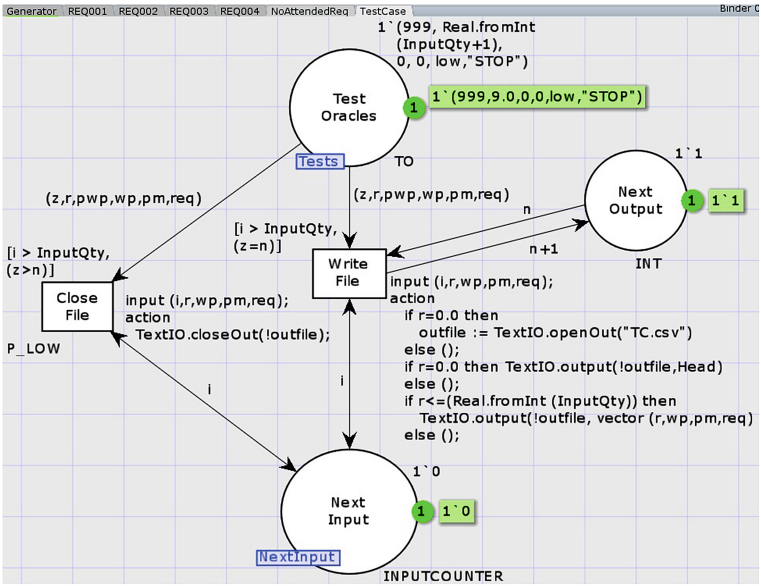

Fig. 6. Page created to save the test cases in a file

Repeated Places. Fusion sets are created for places that are repeated in more than one page. For instance, the place *Input* is contained in all pages, except in *TestCase*, so the fusion set *Inputs* is associated to all the places *Input* (see a thin rectangle at the bottom of this place in Figs. 3, 4 and 5). The same occurs to the places *Test Oracles*, so the fusion set *Tests* is created. This place does not occur in the page *Generator*.

The simplicity behind the creation of these auxiliary components, which are in essence the same despite the system being modelled, shows how it is straightforward to extend the CPN that models a DFRS for the purpose of generating test cases.

4.2 Test Vectors Generation

Once generated the model, as previously explained in Sect. 3, it is possible to generate test cases automatically via CPN simulation: it randomly produces inputs, along with the expected outputs. The tool (CPN Tools) has a number of resources to run simulations; for instance, it can be performed with user intervention or automatically, it can also be repeated n times (simulation replications). Besides generating test cases, simulations can be used to validate the system requirements.

The proposed model is constructed so that the simulation takes place in three steps: generation of random inputs, processing the inputs by the transitions obtained from the system requirements (or by the transition *noREQ*), and recording the test case file. These steps are detailed below.

Entries Generation. In the first step, the inputs are randomly produced by using the *ran* function. As initial values are considered those defined in the DFRS, which are reflected as the initial marking of the place *Input*. The number of inputs is initially preset and stored in the constant *InputQty*. The entries are sorted with the support of place *Next*. Considering the NPP example, Fig. 7 shows 7 entries randomly generated. Initially, the water pressure is 0. Later, it is 10, 4, 7, 2, 3, 1, 11. We note that the second element of the generated tokens refers to the previous value of the water pressure.

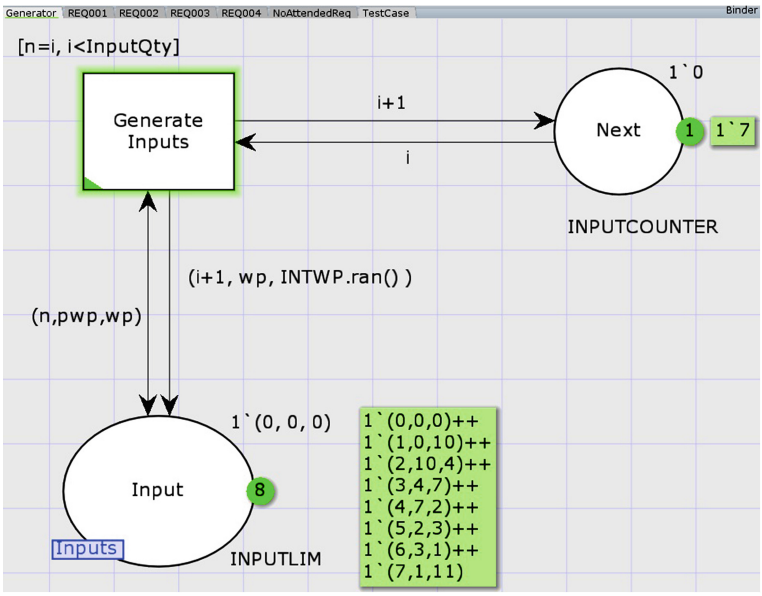

Fig. 7. Generating random entries

System Reaction. In the next step, for each input, the model verifies whether there is at least one enabled transition. If so, this transition is fired, and, as a result, it is obtained the expected system reaction. When firing a transition, it produces as output a tuple. This tuple comprises an ordering index, the previous and current values of the inputs and outputs, besides a label to track which requirements produced the output. The outputs are sorted with the support of the place *Next TO*. If for a given input there is no expected system reaction, the transition defined in the page *NoAttendedReq* is fired. It produces as output a new tuple keeping the output values the same. Considering the NPP example, Fig. 8 shows three test vectors generated and stored in *Test Oracles* via CPN simulation.

Test Vectors. Finally, in the last phase, the transition *Write File* records the test cases in a csv file. To accomplish this task, a *code segment* is used (shown

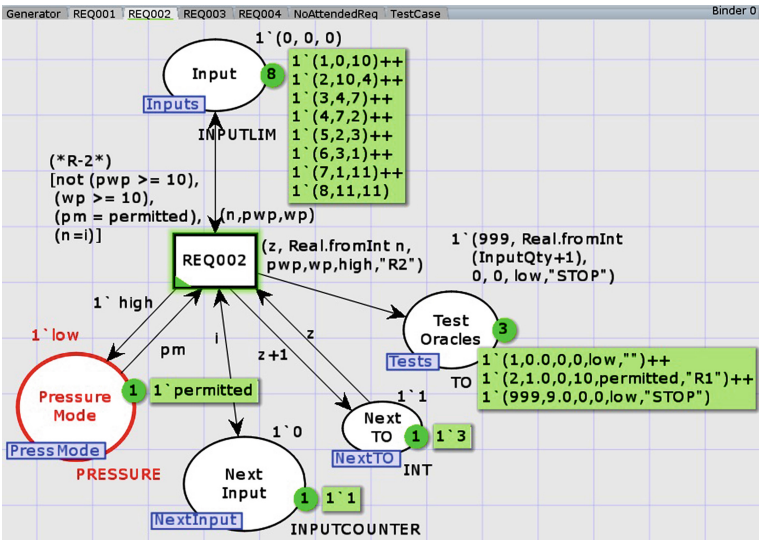

Fig. 8. Obtaining the expected system reaction

in Sect. 4.1), which is executed when this transition fires. The place *Next Output* is used to keep the ordering of the outputs.

In the csv file, the generated test vectors are shown in a tabular form (see Table 1). The first column ($TIME$) is just the ordering index previously explained. The next two columns lists the system input, as well as the corresponding expected output. The last column provides traceability information between the expected output with the system requirements – *no req.* refers to a test vector that was generated by firing the transition *noReq*.

Initially, when the water pressures changes from 0 to 10 (time 0.0 and 1.0, respectively), first, the pressure mode changes to permitted (according to the requirement REQ001), then, immediately after, it changes to high (according to the requirement REQ002). Afterwards, there is no transition associated with a requirement that can fire, and thus, the transition *noReq* fires (repeating the last vector).

Therefore, as one can see, the test generation approach proposed here behaves according to the following cyclic pattern: first, it performs as many transitions related to requirements as possible; when all of these transitions are not enabled, it performs the *noReq* transition; finally, it receives new inputs.

The process for generating test cases via simulation of CPN does not involve complex and intensive computations. Particularly, the reached system states are dynamically computed considering the randomly generated inputs. For instance, this approach contrasts with strategies that need to explore the system complete state space first and, thus, minimises state explosion problems. To generate more relevant test cases, the generation of inputs could be performed in a semi-random fashion in order to satisfy, for instance, user-defined coverage criteria. This improvement is a future work topic.

Table 1. Example of a test case for the simplified NPP

TIME	I: *the_water_pressure*	O: *the_pressure_mode*	REQ. Traceability
0.0	0	low	no req
1.0	10	permitted	REQ001
1.0	10	high	REQ002
1.0	10	high	no req.
2.0	4	permitted	REQ004
2.0	4	low	REQ003
2.0	4	low	no req.
3.0	7	low	no req.
4.0	2	low	no req.
5.0	3	low	no req.
6.0	1	low	no req.
7.0	11	permitted	REQ001
7.0	11	high	REQ002
7.0	11	high	no req.
8.0	11	high	no req

5 Conclusion

We presented in this work a variation of the NAT2TEST strategy that generates test cases from natural-language requirements via simulation of Colored Petri Nets (CPN). To accomplish this task, we define a systematic translation procedure that encodes the internal model of the NAT2TEST approach (*Data-Flow Reactive System* – DFRS) as a CPN. Afterwards, auxiliary components are added to the CPN to allow generation of test cases via simulation. Furthermore, this model is also suitable to analyses that might uncover problems in the original requirements, although this has not been the purpose of this paper.

There are several approaches to derive test cases from (semi-)formal models or directly from requirements expressed in some (controlled) natural language. Here we concentrate on approaches based on Petri nets.

A simple approach to generate test cases using CPN is proposed in [1]. The advantage of this approach is that the specification based on CPN can be validated by simulation (as in our case), but the state space analysis can lead to state explosion, unlike our approach, since the reached states are computed on demand. However, to perform analysis such as requirement completeness and consistency, it is likely that the complete state would need to be visited and, thus, our strategy would also fall into the state space problem.

An efficient approach for building a test suite using the PN-ioco conformance relation is proposed in [12]. Differently from our approach, it does not consider as input NL requirements. In [7], a method is proposed to convert UML 2.0 activity diagrams to a CPN model and apply the Random Walk Algorithm to create test

sequences, differently from our strategy that is based on NL requirements. On the other hand, it considers coverage criteria when generating test cases.

In [16], two techniques are proposed for concurrent systems. One uses CPN-Tree and the other uses Colored Petri Net Graph (CP-graph). CPN-Graph is considered as finite state machines (FSM) and existing test case generation methods based on FSM are applied. In CPN-Tree method the reachability trees reduced by the equivalent marking technique are used to achieve the practical test suite length. Again, the techniques are dependent on the data. Therefore, the major drawback of the existing approaches are possible state space explosion (non-scalability), which is minimised in our approach since we generate test cases via simulation.

As future work, we intend to: (1) apply our approach to more complex examples, (2) compare the obtained results with the other variations of the NAT2TEST strategy, and (3) enhance our CPN model to consider temporal properties of the requirements, besides hybrid systems.

Acknowledgments. We thank Embraer for the partnership related to the NAT2TEST framework and, particularly, Braulio Horta and Ricardo Filho for their valuable contribution. This work was supported by the National Institute of Science and Technology for Software Engineering (INES (www.ines.org.br)), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

References

1. Cai, L., Zhang, J., Liu, Z.: A CPN-based software testing approach. JSW **6**(3), 468–474 (2011)
2. Carvalho, A., Barros, F., Carvalho, A., Cavalcanti, A., Mota, A., Sampaio, A.: NAT2TEST tool: from natural language requirements to test cases based on CSP. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 283–290. Springer, Heidelberg (2015)
3. Carvalho, G., Barros, F., Lapschies, F., Schulze, U., Peleska, J.: Model-based testing from controlled natural language requirements. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2013. CCIS, vol. 419, pp. 19–35. Springer, Heidelberg (2014)
4. 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.) ICFEM 2014. LNCS, vol. 8829, pp. 43–58. Springer, Heidelberg (2014)
5. 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. Sci. Comput. Program. **95**, 275–297 (2014). Part 3(0)
6. Carvalho, G., Sampaio, A., Mota, A.: A CSP timed input-output relation and a strategy for mechanised conformance verification. In: Groves, L., Sun, J. (eds.) ICFEM 2013. LNCS, vol. 8144, pp. 148–164. Springer, Heidelberg (2013)
7. Farooq, U., Lam, C., Li, H.: Towards automated test sequence generation. In: 19th Australian Conference on Software Engineering, ASWEC 2008, pp. 441–450, March 2008
8. Jensen, K.: Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Springer, Berlin (1996)

9. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, Berlin (2009)
10. Jrgensen, J., Tjell, S., Fernandes, J.: Formal requirements modelling with executable use cases and coloured petri nets. Innovations Syst. Softw. Eng. **5**(1), 13–25 (2009)
11. Leonard, E., Heitmeyer, C.: Program synthesis from formal requirements specifications using APTS. Higher-Order Symbolic Comput. **16**(1–2), 63–92 (2003)
12. Liu, J., Ye, X., Li, J.: Colored Petri nets model based conformance test generation. In: 2011 IEEE Symposium on Computers and Communications (ISCC), pp. 967–970, June 2011
13. Milner, R., Harper, R., Tofte, M.: The Definition of Standard ML. MIT Press, Cambridge (1990)
14. Myers, G., Sandler, C., Badgett, T.: The Art of Software Testing. John Wiley, New York (2004)
15. Tjell, S.: Model-based testing of a reactive system with coloured petri nets. In: Hochberger, C., Liskowsky, R. (eds.) Informatik, LNI, vol. 94, pp. 274–281, GI (2006)
16. Watanabe, H., Kudoh, T.: Test suite generation methods for concurrent systems based on coloured Petri nets. In: Proceedings of 1995 Asia Pacific Software Engineering Conference, pp. 242–251, December 1995