Modelling and testing timed data-flow reactive systems in Coq from controlled natural-language requirements

Gustavo Carvalho[a], Igor Meira[a]

[a]*Universidade Federal de Pernambuco – Centro de Informática, 50740-560, Brazil*

Abstract

Data-flow reactive systems (DFRSs) are a class of embedded systems whose inputs and outputs are always available as signals. Input signals can be seen as data provided by sensors, whereas the output data are provided to system actuators. In previous works, verifying properties of DFRS models was accomplished in a programmatic way, with no formal guarantees, and test cases were generated by translating theses models into other notations. Here, we use Coq as a single framework to specify and verify DFRS models. Moreover, the specification of DFRSs in Coq is automatically derived from controlled natural-language requirements. Property verification is defined in both logical and functional terms. The latter allows for easier proof construction. Tests are generated with the support of the QuickChick tool. Considering examples from the literature, but also from the aerospace industry (Embraer), our testing strategy was evaluated in terms of performance and the ability to detect defects generated by mutation; within 8 seconds, we achieved an average mutation score of 75.80%.

Keywords: data-flow reactive systems, interactive theorem proving, Coq, property-based testing, QuickChick, controlled natural language

1. Introduction

Over the years, we have been building a society that is highly dependent on software. In many situations, the software is part of a safety-critical system, and all sorts of failures shall be minimised, since they might be life-threating or impose significant financial losses. Therefore, high trustworthiness levels are typically required for such systems.

Modelling and formal verification are strategies employed to increase system reliability. Creating models promotes a better comprehension and a more precise description of the expected behaviour. Formal verification brings certainty about properties being preserved. For instance, considering the avionics industry, in [1], three cases studies are reported illustrating the use of different classes of formal methods (theorem proving, model checking, and abstract interpretation) to meet reliability levels defined by the standard DO-178C (Software Considerations in Airbone Systems and Equipment Certification). When formal verification is not possible or feasible, testing becomes essential since it can unveil scenarios where implementations do not work as expected.

1.1. Model-based testing

In a model-based testing (MBT) strategy, test cases are derived from models, making the testing process more agile, less susceptible to errors, and less dependent on human interaction. This goal is usually reached by means of automatic generation (and execution) of test cases, besides automatic generation of test data, from specification models.

Here, we focus on models of data-flow reactive systems (DFRS): a class of embedded systems whose inputs and outputs are always available as signals. Additionally, the system behaviour might be time dependent. Models of DFRSs are fully explained in [2]. They have been used to model examples both from the literature and the industry. In this previous work, we also show that these models can be seen as timed input-output transition systems, but, being more abstract, enable automatic extraction from system-level specifications in a controlled natural language, which is an important aspect as discussed in what follows.

Despite the benefits of MBT, those who are not familiar with the models syntax and semantics may be reluctant to adopt theses formalisms. This is particularly more evident when considering formal MBT strategies, when formal models of the system expected behaviour need to be developed. Moreover, most of these models are not available in the very beginning of the project, when usually natural-language requirements are available. One possible alternative to overcome these limitations is to employ NLP techniques to derive the required models from natural-language specifications automatically.

1.2. Natural-language processing

The demand of stating the desired system behaviour using formal models may sometimes be an obstacle for adopting formal MBT techniques, despite all its benefits. The model notations may be not easy to interpret by, for instance, aerospace and automotive development engineers. Hence, a specialist (usually mathematicians, logicians, computer engineers and scientists) is required when such languages, and their corresponding techniques, are used in business contexts. Furthermore, most of these models are not yet available in the very beginning of the system development project.

As previously said, one possible alternative to overcome these limitations is to provide means for deriving specification models automatically from the already existing documentation, in particular, natural-language requirements. In this sense, NLP techniques can be helpful. If formal models are derived from natural-language requirements, besides applying MBT techniques, one can reason formally about properties of specifications that can be difficult to analyse by means of manual inspection, such as inconsistency and incompleteness.

Typically, there is a trade-off concerning the application of NLP in MBT. Some studies are able to analyse a broad range of sentences, whereas others rely on controlled versions of natural language (CNL). The works that adopt the former approach usually depend on a higher level of user intervention to derive models and to generate test cases. Differently, the restrictions imposed by a CNL might allow a more automatic approach when generating models and test cases. Ideally, a compromise between these two possibilities should be sought to provide a useful degree of automation along with a natural-language specification feasible to be used in practice.

In this work, seeking for automation, we adopt a CNL for describing the system requirements. In [2], we provide a comprehensive explanation of how models of DFRSs can be automatically derived from SysReq-CNL, a CNL specially designed for editing requirements of data-flow reactive systems.

1.3. The NAT2TEST$_{Coq}$ strategy

Considering a characterisation of DFRSs in the Coq proof assistant [3], we expand here the limits of the NAT2TEST strategy [4], which is devised to generate test cases from natural-language requirements. The specialisations of this strategy allow for exploring the benefits of different formal techniques, such as SMT solving, model and refinement checking, and simulation.

The specialisation proposed in this paper (hereafter, named NAT2TEST$_{Coq}$) is the first one based on an interactive proof assistant (Coq), which also brings new benefits and possibilities. For instance, in our previous efforts, verification of consistency properties of DFRS models was accomplished in a programmatic way, decoupled from the formal consistency definition. Here, we use the Coq proof assistant as a single framework to both specify and verify properties of DFRS models. Property verification is defined in both logical and functional terms. The latter allows for easier proof construction, besides contributing to the development of correct-by-construction tools (via extraction of Haskell/Ocaml code from the functional definitions), which is currently a future work.
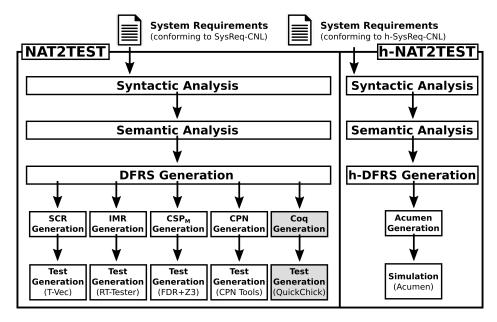
Figure 1: NAT2TEST – a strategy for generating test cases based on different formalisms

Furthermore, tests are generated via the QuickChick tool [5]. Considering examples from the literature, but also from the aerospace industry (Embraer[1]), our testing strategy was evaluated in terms of performance and the ability to detect defects generated by mutation; within 8 seconds, we achieved an average mutation score of 75.80%.

Therefore, the main contribution of this work is a single and consistent framework, based on Coq, for modelling, verifying and testing timed reactive systems from natural-language requirements; besides providing empirical evidence on the quality of our test generation strategy via mutant-based strength analysis. The remainder of this paper is organised as follows. Section 2 discusses the main concepts related to this work (the NAT2TEST strategy, data-flow reactive systems, interactive theorem proving, and property-based testing). Sections 3 and 4 present our logical and functional characterisation of symbolic and expanded data-flow reactive systems, respectively. Section 5 addresses test generation, based on the QuickChick tool. Our empirical analyses are presented in Section 6. Finally, Section 7 concludes this paper by discussing related and future work.

2. Background

Now, we present the foundational concepts related to this work: the NAT2TEST strategy (Section 2.1), DFRS models (Section 2.2), Coq (Section 2.3), and property-based testing (Section 2.4).

2.1. The NAT2TEST strategy

The NAT2TEST strategy is tailored to generate tests for timed data-flow reactive systems, considering different internal and hidden formalisms (see Figure 1). This 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 chosen.

---

[1]Embraer website: https://embraer.com/global/en

Syntactic analysis. In this work, requirements are written according to a CNL based on English: the SysReq-CNL, specially designed for editing requirements of data-flow reactive systems. The first phase of the NAT2TEST strategy is responsible for verifying whether the requirements are in accordance with the SysReq-CNL grammar. For each valid requirement, its corresponding syntax tree is identified.

As a running example, we consider a Vending Machine (VM) (adapted from the coffee machine presented in [6]). 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; the former is produced within 10 to 30 seconds, whereas the latter within 30 to 50 seconds. After producing coffee, the system returns to the idle state.

The following sentence exemplifies a requirement (REQ001) that adheres to the SysReq-CNL grammar: *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.*

Semantic analysis. In the second phase, the requirements are semantically analysed using the case grammar theory [7]. In this theory, a sentence is not analysed in terms of the syntactic categories or grammatical functions, but in terms of the semantic (thematic) roles played by each word/group of words in the sentence. Therefore, for each syntax tree, the group of words that correspond to a thematic role is identified. The collection of thematic roles for a requirement is called a requirement frame.

Table 1 shows the requirement frame for REQ001. We note that the thematic roles are grouped into conditions and actions. The roles that appear in actions are the following: *Action* – the action performed if the conditions are satisfied; *Agent* – entity who performs the action; *Patient* – entity who is affected by the action; and *To Value* – the patient value after action completion. Similar roles appear in conditions.

Table 1: Example of requirement frame for REQ001 (VM)

Condition #1 - Main Verb (Condition Action): *is*			
Condition Patient:	*the system mode*	Condition From Value:	–
Condition Modifier:	–	Condition To Value:	*idle*
Condition #2 - Main Verb (Condition Action): *changes*			
Condition Patient:	*the coin sensor*	Condition From Value:	–
Condition Modifier:	–	Condition To Value:	*true*
Action - Main Verb (Action): *reset*			
Agent:	*the coffee machine system*	To Value:	–
Patient:	*the request timer*		
Action – Main Verb (Action): *assign*			
Agent:	*the coffee machine system*	To Value:	*choice*
Patient:	*the system mode*		

DFRS generation. Afterwards, the third phase derives DFRS models – an intermediate formal characterisation of the system behaviour from which other formal notations can be derived. The possibility of exploring different formal notations allows analyses from several perspectives, using different languages, tools, and techniques. Besides that, it makes our strategy extensible. Models of DFRSs are explained in the following section. For a comprehensive explanation of how DFRS models are derived from requirement frames, we refer to [2].

Test generation. Test generation is achieved by translating DFRS models into internal and hidden formalisms. In what follows, we list the possibilities currently supported by the NAT2TEST strategy.

- NAT2TEST$_{SCR}$: based on *Software Cost Reduction* – SCR [8] (more details in [9])

- NAT2TEST$_{IMR}$: based on *Internal Model Representation* – IMR [10] (more details in [11])

- NAT2TEST$_{CSP}$: based on *Communicating Sequential Processes* – CSP [12] (more details in [13])

- NAT2TEST$_{CPN}$: based on *Coloured Petri Nets* – CPN [14] (more details in [15])

- NAT2TEST$_{Coq}$: based on *Coq* [3] – the main contribution of this paper

As said before, exploring different formal notations allows test generation from several perspectives, using different languages, tools, and techniques. The strategies NAT2TEST$_{SCR}$ and NAT2TEST$_{IMR}$ generate test cases with the support of commercial testing tools: T-VEC[2] and RT-Tester[3], respectively. Differently, the NAT2TEST$_{CSP}$ strategy reuses a general purpose refinement checker (FDR[4]) and SMT solver (Z3[5]) to deliver a formal and sound testing theory. Scalability is a known issue of this specialisation of the NAT2TEST strategy. Differently, the NAT2TEST$_{CPN}$ strategy aims at efficiency by generating test cases via random simulation of CPN models. Table 2 summarises the languages, tools and techniques considered by the aforementioned specialisations in order to generate test cases.

Table 2: Specialisations of the NAT2TEST strategy – techniques for generating test cases

	Internal formalism	Applied techniques	Tools integration
NAT2TEST$_{SCR}$	SCR	SMT solving	T-VEC
NAT2TEST$_{IMR}$	IMR	SMT solving	RT-Tester
NAT2TEST$_{CSP}$	CSP$_M$	Model checking + SMT solving	FDR + Z3
NAT2TEST$_{CPN}$	CPN	Simulation	CPN Tools
NAT2TEST$_{Coq}$	Coq	Property-based testing	Coq + QuickChick

The NAT2TEST$_{Coq}$ strategy distinguishes itself by integrating the NAT2TEST strategy with a technique not explored so far (proof assistants), and generating test cases via property-based testing. Now, besides test generation, it is also possible to develop proof scripts in Coq for proving relevant properties.

Other extensions. In [16], we extend our CNL to allow the specification of environment restrictions and, thus, how the system interacts with its surrounding environment. This extension has only been incorporated into the NAT2TEST$_{CSP}$ specialisation. In this way, unrealistic interactions between the system and the environment are not considered when generating test cases via FDR + Z3.

In [17], we allow the specification in natural language of system properties (in the style of temporal logic). These properties, along with the system requirements, are translated into CTL formulae and NuSMV models, respectively. With the aid of the NuSMV model checker [18], it is possible to assess whether the specified properties are satisfied by the NuSMV models. Here, test generation is not the ultimate goal, but model checking requirements.

Finally, in [19], we discuss a vertical adaptation of the NAT2TEST strategy in order to simulate hybrid systems (featuring the integration of discrete and continuous behavioural aspects) also from requirements adhering to a CNL. Therefore, in this work we revisit each phase of the NAT2TEST strategy, now considering the h-SysReq-CNL (an extension of the SysReq-CNL where it is possible to define differential equations) and a hybrid version of DFRS models (h-DFRS). Simulation is enabled by translating h-DFRS models into Acumen [20], which is a language and tool for the specification and simulation of hybrid systems.

---

[2]T-VEC website: https://www.t-vec.com/
[3]RT-Tester website: https://www.verified.de/products/rt-tester/
[4]FDR website: https://www.cs.ox.ac.uk/projects/fdr/
[5]Z3 website: https://github.com/Z3Prover/z3

2.2. Data-flow reactive systems

A data-flow reactive system (DFRS) is an embedded system whose inputs and outputs are always available, as signals. The input signals can be seen as data provided by sensors, whereas the outputs are data provided to actuators. A DFRS can also have internal timers, which are used to trigger time-based behaviour. There are two models of DFRSs: a symbolic (s-DFRS) and an expanded (e-DFRS) one. Briefly speaking, the former comprises an initial state, along with functions that describe the system behaviour (how the system state might evolve). Differently, an e-DFRS represents the system behaviour as a state-based machine; it can be seen as an expansion of its symbolic counterpart by applying the s-DFRS functions to its initial state, but also to the new reachable states.

As a running example, we consider the VM (presented previously). In this example, we have two input signals related to the coin sensor (*sensor*) and the coffee request button (*request*). A *true* value means that a coin was inserted or that the coffee request button was pressed, respectively. There are two output signals: one related to the system mode (*mode*) and another to the vending machine output (*output*). The values communicated by these signals reflect the system's possible modes (*choice* $\mapsto 0$, *idle* $\mapsto 1$, *strong coffee* $\mapsto 2$, and *weak coffee* $\mapsto 3$) and the possible outputs (*strong* $\mapsto 0$, and *weak* $\mapsto 1$). The VM has just one timer: the *request* timer, which is used to register the moments when a coin is inserted, when the coffee request button is pressed, and when the coffee is produced.

Figure 2 illustrates a scenario assuming continuous observation of the input and output signals. If we had chosen to observe the system discretely, we would have a similar scenario, but with a discrete number of samples over time.
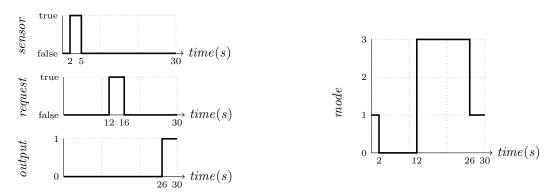

Figure 2: Example of signals for the vending machine

In this scenario, a coin is inserted 2s after starting the machine, and a coffee request is performed 10s later. The first input drives the system to the *choice* mode, whereas the second one to the *weak coffee* mode. A weak coffee is produced 14s after the request, which is reflected by changing the machine *output* signal.

An s-DFRS 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. The initial state is s_0, and F is a set of functions describing the system behaviour.

An e-DFRS differs from the symbolic one as it encodes the system behaviour as a state-based machine, whereas an s-DFRS does that symbolically via definitions of functions. An e-DFRS represents a timed system with continuous or discrete behaviour modelled as a state-based machine. States are obtained from an s-DFRS by applying its functions to non-stable states (when a system reaction is expected), but also letting the time evolve. Therefore, an e-DFRS is a 7-tuple: (I, O, T, $gcvar$, s_0, S, TR), where TR is a transition relation associating states in S by means of delay and function transitions. A delay transition represents the observation of the input signals' values after a given delay, whereas the function transition represents how the system reacts to the input signals: the observed values of the output signals. The transitions are encoded as assignments to input and output variables as well as timers.

Considering the example presented in Figure. 2, Figure. 3 shows some states of the e-DFRS representation for the vending machine. The initial state considers the initial value of all system variables. The delay

transition represents the change of the sensor signal from *false* to *true* after elapsing 2 seconds. Note that the value of *sensor* and the system global clock (*gc*) is updated in the state reached by the delay transition. At this moment, a system reaction is expected, which is characterised by a function transition, updating the system mode, besides resetting the request timer. Here, the reset operation is represented as assigning to the timer the current system global clock. The underlying reason is later explained. The function transition happens instantaneously (time does not evolve).
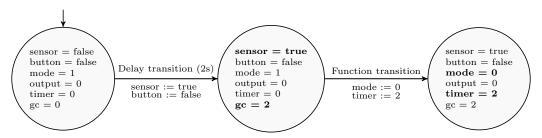

Figure 3: Some states of the e-DFRS representation for the vending machine

As said before, we refer to [2] for a comprehensive explanation of DFRS models, besides showing how they can be derived from natural-language requirements.

2.3. The Coq proof assistant

Opposed to automatic theorem provers, which aim to develop proofs in a full automatic manner, interactive theorem provers (also known as proof assistants) are tools that mix human interaction and some degree of automation when building proofs. Here, we present Coq [3], which is used later in this work.

Coq employs a functional language (Gallina), which is similar to Haskell, to describe algorithms. Computer-verified proofs are developed interactively using tactics, which have some limited support for automation via the tactics language (Ltac). As a logic system, Coq considers a higher-order logic. In what follows, we briefly address these three topics (Gallina, tactics, and automation). Moreover, we also explain the benefits and limitations of functional, logic, and inductive definitions in Coq.

2.3.1. The Gallina language

Gallina is a typical functional language, with support to define new types, besides polymorphic and higher-order functions. When defining a non-recursive function, one should use the keyword *Definition*. See the following example (*is_empty*) of a polymorphic function (valid for any given type T) that yields a logic value (*True* or *False*) indicating whether the given list is empty.

```
Definition is_empty {T : Type} (l : list T) : Prop :=
  match l with
  | [] ⇒ True
  | _ ⇒ False
  end.
```

Recursive functions shall use *Fixpoint*. The following example (*length*) yields the number of elements of a given list l. By pattern matching, if l is empty, the function yields 0. Otherwise, it yields the length of the list tail (*tl*) plus 1.

```
Fixpoint length {T : Type} (l : list T) : nat :=
  match l with
  | [] ⇒ 0
```

```
  | h :: tl ⇒ 1 + length tl
  end.
```

Differently from other functional languages, in Coq, all functions must terminate on all inputs. To ensure that, each recursion must structurally decrease some (the same) argument. If the decreasing analysis performed by the tool cannot identify such an argument, the corresponding recursive function is not defined. In the previous example (*length*), the decreasing argument is the list itself; each recursive call is performed on a smaller list.

2.3.2. Building proofs with tactics

In Coq, proofs are developed with the aid of tactics[6]. In the following example, we prove that the length of a list obtained by an append is equal to the sum of the lengths of the appended lists. Let *app* be the appending function, the theorem *length_app* formalises the previous statement.

```
Theorem length_app :
  ∀ (T : Type) (l1 l2 : list T), length (app l1 l2) = length l1 + length l2.
Proof.
  intros. induction l1.
   - simpl. reflexivity.
   - simpl. rewrite IHl1. reflexivity.
Qed.
```

The tactics employed modifies the proof goal in order to demonstrate its truth. Table 3 shows how the proof goal evolves after processing each tactic. The command `Proof.` starts the proof environment, loading the proof goal. After that, `intros.` performs universal instantiation, in order to prove the goal for arbitrary values of `T`, `l1`, and `l2`. The command `induction l1.` performs induction on *l1*. This creates two subgoals: base case, and inductive step. The symbol `-` (optional) tells Coq that we now focus on the next subgoal (base case).

After simplifying the proof goal of the base case (`simpl.`), we are left to prove that `length l2 = length l2`, which is trivially true (proved by `reflexivity.`). In the inductive step, assuming the hypothesis `IHl1 : length (app l1 l2) = length l1 + length l2`, we need to prove that `length (app (a :: l1) l2) = length (a :: l1) + length l2` holds. After simplification (`simpl.`), the goal becomes: `S (length (app l1 l2)) = S (length l1 + length l2)`.

By rewriting the goal considering *IHl1* (`rewrite IHl1.`) we get `S (length l1 + length l2) = S (length l1 + length l2)`, provable by `reflexivity.`. The command `Qed.` finishes the prove. Each command is verified by Coq, and it can only be applied if the underlying premises for its application are satisfied. This ensures the construction of a computer-verifiable proof.

2.3.3. Proof automation

Support for proof automation comes as tacticals (higher-order tactics – i.e., tactics that take other tactics as arguments), user-defined tactics, and some decision procedures. To illustrate some of these, consider the proof that a list is empty if, and only if, its length is 0.

---

[6]Index of built-in tactics: https://coq.inria.fr/refman/coq-tacindex.html

Table 3: Proof of theorem *length_app*

Command	Proof state
`Proof.`	`forall (T : Type) (l1 l2 :  list T),` `length (app l1 l2) = length l1 + length l2`
`intros.`	`length (app l1 l2) = length l1 + length l2`

`induction l1.`	_____(1/2)
	`length (app [] l2) = length [] + length l2`
	_____(2/2)
	`length (app (a ::  l1) l2) = length (a ::  l1) +`
	`length l2`

`-`	`length (app [] l2) = length [] + length l2`
`simpl.`	`length l2 = length l2`
`reflexivity.`	`This subproof is complete`

`-`	`IHl1 :   length (app l1 l2) = length l1 + length l2`
	————————————————————————————
	`length (app (a ::  l1) l2) = length (a ::  l1) +`
	`length l2`

`simpl.`	`IHl1 :   length (app l1 l2) = length l1 + length l2`
	————————————————————————————
	`S (length (app l1 l2)) = S (length l1 + length l2)`

`rewrite IHl1.`	`IHl1 :   length (app l1 l2) = length l1 + length l2`
	————————————————————————————
	`S (length l1 + length l2) = S (length l1 + length`
	`l2)`

`reflexivity.`	`No more subgoals.`
`Qed.`	`length_app is defined`

```
Theorem empty_length_0 :
  ∀ (T : Type) (l : list T),
    is_empty l ↔ length l = 0.
Proof.
  intros. destruct l.
  - split.
    + simpl. intro H. reflexivity.
    + simpl. intro H. reflexivity.
  - split.
    + simpl. intro H. inversion H.
    + simpl. intro H. inversion H.
Qed.
```

```
Ltac trivial_hypo :=
    try (simpl ; intro H ; reflexivity).

Ltac absurd_hypo :=
    try (simpl ; intro H ; inversion H).

Theorem empty_length_0' :
  ∀ (T : Type) (l : list T),
    is_empty l ↔ length l = 0.
Proof.
  intros. destruct l ; split ;
  repeat (trivial_hypo ; absurd_hypo).
Qed.
```

In our first try (*empty_length_0*), we perform a case analysis on `l` (i.e., empty and non-empty list). Since the goal involves an equivalence (↔), we use the tactic `split.` to generate two subgoals considering both sides of the implication. The first two cases (when the list is empty) can be trivially proved by reflexivity. The two others (when the list is not empty) are proved by contradiction (`inversion H`), since we have a contradictory hypothesis `H` in the proof context: a non-empty list is empty, or the length of a non-empty list is 0. As one can see, there is some degree of repetition, and thus room for automation, in this proof.

In our second try (*empty_length_0'*), we define two tactics (*trivial_hypo*, and *absurd_hypo*), which tries to prove the current goal by reflexivity or contradiction, respectively. Then, the theorem is proved by trying to apply these two tactics many times (`repeat`). The command `;` applies the following commands to all

subgoals, and not only to the next one.

2.3.4. Functional, logical, and inductive characterisations

Another important aspect of Coq to this work is the possibility to define aspects functionally, logically, or inductively. To give a concrete example, consider the following definitions of whether a number is even. The first definition (*evenb*) is a function that yields true (boolean value) if n is even, false otherwise. In this definition, S denotes the successor of a natural number. If n is the successor of the successor of some number n', to assess whether n is even it suffices to assess whether n' is even.

```
Fixpoint evenb (n : nat) : bool :=        Definition is_even (n : nat) : Prop :=
  match n with                              ∃ k, n = k + k.
  | 0 ⇒ true
  | 1 ⇒ false                             Inductive even : nat → Prop :=
  | S (S n') ⇒ evenb n'                     | ev_0 : even 0
  end.                                      | ev_SS : ∀ n, even n → even (S (S n)).
```

The second definition (*is_even*) defines this concept in logical terms: a natural number n is even if, and only if, there is a natural number k, such that $n = k + k$. The third, and last, definition (*even*) characterises this concept inductively. The definitions *ev_0* and *ev_SS* can be seen as inference rules, stating that 0 is even (*ev_0*), and that if n is even, the successor of its successor is also even (*ev_SS*).

Although these three definitions properly capture the concept of being even, proving facts using these definitions might differ significantly. For the last two definitions (logical and inductive ones), one will need to use specific tactics to deal with the existential quantifier and the inference rules, respectively. Differently, concerning the functional definition, one can use the tactic `simpl.` to simplify the proof goal by evaluating the function *evenb* for the given arguments. However, in some situations, due to the termination requirement of Coq for functions, one cannot rely on a purely functional definition.

In this work, when dealing with concrete examples of DFRSs, we favor their functional characterisation to enable automatic proof of model consistency.

2.4. Property-based testing

Testing is an extremely important task for software development, also complimentary to proofs. Even in the presence of proved components, we typically need to integrate them to unproved ones, and thus we need to rely on testing to analyse integration. Additionally, testing can be used as a quick tool to evaluate properties, before trying to prove them. If we submit a property to a large and relevant number of test cases, and it does not fail, we get confidence on its correctness. If it fails, we save proof effort on trying to prove `False`.

Property-based testing, famous in the functional world due to the QuickCheck framework for Haskell [21], consists of random generation of input data in order to test a computable (executable) property. It comprises four ingredients: (i) an executable property P, (ii) generators of random input values for P, (iii), printers for reporting counterexamples, and (iv) shrinkers to minimise counterexamples.

A simple example shown in the QuickCheck manual[7] describes how to test whether the reverse of the reverse of a list is equal to the original list. First, one needs to define this property in Haskell:

prop_RevRev xs = reverse (reverse xs) == xs
where types = xs::[Int]

Then, QuickCheck is called to try to falsify the property. In this case, no counterexample is found, which is expected, since the property is actually true. However, if testing whether the reverse of a list is equal to the original list, a counterexample should be easily found, and presented to the user.

The concept of property-based testing is supported in Coq via the QuickChick tool, which is an adaptation of QuickCheck ideas for the Coq proof assistant. In this work, we use the QuickChick tool for generating test cases for DFRSs.

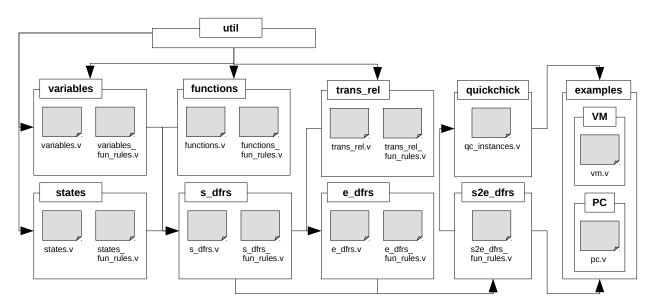[7]QuickCheck: http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html

Figure 4: Characterisation of DFRSs in Coq

3. Characterisation of symbolic DFRSs in Coq

For a comprehensive explanation of DFRSs (using Z), a discussion of their expressiveness, along with theoretical and practical validations, besides an explanation of how such models can be automatically derived from controlled natural-language requirements, we refer to [2]. Here, we highlight some aspects of our Coq representation, which main benefit is to have a single computer-verifiable framework for specification, verification and testing. It also enables the extraction of functional code (Haskell/Ocaml) directly from the verified and formal functional definitions; contributing to the development of correct-by-construction tools.

The general architecture of our characterisation of DFRSs in Coq[8] is presented in Figure 4. The folders *variables*, *states*, and *functions* define the constituent elements of a symbolic DFRS (*s_dfrs*). The folder *e_dfrs* contains the definitions of an expanded DFRS, which is built upon the symbolic one and the definition of a transition relation (*trans_rel*). Each folder has a main *.v file (named after the folder's name), which provides a logical characterisation. There are two other files: *_fun_rules.v (a functional characterisation), and *_fun_ind_equiv.v (proving that both characterisations are equivalent – not shown in Figure 4 to save space). These proofs allow us to create instances of DFRSs, proving automatically that they are consistent. This aspect is further explained later.

In *s2e_dfrs*, we have functions that allow for a dynamic expansion of an e_DFRS from a given s_DFRS. Part of these functions are used by our test generation module (defined in *quickchick*). Finally, the examples considered in this paper are defined in *examples*. One might rely on the *quickchick* module to generate test cases or on the *s2e_dfrs* module to perform bounded expansion of an *e_DFRS*. In what follows, we detail our Coq characterisation of DFRSs.

3.1. Logical characterisation of symbolic DFRSs

As said before, an s_DFRS is a 6-tuple: (I, O, T, *gcvar*, s_0, F), comprising (input, output, and timer) variables, besides a global clock, along with an initial state, and a set of functions describing the system behaviour.

---

[8]Available online in: https://github.com/igormeira/DFRScoq

3.1.1. Variables

System variables ($SVARS$), which might represent input or output variables, are defined as a list of pairs of names and types ($TYPE$). A variable name ($VNAME$) is basically a string ($NAME$ is defined as a string) that cannot be the reserved name of the system global clock, which is "gc".

```
Definition ind_rules_svars (svars :
list (VNAME × TYPE)) : Prop :=
    is_function (map (fun p ⇒ fst p)
                 svars) comp_vname
  ∧ ¬ (length svars = 0)
  ∧ ind_svars_valid_type svars.
```

```
Record VNAME : Set := mkVNAME {
    vname : NAME
  ; rules_vname : ind_rules_vname vname
}.
Record SVARS : Set := mkSVARS {
    svars : list (VNAME × TYPE)
  ; rules_svars : ind_rules_svars svars
}.
```

It is important to note the usage of `Record`. In Coq, differently from typical programming languages, a record might comprise data values (*vnames* and *svars*), but also properties (predicates) that need to hold (*rules_vname* and *rules_svars*). Therefore, when creating an instance of a record, it is necessary to prove that the corresponding properties hold. This feature brings cohesion between structural definition and consistency properties.

For the $VNAME$ record, the property *rules_vname* enforces that the variable name (*vname*) is different from "gc". Concerning $SVARS$, we have that *svars* needs to be a non-empty function of variable names to types (no name repetition, and each name is mapped to a single type). Besides that, we need to have type consistency. These properties are defined with the aid of propositional functions (i.e., functions that build predicates from the given arguments): *ind_rules_vname* (not shown here), and *ind_rules_svars*. A record instance is created as follows.

```
Definition the_coin_sensor : VNAME.
Proof.
    apply (mkVNAME "the_coin_sensor").
    unfold ind_rules_vname, gc, not. intro H. inversion H.
Defined.
```

First, we give a name for the instance (*the_coin_sensor*), and then Coq enters on proof mode. After providing the value for the instance ("the_coin_sensor") via the command `apply (mkVNAME ...).`, we need to prove that the defined properties hold for the given value; in this case, that "the_coin_sensor" is not equal to "gc". By unfolding the definitions of *ind_rules_vname*, *gc* and *not* we are left to prove that `string_dec` "*the_coin_sensor*" "*gc*" → `False`. In Coq, the logical negation ($\neg P$) is modelled as $P \to$ `False`. Then, we move the antecedent of the implication to the proof context (`intro H.`), and finish the proof since H is a contradiction (`inversion H.`). This feature (records with values and predicates) prevents us from creating inconsistent instances (violating rules).

It is worth noting that the proof is finished with `Defined.` instead of `Qed.`. This makes the definition transparent, and it can be unfolded later (we will be able to retrieve the string associated with *vname*). When a proof is finished with `Qed.`, it is marked as opaque (proof irrelevance). To model system timers, we define a different record $STIMERS$. DFRS variables are defined as follows:

```
Record DFRS_VARIABLES : Set := mkDFRS_VARIABLES {
    I : SVARS ; O : SVARS ; T : STIMERS ; gcvar : NAME × TYPE
  ; rules_dfrs_variables : ind_rules_dfrs_variables I O T gcvar
}.
```

The definition of *ind_rules_dfrs_variables* guarantees that: (i) the name of the *gc* variable is the string

"gc", (ii) I, O, and T are disjoint (different names), and (iv) we have type consistency between timers and the global clock (they share the same type). For the vending machine, we have the following definitions.

```
Definition vm_I : SVARS. Proof.
  apply (mkSVARS [(the_coin_sensor, Tbool) ; (the_coffee_request_button, Tbool)]).
  (* proof omitted *) Defined.
Definition vm_variables : DFRS_VARIABLES. Proof.
  apply (mkDFRS_VARIABLES vm_I vm_O vm_T vm_gcvar). (* proof omitted *)
Defined.
```

The element *vm_I* defines the input variables; *vm_O*, *vm_T*, and *vm_gcvar* are analogous. Note that the element *the_coin_sensor* was defined in a previous example.

3.1.2. Initial state

A state is a list of names mapped to a pair of values. See the following definitions.

```
Record STATE : Set := mkSTATE {
  state : list (NAME × (VALUE × VALUE))
  ; rules_state : ind_rules_state state }.
Record DFRS_INITIAL_STATE : Set :=
  mkDFRS_INITIAL_STATE { s0 : STATE }.
```

The pair elements are the previous/current variable values. The property *ind_rules_state* enforces that *state* is also a function. The initial state for the VM is: both input signals are false, the system mode is idle (*i 1*), the machine output is strong coffee (*i 0*), and the timer and the global clock are equal to 0.

```
Definition vm_state : STATE.
Proof. apply (mkSTATE
                [("the_coin_sensor", (b false, b false));
                 ("the_coffee_request_button", (b false, b false));
                 ("the_system_mode", (i 1, i 1));
                 ("the_coffee_machine_output", (i 0, i 0));
                 ("the_request_timer", (n 0, n 0)); ("gc", (n 0, n 0))]).
  (* proof omitted *)
Defined.
```

3.1.3. Functions

A DFRS might comprise multiple concurrent components. The behaviour of each component is described by a function. The behaviour of the entire s_DFRS is then defined as a list of functions (F), which cannot be empty (ensured by *ind_rules_dfrs_functions*). Each function (*function*) is a list of 4-tuples: a static guard, a timed guard, a list of assignments, and requirement traceability information. The first two elements define the static and timed conditions necessary to be met to react by performing the respective assignments. One of these two conditions can be empty, but not both (ensured by *ind_rules_function*).

```
Record FUNCTION : Set := mkFUNCTION {
  function : list (EXP × EXP × ASGMTS × REQUIREMENT) ;
  rules_function : ind_rules_function function }.
Record DFRS_FUNCTIONS : Set := mkDFRS_FUNCTIONS {
  F : list FUNCTION ; rules_dfrs_functions : ind_rules_dfrs_functions F }.
```

The aforementioned requirement of the VM (REQ001) says that "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". Part of the formalisation of this requirement is shown below: *req001_sg_disj3* models one clause of its static guard (the current system mode is equal to idle—1), whereas *req001_asgmt2* models one of its assignments (updating the system mode to choice—0). The term *DIS* refers to a list of disjunctive clauses; in this example, we have a conjunction of two elements, each one with a single disjunction. In our work, the conditions adhere to a Conjunctive Normal Form (CNF).

```
Definition req001_sg_disj3 : DISJ.
Proof. apply (mkDISJ [mkBEXP (current (the_system_mode)) eq (i 1)]).
  (* proof omitted *) Defined.

Definition req001_asgmt2 : ASGMT.
Proof. apply (mkASGMT (the_system_mode, (i 0))). Defined.
```

3.1.4. s_DFRSs

An s_DFRS is composed by the previously defined elements. Various consistency properties are enforce by *ind_rules_s_dfrs*; for instance, the initial state must provide values for all system variables. We refer to our git repository for the definition in details of *ind_rules_s_dfrs*.

```
Record s_DFRS : Set := mkS_DFRS {
   s_dfrs_variables : DFRS_VARIABLES ;
   s_dfrs_initial_state : DFRS_INITIAL_STATE ;
   s_dfrs_functions : DFRS_FUNCTIONS
   ; rules_s_dfrs : ind_rules_s_dfrs s_dfrs_variables s_dfrs_initial_state s_dfrs_functions
}.
```

3.2. Functional characterisation of symbolic/expanded DFRSs

When defining the element *the_coin_sensor*, it was necessary to prove that the variable name is not "gc". When more complex consistency rules need to be proved, the proof script becomes equally more complex, which inhibits automation. A workaround consists in providing functionally-defined consistency rules, which are logically equivalent to their logical/inductive counterparts.

The equivalence proof for *ind_rules_vname* (a variable name shall be different from "gc") is shown below. We prove both sides of the implication (`split.`) by reaching (via different ways) a contradiction in the proof (`inversion H'`).

```
Theorem theo_rules_vname :
   ∀ (vname : NAME), ind_rules_vname vname ↔ fun_rules_vname vname = true.
Proof.
   intros. unfold fun_rules_vname, ind_rules_vname. split.
   - intro H. unfold not in H. apply eq_true_not_negb. unfold not. intro H'.
     rewrite theo_string_dec in H. apply H in H'. inversion H'.
   - intro H. unfold not. intro H'. rewrite negb_true_iff in H.
     rewrite theo_string_dec in H'. rewrite H in H'. inversion H'.
Qed.
```

Now it is possible to define *the_coin_sensor* as follows. We apply the theorem *theo_rules_vname* to rewrite the proof goal considering the functional characterisation. Then, it suffices to execute the tactic `reflexivity.`, which simplifies the goal by performing the necessary computations, besides concluding the

proof.

```
Definition the_coin_sensor : VNAME.
Proof.
   apply (mkVNAME "the_coin_sensor"). apply theo_rules_vname. reflexivity.
Defined.
```

Actually, all proofs omitted in the previous examples follow this pattern. Nevertheless, the logical/inductive characterisation is still useful, mainly when dealing with infinite aspects, which appear on expanded DFRSs. The functional counterpart needs to be bounded to some exploration limit.

4. Characterisation of expanded DFRSs in Coq

An e_DFRS is defined in terms of a transition relation (a list of transitions – *TRANS*). Each transition relates two states by means of a label (*TRANS_LABEL*). A label denotes a function (*func*) or a delay (*del*) transition. A function transition models system reaction; it changes the value of output variables and timers. A delay transition models time evolving (*DELAY*), besides modifying the value of system inputs. A number of consistency rules are enforced by *rules_TR*.

```
Inductive TRANS_LABEL : Type :=
 | func : (ASGMTS × REQUIREMENT)
        → TRANS_LABEL
 | del : (DELAY × ASGMTS)
        → TRANS_LABEL.
Record TRANS : Set := mkTRANS {
  STS : STATE × TRANS_LABEL
       × STATE }.
```

```
Record TRANSREL : Set :=
 mkTRANSREL {
   transrel : list TRANS }.
Record
 DFRS_TRANSITION_RELATION
 := mkDFRSTRANSITIONREL {
   TR : TRANSREL ;
   rules_TR : ind_rules_TR TR ; }.
```

An e_DFRS is defined as a combination of variables, states, and a transition relation. As said before, an e_DFRS is obtained by the expansion of the corresponding s_DFRS, by letting the time evolve (performing delay transitions), and observing how the system reacts to input stimuli (performing function transitions).

```
Record e_DFRS : Set := mkE_DFRS {
   e_dfrs_variables : DFRS_VARIABLES;
   e_dfrs_states : DFRS_STATES;
   e_dfrs_transition_relation : DFRS_TRANSITION_RELATION;
}.
```

Since time is always expected to be able to evolve, and the system global clock is part of the state, an e_DFRS comprises an infinite number of states. Therefore, a function that expands a symbolic DFRS, yielding the obtained e_DFRS, would never terminate its execution and, thus, cannot be defined in Coq. In the following section, we explain how we can perform a bounded construction of an e_DFRS.

4.1. Bounded construction of expanded DFRSs

Typically, an e_DFRS consists of an infinite number of states, since time might always evolve, reaching a new state (different value for the global clock). Therefore, although one can characterise all states of an e_DFRS inductively, a functional definition needs to be bounded. Here, we restrict the number of recursive calls of *buildTR*, which expands dynamically an s_DFRS, up to *num*. When it reaches 0, the function stops. When there are still states to visit (expand), and the limit has not been reached, new states are generated with the aid of the auxiliary function *genTransitions*. Then, the function recurses decreasing the limit by

one.

```
Fixpoint buildTR (toVisit visited : list STATE) (I Out T : list (VNAME × TYPE))
  (F : list (list FUNCTION)) (possibilities : list (VNAME × list VALUE))
  (num : nat) : list TRANS :=
  match toVisit, num with
  | [] , _ ⇒ []
  | _ :: _, 0 ⇒ []
  | h :: t, S n' ⇒ let tr1 := genTransitions h I Out T F possibilities
                in if in_state_list h.(variables) visited beq_state
                  then buildTR t visited I Out T F possibilities n'
                  else tr1.(transrel) ++
                      buildTR (t ++ (get_list_states tr1.(transrel) (h :: visited)))
                              (h :: visited) I Out T F possibilities n'
  end.
```

The function *genTransitions* first assesses whether a given state is stable (no system reaction is expected, which means that no static and timed guards evaluate to true). If it is stable, the function generates delay transitions (with the configured time step) considering all possible combinations of input values. Otherwise, the function generates function transitions considering the assignments associated with the static and timed guards that evaluate to true.

```
Definition genTransitions (s : STATE) (I O T : list (VNAME × TYPE))
  (F : list (list FUNCTION)) (possibilities : list (VNAME × list VALUE))
  : TRANSREL :=
  let entries := union_lists (map (fun f : FUNCTION ⇒ f.(function)) (union_lists F)) in
  let combinations := gen_asgmts_combination (possible_asgmts possibilities) [[]] in
  if is_stable s (List.app I O) T F
    then mkTRANSREL (make_trans_del s I T combinations)
    else mkTRANSREL (make_trans_func s (I ++ O) T entries).
```

This bounded and functional exploration of e_DFRSs can support the development of simulators for e_DFRSs. Recall that it is possible to extract Haskell and Ocaml code directly from functional definitions in Gallina. Simulation is an important validation technique, since it enables the analysis of whether the created model properly captures the system's expected behaviour.

5. Generating test cases with QuickChick

Besides bounded exploration, we also allow for the generation of test data (via QuickChick) by sampling valid traces. To achieve this goal, we define a function (*genValidTrace*) that, given an s_DFRS, yields random valid traces (a list of transitions). To generate valid sequences of transitions, this function relies upon *genTransitions*, previously defined (see Section 4.1). From the initial state, it generates the possible transitions. Then, it randomly chooses one possible transition, and calls *genValidTrace* recursively, considering as the current state the one reached by the selected transition. The generation of a trace stops when *size* reaches 0 (similarly to *num* in *buildTR* – see Section 4.1), but also when *num* has not reached 0 yet. However, this last situation happens with a lower probability. This is achieved via the operator *freq*.

```
Fixpoint genValidTrace (st : STATE) (dfrs : s_DFRS)
(possibilities : list (VNAME × list VALUE)) (size : nat) : G trace :=
  match size with
```

$| \ 0 \Rightarrow ret \ []$
$| \ S \ size' \Rightarrow \texttt{let}$
$\quad tr := (genTransitions \ st \ dfrs.(s\_dfrs\_variables).(I).(svars)$
$\qquad\qquad dfrs.(s\_dfrs\_variables).(O).(svars) \ dfrs.(s\_dfrs\_variables).(T).(stimers)$
$\qquad\qquad [dfrs.(s\_dfrs\_functions).(F)] \ possibilities).(transrel)$
$\quad \texttt{in} \ freq \ [\ (1, \ ret \ []) \ ;$
$\qquad\qquad (size, \ x \leftarrow next\_label \ st \ tr \ ;;$
$\qquad\qquad\qquad xs \leftarrow genValidTrace \ (nextState \ st \ x \ tr) \ dfrs \ possibilities \ size' \ ;; \ ret \ (x :: xs)) \ ]$

When we sample *genValidTrace* with QuickChick, by default, it performs 11 calls to the function. See the QuickChick documentation[9] for more information. Table 4 shows, in a tabular form, a fragment of an output (test data) generated via QuickChick. The labels *time*, *sensor*, *request*, *mode*, and *output* refer to the system global clock, the coin sensor, the coffee request button, the system mode, and the coffee machine output, respectively. For this example, the configured time step was 1 and, thus, we see the system global clock evolving by 1 time unit per test step.

Table 4: Example of test data generated via QuickChick (VM)

time	sensor	request	mode	output
0	false	false	1	0
1	false	false	1	0
2	true	true	0	0
3	true	true	0	0
4	false	false	0	0

In this example, a coin is inserted and the coffee request button is pressed both at the time (2 seconds after the test beginning). As expected, the system mode changes to choice (represented as 0). When the system global clock is 4, 2 seconds later, the coin sensor becomes false again, and the coffee request button is released. Although not shown in this tabular representation, as we keep requirement traceability information, when defining functions (see Section 3.1 – Functions), we can also extract requirement coverage information from the generated test data.

5.1. Considerations on soundness and tool certification

In order to develop a sound model-based testing theory, typically, it is necessary to consider the following elements: (i) adopt a formal specification language, (ii) assume that it is possible to represent the implementation behaviour using the same language (testability hypothesis), (iii) define an implementation relation expressing correctness of implementations with respect to specification models, (iv) define a test generation and a test execution procedure, and (v) finally prove that these procedures are sound with respect to the implementation relation (i.e., if the execution of a generated test case fails, it means that the implementation under test is not related to the considered specification model by the adopted implementation relation).

In this paper, we use Gallina as a formal specification language, and we use property-based testing tool (QuickChick) to generate test data (input and expected output data). To develop a sound testing theory, it would be necessary to define the missing elements, namely: an implementation relation, a test execution procedure, besides taking into account the details on how QuickChick performs property-based testing. Then, we would have the necessary ingredients to prove the soundness of this Gallina-based testing theory. Regarding the implementation relation, a possible candidate would be a relation similar to the one defined in [13]: csptio, a conformance relation for CSP timed input-output conformance relation, which also addressed data-flow reactive systems.

---

[9]QuickChick manual: softwarefoundations.cis.upenn.edu/qc-current/

Finally, in some critical domains, tool certification is mandatory prior to its integration into the development process; unless the tool outputs are manually inspected, and evidence is produced in favor of the correctness of them. In our work, we do not expect the integration of the NAT2TEST tool into a development tool chain without manual inspection. Our goal is to aid the verification team by producing test data, but it remains as the team responsibility the analyses of whether the system has been properly tested.

6. Empirical analyses

We evaluate our work by considering the VM and an example from the aerospace domain (provided by Embraer): a priority command function (PC) that decides whether the pilot or copilot side stick will have priority in controlling the airplane. Our evaluation considers performance and quality aspects. All data presented here consider multiple executions. The main threat to validity is external. We cannot generalise the conclusions, since few and not very complex examples were considered. Nevertheless, the results allow interesting insights and provide feasibility evidence for our Coq characterisation and testing strategy.

We generated (multiple times) three datasets: performing 1, 5, and 10 calls to the QuickChick sampling function. Each call performs 11 calls to *genValidTrace*. Therefore, each dataset contained 11, 55, and 110 test cases, respectively. The size of each test is bounded to $size = 100$ (see Section 5) – up to 100 delay/function transitions. The time to generate each dataset is small: VM (1.29s/0.01s; 3.75s/0.27s; 7.03s/0.63s) and PC (1.37s/0.03s; 3.96/0.04s; 7.39s/0.80), μ/σ for each dataset[10]. The time is linearly proportional to the number of sampling calls.

A mutant-based strength analysis was used to assess the quality of the generated tests. Mutation operators yield a trustworthy comparison of test cases strength because they create erroneous programs in a controlled and systematic way [22]. A good test suite should be able to detect the introduced error (kill the mutant). Sometimes, the alive mutant is equivalent to the correct program. In general, this verification is undecidable and too error-prone to be made manually.

We follow a conservative approach: all alive mutants are not equivalent ones. This assumption makes the results of the empirical analyses the worst case. We considered C reference implementations, which were mutated via SRCIROR [23]: 255/229 mutants were generated for the VM/PC, respectively. We wrote C test drivers to run all mutants against all generated datasets. Figure 5 shows the mutation score (ratio of killed/generated mutants) for the VM and PC.

In average, the mutation score was 75.80% (VM: $\mu = 74.56\%/\sigma = 11.00\%$; PC: $\mu = 77.03\%/\sigma = 2.81\%$). For the VM, an outlier dataset killed all but one mutant. We inspected the alive one, and it is equivalent to the original code. For the PC, it is worth noting that 5 and 10 calls to the sampling function yielded very similar results. We believe these results are promising, considering the observed performance, besides being fully automatic[11]. Higher scores should be pursued by complementing the dataset with specialist-defined test scenarios.

7. Conclusion

This paper presents a logical and a functional characterisation of timed reactive systems in Coq, which can be automatically derived from natural-language requirements. This allows for a single and consistent framework for specifying, verifying, and testing such systems. Moreover, it contributes to the development of correct-by-construction tools, since it is possible to extract Haskell or Ocaml code from the verified and formal functional definitions. Furthermore, tests are automatically generated with the QuickChick tool. Empirical analyses, considering examples both from the literature and the industry, showed that our testing strategy is fast, and can detect about 75.80% defects introduced by mutation.

---

[10]Considering an i7 @ 2.40GHz x 4, with 8 GB of RAM running Ubuntu 16.04 LTS.

[11]All empirical data and scripts for the VM are available online: see Footnote 8. The files regarding the PC example cannot be made publicly available.
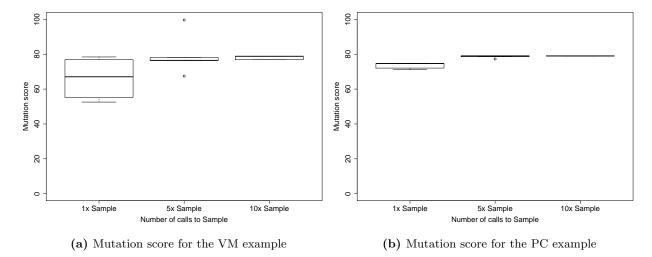
(a) Mutation score for the VM example

(b) Mutation score for the PC example

Figure 5: Mutant-based strength analysis

The integration of (interactive) theorem provers and testing strategies is a relevant research topic, and thus has been developed by many researches. Similarly to Coq, other provers provide integration between proofs and tests. For instance, considering the theorem prover Isabelle/HOL, we refer to [24, 25, 26]. In general, similar functionalities are provided among these different options. Modelling and/or testing timed systems using (interactive) theorem provers is addressed, for example, in [27, 28, 29, 30], considering timed connectors, real-time operating systems, programmable logic controllers, and Java code, respectively. Differently, our work focus on models of system-level requirements. In this direction, we have the works reported in [31, 32], which consider as modelling notation timed automata and Circus, respectively. However, these models are not derived from natural-language requirements, which is the case here.

7.1. Future work

All proofs regarding the equivalence of logical and functional characterisations of DFRS models have been completed. The generation of Coq specifications and test generation via QuickChick have been integrated into the NAT2TEST tool. Furthermore, more empirical analyses have been carried out. These new results will be fully presented and explained in the final version of this paper.

Besides these actions, which have already been carried out, we also plan to support specialist-defined test scenarios. One will be able to define fragments of a test scenario, and, with the aid of QuickChick, we will find a valid trace that includes the user-defined scenario. This will allow for complementing the testing campaign with scenarios that need to be tested, which might not be necessarily covered by a random test-generation strategy.

References

[1] D. Cofer, S. Miller, DO-333 Certification Case Studies, in: J. M. Badger, K. Y. Rozier (Eds.), NASA Formal Methods, Springer International Publishing, Cham, 2014, pp. 1–15.

[2] G. Carvalho, A. Cavalcanti, A. Sampaio, Modelling timed reactive systems from natural-language requirements, Formal Aspects of Computing 28 (5) (2016) 725–765. doi:10.1007/s00165-016-0387-x.

[3] Y. Bertot, P. Castran, Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, 1st Edition, Springer Publishing Company, Incorporated, 2010.

[4] G. Carvalho, F. Barros, A. Carvalho, A. Cavalcanti, A. Mota, A. Sampaio, NAT2TEST Tool: From Natural Language Requirements to Test Cases Based on CSP, in: R. Calinescu, B. Rumpe (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2015, pp. 283–290.

[5] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, B. C. Pierce, Foundational property-based testing, in: C. Urban, X. Zhang (Eds.), Interactive Theorem Proving, Springer International Publishing, Cham, 2015, pp. 325–343.

[6] K. G. Larsen, M. Mikucionis, B. Nielsen, Online Testing of Real-time Systems Using Uppaal, Springer Berlin Heidelberg, 2005, pp. 79–94.

[7] C. J. Fillmore, The Case for Case, in: Bach, Harms (Eds.), Universals in Linguistic Theory, New York: Holt, Rinehart, and Winston, 1968, pp. 1–88.

[8] K. Heninger, D. Parnas, J. Shore, J. Kallander, Software Requirements for the A-7E Aircraft - TR 3876, Tech. rep., U.S. Naval Research Laboratory (1978).

[9] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, M. Blackburn, NAT2TEST$_{SCR}$: Test case generation from natural language requirements based on SCR specifications, Science of Computer Programming 95, Part 3 (0) (2014) 275 – 297.

[10] J. Peleska, E. Vorobev, F. Lapschies, C. Zahlten, Automated Model-Based Testing with RT-Tester, Tech. rep., Universität Bremen (2011).

[11] G. Carvalho, F. Barros, F. Lapschies, U. Schulze, J. Peleska, Model based testing from controlled natural language requirements, in: International Workshop on Formal Methods for Industrial Critical Systems, 2013.

[12] A. W. Roscoe, Understanding Concurrent Systems, Springer, 2010.

[13] G. Carvalho, A. Sampaio, A. Mota, A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification, in: Proceedings of International Conference on Formal Engineering Methods, 2013.

[14] K. Jensen, L. Kristensen, Coloured Petri Nets: Modelling and Validation of Concurrent Systems, 2009. `doi:10.1007/b95112`.

[15] B. Cesar F. Silva, G. Carvalho, A. Sampaio, CPN simulation-based test case generation from controlled natural-language requirements, Science of Computer Programming`doi:10.1016/j.scico.2019.04.001`.

[16] T. Santos, G. Carvalho, A. Sampaio, Formal Modelling of Environment Restrictions from Natural-Language Requirements, in: T. Massoni, M. R. Mousavi (Eds.), Formal Methods: Foundations and Applications, Springer International Publishing, Cham, 2018, pp. 252–270.

[17] S. Barza, G. Carvalho, J. Iyoda, A. Sampaio, A. Mota, F. Barros, Model Checking Requirements, in: L. Ribeiro, T. Lecomte (Eds.), Formal Methods: Foundations and Applications, Springer International Publishing, Cham, 2016, pp. 217–234.

[18] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NUSMV: a new symbolic model checker, International Journal on Software Tools for Technology Transfer 2 (4) (2000) 410–425. `doi:10.1007/s100090050046`.
URL `https://doi.org/10.1007/s100090050046`

[19] B. Oliveira, G. Carvalho, M. R. Mousavi, A. Sampaio, Simulation of hybrid systems from natural-language requirements, in: 2017 13th IEEE Conference on Automation Science and Engineering (CASE), 2017, pp. 1320–1325. `doi:10.1109/COASE.2017.8256284`.

[20] W. Taha, A. Duracz, Y. Zeng, K. Atkinson, F. A. Bartha, P. Brauner, J. Duracz, F. Xu, R. Cartwright, M. Konečný, E. Moggi, J. Masood, P. Andreasson, J. Inoue, A. Sant'Anna, R. Philippsen, A. Chapoutot, M. O'Malley, A. Ames, V. Gaspes, L. Hvatum, S. Mehta, H. Eriksson, C. Grante, Acumen: An Open-Source Testbed for Cyber-Physical Systems Research, in: B. Mandler, J. Marquez-Barja, M. E. Mitre Campista, D. Cagáňová, H. Chaouchi, S. Zeadally, M. Badra, S. Giordano, M. Fazio, A. Somov, R.-L. Vieriu (Eds.), Internet of Things. IoT Infrastructures, Springer International Publishing, Cham, 2016, pp. 118–130.

[21] K. Claessen, J. Hughes, Quickcheck: A lightweight tool for random testing of haskell programs, in: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, ACM, USA, 2000, pp. 268–279.

[22] J. H. Andrews, L. C. Briand, Y. Labiche, Is Mutation an Appropriate Tool for Testing Experiments?, in: Proceedings of International Conference on Software Engineering, ACM, New York, 2005, pp. 402–411.

[23] F. Hariri, A. Shi, SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, ACM, New York, NY, USA, 2018, pp. 860–863.

[24] A. D. Brucker, B. Wolff, hol-TestGen, in: M. Chechik, M. Wirsing (Eds.), Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 417–420.

[25] A. D. Brucker, B. Wolff, On theorem prover-based testing, Formal Aspects of Computing 25 (5) (2013) 683–721.

[26] S. Berghofer, T. Nipkow, Random testing in Isabelle/HOL, in: Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004., 2004, pp. 230–239.

[27] W. Hong, M. S. Nawaz, X. Zhang, Y. Li, M. Sun, Using Coq for Formal Modeling and Verification of Timed Connectors, in: A. Cerone, M. Roveri (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2018, pp. 558–573.

[28] A. D. Brucker, O. Havle, Y. Nemouchi, B. Wolff, Testing the IPC Protocol for a Real-Time Operating System, in: A. Gurfinkel, S. A. Seshia (Eds.), Verified Software: Theories, Tools, and Experiments, Springer, Cham, 2016, pp. 40–60.

[29] H. Wan, G. Chen, X. Song, M. Gu, Formalization and Verification of PLC Timers in Coq, in: 2009 33rd Annual IEEE International Computer Software and Applications Conference, Vol. 1, 2009, pp. 315–323.

[30] W. Ahrendt, C. Gladisch, M. Herda, Proof-based Test Case Generation, Springer International Publishing, Cham, 2016, pp. 415–451.

[31] C. Paulin-Mohring, Modelisation of Timed Automata in Coq, in: N. Kobayashi, B. C. Pierce (Eds.), Theoretical Aspects of Computer Software, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 298–315.

[32] A. Feliachi, M.-C. Gaudel, M. Wenzel, B. Wolff, The Circus Testing Theory Revisited in Isabelle/HOL, in: L. Groves, J. Sun (Eds.), Formal Methods and Software Engineering, Springer, Berlin, Heidelberg, 2013, pp. 131–147.