Science of Computer Programming 201 (2021) 102537

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Validating, verifying and testing timed data-flow reactive systems in Coq from controlled natural-language requirements

Gustavo Carvalho *, Igor Meira

Universidade Federal de Pernambuco – Centro de Informática, 50740-560, Brazil

ARTICLE INFO

Article history:
Received 17 July 2019
Received in revised form 15 June 2020
Accepted 25 August 2020
Available online 27 August 2020

Keywords:
Timed data-flow reactive system
Interactive theorem proving
Property-based testing
Controlled natural language

ABSTRACT

Data-flow reactive systems (DFRSs) form 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 well-formedness properties of DFRS models was accomplished in a programmatic way, with no formal guarantees, and test cases were generated by translating these models into other notations. Here, we use Coq as a single framework to specify, validate and verify DFRS models. Moreover, the specification of DFRSs in Coq is automatically derived from controlled natural-language requirements, and well-formedness properties are formally verified with no user intervention. System validation is supported by bounded exploration of models; general and domain-specific system property verification is supported by the development of proof scripts, and test generation is achieved with the aid of the QuickChick tool. Considering examples from the literature, but also from the aerospace (Embraer) and the automotive (Mercedes) industries, our automatic testing strategy was evaluated in terms of performance and the ability to detect defects generated by mutation. Within seconds, test cases were generated automatically from the requirements, achieving an average mutation score of about 75%.

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 failures must be minimised, since they might be life-threating or impose significant financial losses. Modelling and formal verification are strategies employed to increase system reliability. 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). Creating models promotes a better comprehension and a more precise description of the expected behaviour. Formal verification brings certainty about properties being preserved. When formal verification is not possible or feasible, testing becomes essential since it can unveil scenarios where implementations do not work as expected.

* Corresponding author.
E-mail addresses: ghpc@cin.ufpe.br (G. Carvalho), iam2@cin.ufpe.br (I. Meira).

https://doi.org/10.1016/j.scico.2020.102537
0167-6423/

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 from specification models.

Here, we focus on models of timed 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 a model's syntax and semantics may be reluctant to adopt these formalisms. Moreover, most of these models are not available in the very beginning of the project, when usually only natural-language requirements are available. One possible alternative to overcome these limitations is to employ natural-language processing (NLP) techniques to derive the required models from natural-language specifications automatically.

1.2. Natural-language processing

Stating the desired system behaviour using models may sometimes be an obstacle for adopting MBT techniques, despite all its benefits. The model notations may be not easy to interpret by, for instance, aerospace and automotive development engineers. Hence, specialists (usually mathematicians, logicians, computer engineers and scientists) are required when such languages, and their corresponding techniques, are used in business contexts. Furthermore, most of these models are not 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 models are derived from natural-language requirements, besides applying MBT techniques, one can reason formally about specification properties that can be difficult to analyse by means of manual inspection. For instance, it might not be trivial to guarantee that a specification is deterministic by only reading its textual documentation. However, given an appropriate model of the system, one could apply model-checking techniques to prove such a property.

Typically, there is a trade-off concerning the application of NLP in MBT. Some strategies 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. On the other hand, 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.

1.3. The NAT2TEST$_{Coq}$ strategy

Seeking 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 timed data-flow reactive systems. This is part of a broader strategy (NAT2TEST [3]) that supports the generation of test cases from controlled natural-language requirements by reusing different notations and techniques.

In previous works, verifying well-formedness properties of DFRS models is accomplished in a programmatic way, with no formal guarantees: DFRS models are encoded as Java objects and the verification of well-formedness properties is accomplished by algorithms implemented in Java. It is important to say that the well-formedness properties are not necessarily true by construction. Since the DFRS models are extracted from controlled natural-language requirements, we might have inconsistencies (e.g., for some variable, we might have type inconsistencies between different requirements). Additionally, to generate test cases it is necessary to translate DFRS models into other less abstract notations, such as Software Cost Reduction (SCR [4]) and Communicating Sequential Processes (CSP [5]).

In this work, we extend the NAT2TEST strategy, using the Coq proof assistant [6] as a single framework to validate and to verify DFRS models, in addition to generating test cases. As a consequence, all DFRS well-formedness properties are formally verified, and with no user intervention. System validation is supported by bounded exploration of models. General and domain-specific system property verification is supported by the development of proof scripts. Test generation is achieved with the aid of the QuickChick tool, with no need to further translate our Coq characterisation of DFRSs into other notations. We refer to this extension of the NAT2TEST strategy as NAT2TEST$_{Coq}$.

Considering examples from the literature, but also from the aerospace (Embraer[1]) and the automotive (Mercedes) industries, our automatic testing strategy was evaluated in terms of performance and the ability to detect defects generated by mutation. Within seconds, test cases were generated automatically from the requirements, achieving an average mutation

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

score of about 75%. Discarding equivalent mutants, in the example provided by Embraer, the actual mutation score is 100%; the generated test cases were capable of detecting all systematically introduced errors.

Therefore, the main contribution of this work is a single framework for validating, verifying, and testing timed data-flow reactive systems from controlled natural-language requirements. In more detail, we have:

- A Coq characterisation of symbolic and expanded data-flow reactive systems;
- System validation via bounded exploration of models in Coq;
- System property verification via the development of proof scripts in Coq;
- Test generation from Coq models using the QuickChick tool;
- Tool support developed using the Eclipse RCP framework;
- Empirical analyses considering examples from the literature and industry.

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. Section 3 presents our characterisation of symbolic and expanded data-flow reactive systems. Section 4 shows how our Coq characterisation can be used to validate system specifications, to prove system properties, besides generating test cases with the aid of the QuickChick tool. The developed tool support, along with our empirical analyses, are presented in Section 5. Finally, Section 6 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 Fig. 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.

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 [7]). 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 [8]. 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.

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].

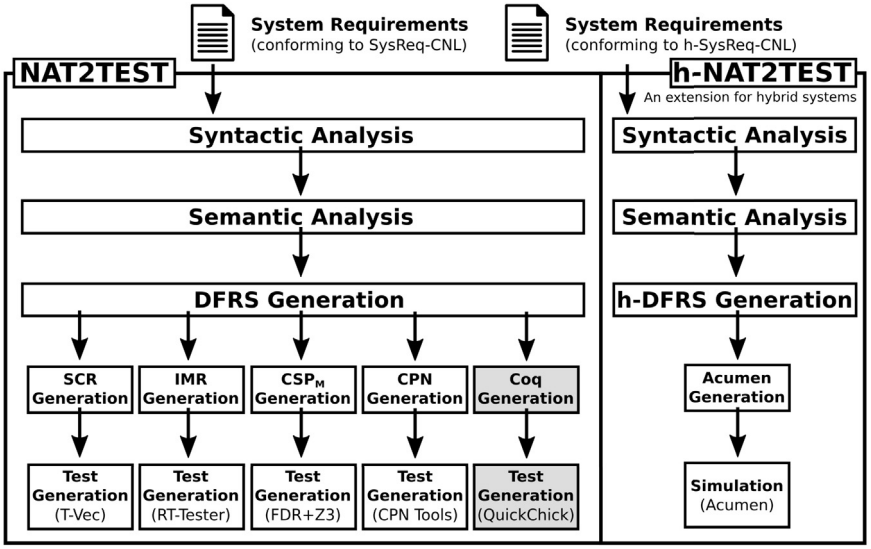


Fig. 1. NAT2TEST – a strategy for generating test cases based on different formalisms.

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*		

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 [4] (more details in [9]);
- NAT2TEST$_{IMR}$: based on *Intermediate Model Representation* – IMR [10] (more details in [11]);
- NAT2TEST$_{CSP}$: based on *Communicating Sequential Processes* – CSP [5] (more details in [12]);
- NAT2TEST$_{CPN}$: based on *Coloured Petri Nets* – CPN [13] (more details in [14]);
- NAT2TEST$_{Coq}$: based on *Coq* [6] – 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] [15]) and SMT solver (Z3[5] [16]) 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.

Each specialisation has its own advantages, summarised as follows:

- NAT2TEST$_{SCR}$: scalability, test coverage criteria;

[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.

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$_{\mathrm{M}}$	Model checking + SMT solving	FDR + Z3
NAT2TEST$_{CPN}$	CPN	Simulation	CPN Tools
NAT2TEST$_{Coq}$	Coq	Property-based testing	Coq + QuickChick

- NAT2TEST$_{IMR}$: scalability, test coverage criteria;
- NAT2TEST$_{CSP}$: formal testing theory, symbolic time representation;
- NAT2TEST$_{CPN}$: scalability, perspective of formal testing theory;
- NAT2TEST$_{Coq}$: scalability, unified and formal framework for validation, verification and testing.

The first two specialisations (NAT2TEST$_{SCR}$ and NAT2TEST$_{IMR}$) support important coverage criteria when generating test cases, such as MC/DC, which is required by standards such as DO-178C. However, the testing theory is not formal [17] and, thus, one cannot formally argue in favour of its soundness. In order to develop a formal 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).

Differently, the specialisation based on CSP (NAT2TEST$_{CSP}$) has a formal testing theory, which considers a symbolic time representation; it takes into account both discrete- and continuous-time systems. However, scalability is an issue. The specialisation based on CPNs (NAT2TEST$_{CPN}$) is better with respect to scalability, and we already have in the literature (not connected to our working context) implementation relations for CPNs.

A common limitation shared by all specialisations but NAT2TEST$_{Coq}$ is that the verification of the well-formedness properties of any valid DFRS is performed in a programmatic way using Java. Despite of all the attention to the details during the implementation, there is not definite proof that the algorithms in Java reflect precisely the formal definition of the well-formedness conditions. In NA2TEST$_{Coq}$, these properties are formally defined and checked in the same environment, with a formal guarantee (proof) that the (functional) verification reflects the (logical) formal definitions.

Another limitation of all specialisations but NAT2TEST$_{Coq}$ relates to semantic equivalence between representations. An expanded DFRS can be seen as the semantics of the corresponding symbolic DFRS. When representing, for instance, a symbolic DFRS as CSP processes, we have no guarantees that the obtained labelled transition system is semantically equivalent to the expanded DFRS. In these specialisations, we just consider the more concrete representation (e.g., CSP processes) as the semantics of DFRS models. In NA2TEST$_{Coq}$, there is no need to further translate DFRS models to other notations, since validation, verification and testing is supported in the same environment (Coq).

In summary, the NAT2TEST$_{Coq}$ specialisation distinguishes itself by creating a scalable, unified and formal framework for validating, verifying and testing timed data-flow reactive systems. The development of a formal testing theory considering this specialisation is beyond the scope of this work, but this is an interesting perspective to explore in the future.

Other extensions In [18], 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 [19], 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 [20], 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 [21], 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 [22], which is a language and tool for the specification and simulation of hybrid systems.

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

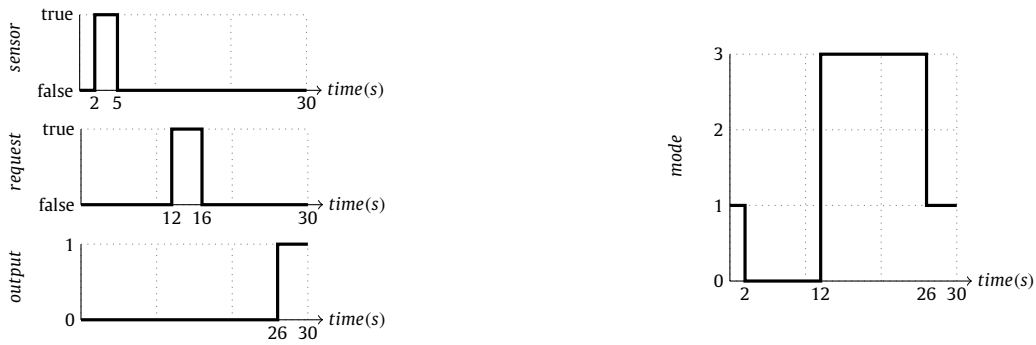


Fig. 2. Example of signals for the vending machine.

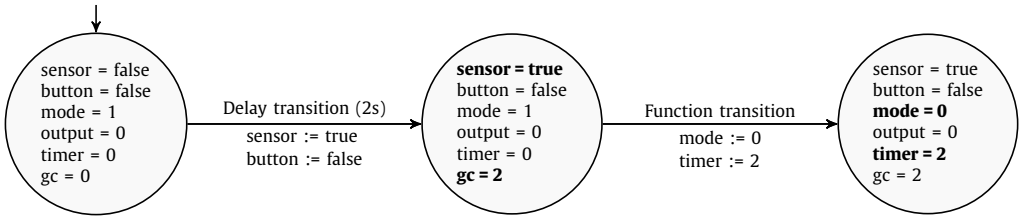


Fig. 3. Some states of the e-DFRS representation for the vending machine.

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.

Fig. 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.

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*, variable whose values are discrete or continuous 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, but also letting the time evolve. Informally speaking, a state is not stable when some system reaction is expected. In Section 3.2 we provide a formal definition for state stability. 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 Fig. 2, Fig. 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 rep-

resented 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).

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 [6], 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.

The Gallina language It is a typical functional language where it is possible to define new types, besides polymorphic and higher-order functions [6]. 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 cannot be defined.

Building proofs with tactics Proofs are developed with the aid of tactics.[6] Each tactic modifies the proof context (proof hypotheses) or the proof goal in order to demonstrate its truth. For instance, the tactics `intros`, `simpl`, and `destruct` are used to perform universal instantiation, expression simplification and case analysis, respectively. The tactic `inversion` *H* can be used to finish a proof when *H* is an absurd hypothesis in the proof context. The tactic *reflexivity* finishes the proof when the proof goal involves a trivial equality.

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. For instance, one can write `repeat` *T* to repeatedly apply the tactic *T*. It is also possible to apply different tactics based on pattern matching on the proof context and proof goal. Although the evaluation of a function in Coq necessarily terminates, tactic evaluation might not terminate, meaning that one has failed to construct a proof.

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 :=
 match n with
 | 0 ⇒ true
 | 1 ⇒ false
 | S (S n') ⇒ evenb n'
 end.
```

```
Definition is_even (n : nat) : Prop :=
  ∃ k, n = k + k.

Inductive even : nat → Prop :=
  | ev_0 : even 0
  | 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 favour their functional characterisation to enable automatic proof of model consistency.

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

2.4. Property-based testing

Testing is an extremely important task for software development, also complementary 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 [23], 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.

3. Characterisation of symbolic and expanded DFRSs in Coq

In [2], we first defined models of data-flow reactive systems. We use Z as the formalisation language to define the structure of such models and their well-formedness conditions. In this previous work, we also propose algorithms for deriving such models from controlled natural-language requirements. The verification of the well-formedness conditions is performed in a programmatic way using Java. Despite of all the attention to the details during the implementation, there is not definite proof that the algorithms in Java reflect precisely all well-formedness conditions formally defined in Z.

Here, we have carefully and systematically rewritten in Coq our previous Z characterisation of DFRS models. Sections 3.1 and 3.2 present our characterisation of symbolic and expanded DFRSs in Coq, respectively. Now, when creating an instance of a DFRS (in Coq), we are obliged to prove (in Coq) all of its well-formedness conditions (also defined in Coq). Therefore, one can define an instance of a DFRS model if, and only if, all of its well-formedness conditions are proved to hold. Moreover, as detailed in Section 3.3, this proof is performed with no need for user interaction; it is completely automatic.

The general architecture of our characterisation of DFRSs in Coq[8] is presented in Fig. 4. The modules *variables*, *states*, and *functions* define the constituent elements of a symbolic DFRS (*s_dfrs*). The module *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*). Within each module, we have a logical and a functional characterisation of well-formedness properties, besides the proof that both of them are equivalent. These equivalence proofs allow us to create instances of DFRSs, proving automatically that they are consistent.

In *s2e_dfrs*, we have functions that allow for a dynamic expansion of an e_DFRS from a given s_DFRS, besides definitions related to the verification of general system properties. 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*. In what follows, we detail our Coq characterisation of DFRSs. In terms of size (lines of code – LOC), our Coq characterisation has about 2.5 KLOC of definitions and 1.5 KLOC of proof scripts.

It is important to emphasise that, except *examples*, all modules presented in Fig. 4 are general and domain independent. The code within examples, which is automatically generated from the controlled natural-language requirements, can be seen as instances of the types defined in our general theory of data-flow reactive systems.

3.1. Characterisation of symbolic DFRSs in Coq

An s_DFRS comprises (input, output, and timer) variables, besides a global clock, along with an initial state, and a set of functions describing the system behaviour. In the following sections, we define each of these elements in Coq.

3.1.1. Variables

Let *NAME* be a string and *gc* be the name of the system global clock (the string "gc"), a variable name (*VNAME*) is defined as any element of *NAME* except by the value of *gc*. This restriction is formalised by the propositional function

[7] QuickCheck: http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html.
[8] Available online in: https://github.com/igormeira/DFRScoq.

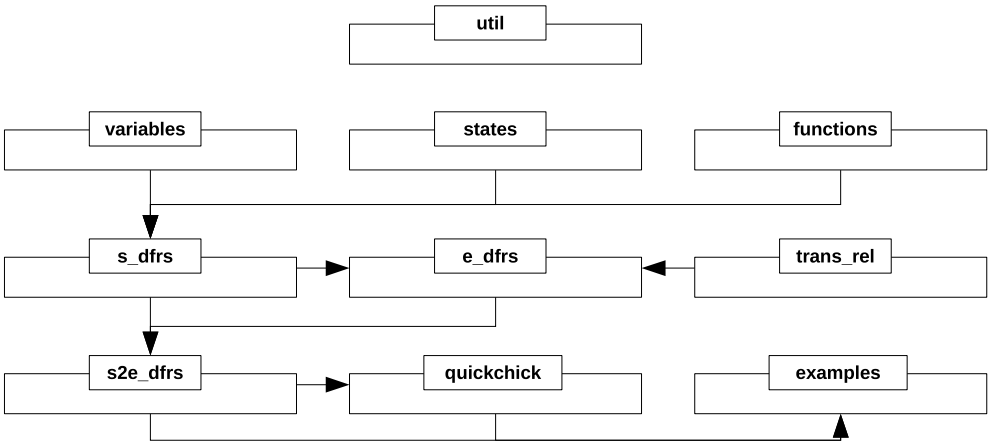


Fig. 4. Characterisation of DFRSs in Coq.

(functions that build predicates from the given arguments) *ind_rules_vname*; *string_dec* is an auxiliary propositional function that compares strings. It is important to note the usage of `Record`. In Coq, a record might comprise data values (*vname*), but also properties (predicates) that need to hold (*rules_vname*). When creating an instance of *VNAME* (i.e., declaring the name of a variable), it is necessary to prove that all defined properties hold. This feature brings cohesion between structural definition and well-formedness properties.

```
Definition NAME := string.
Definition gc : NAME := "gc".
Definition ind_rules_vname (vname : NAME) : Prop := ¬ string_dec vname gc.

Record VNAME : Set := mkVNAME {
  vname : NAME ; rules_vname : ind_rules_vname vname
}.
```

Considering the VM example, the following code illustrates how to declare the name of a variable (create an instance of the record *VNAME*). 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 well-formedness 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"* $\rightarrow$ `False`. In Coq, the logical negation ($\neg P$) is modelled as $P \rightarrow$ `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.`). As said before, this feature (records with values and predicates) prevents us from creating inconsistent instances (violating rules).

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

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).

System variables (*SVARS*), which might represent input or output variables, should be defined as any finite partial function f from *VNAME* to *TYPE* (a mapping between variable names and their corresponding types), where f is not empty and the range of f is a subset of $\{bool, int, float\}$ (the allowed types for system variables). These basic types are sufficient since we are dealing with embedded systems, where variables denote signals. In Coq, since all functions are total, we define *SVARS* as a list of pairs of variable names and types (*svars*), in conjunction with a predicate that enforce the required properties.

```
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 SVARS : Set := mkSVARS {
  svars : list (VNAME × TYPE)
  ; rules_svars : ind_rules_svars svars
}.
```

The aforementioned properties are defined with the aid of the propositional function *ind_rules_svars*. The yielded predicate is true if, and only if, the list *svars* indeed represents a function (*is_function* is an auxiliary propositional function that captures this requirement), this list is not empty and the types of the variables are within the allowed ones (*ind_svars_valid_type* is another auxiliary propositional function that captures this requirement). The name *comp_vname* refers to a global definition (function) that defines how variable names are compared. System timers (*STIMERS*) are defined analogously, but with different well-formedness properties (e.g., the type of a timer cannot be neither *bool* nor *int*, since it cannot be a non-negative number).

DFRS variables are defined as follows: a list of input (I) and output (O) variables, besides timers (T) and the system global clock (*gcvar*). The element *rules_dfrs_variables* captures the well-formedness properties of DFRS variables.

```
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 (iii) we have type consistency between timers and the global clock (they share the same type).

3.1.2. Initial state

A state is a list of names mapped to a pair of values. The pair values are the previous/current variable values. Keeping in the state the previous value allows for triggering system reactions in the exact moment a variable changes from one particular value to another. The property *rules_state* enforces that *state* is also a function.

```
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
}.
```

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 *rules_dfrs_functions*). Each function (*function*) is a list of 4-tuples: a static guard (*EXP*), a timed guard (*EXP*), a list of assignments (*ASGMTS*), and requirement traceability information (*REQUIREMENT*). 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 *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". In what follows, we show how this requirement is formalised in Coq. First, three conditions are defined to capture the conditions mentioned by REQ001. The term *DIS* refers to a list of disjunctive clauses; in these conditions, we have a single element in each list of disjunctive clauses.

```
Definition REQ001_sg_disj1 : DISJ.
Proof. apply (mkDISJ [mkBEXP (previous (the_coin_sensor)) eq (b false) ]). (* proof *) Defined.
Definition REQ001_sg_disj2 : DISJ.
Proof. apply (mkDISJ [mkBEXP (current (the_coin_sensor)) eq (b true)]). (* proof *) Defined.
Definition REQ001_sg_disj3 : DISJ.
```

```
Proof. apply (mkDISJ [mkBEXP (current (the_system_mode)) eq (i 1)]). (* proof *) Defined.
```

Then, the static guard (*REQ001_sg*) related to the requirement REQ001 is defined as the conjunction of the three previously defined conditions. Therefore, we have a conjunction of three elements, each one with a single disjunction. In our work, the static and timed guards adhere to a Conjunctive Normal Form (CNF). Since this requirement is not dependent on time information, its corresponding timed guard (*REQ001_sg*) comprises an empty list of conditions.

```
Definition REQ001_sg : EXP.
Proof. apply (mkCONJ [REQ001_sg_disj1 ; REQ001_sg_disj2 ; REQ001_sg_disj3]). Defined.
Definition REQ001_tg : EXP.
Proof. apply (mkCONJ []). Defined.
```

Now, we define the expected system reaction: when the aforementioned conditions are met, the system reacts by resetting the request timer (*REQ001_asgmt1*), and assigning choice (*i 0*) to the system mode (*REQ001_asgmt2*). These two assignments are considered the expected system reaction (*REQ001_asgmts*).

```
Definition REQ001_asgmt1 : ASGMT.
Proof. apply (mkASGMT (the_request_timer, (n 0))). Defined.
Definition REQ001_asgmt2 : ASGMT.
Proof. apply (mkASGMT (the_system_mode, (i 0))). Defined.
Definition REQ001_asgmts : ASGMTS.
Proof. apply (mkASGMTS [REQ001_asgmt1 ; REQ001_asgmt2]). (* proof *) Defined.
```

The omitted proofs guarantee that all definitions meet the well-formedness conditions. As explained in Section 3.3, all of them are discharged automatically; no need for the user to write an example-specific proof script. Regardless of the system, the proof script is always the same.

3.1.4. s_DFRSs

An s_DFRS is defined as follows. Various consistency properties are enforced by *rules_s_dfrs*; for instance, the initial state must provide values for all system variables, the static and timed guads, as well as the assignments, must also be type consistent with the declaration of system variables. We refer to our git repository for the details; all Coq code is available there.

```
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
}.
```

It is import to say that the well-formedness properties are not necessarily true by construction. Since the definitions are extracted from controlled natural-language requirements, we might have inconsistencies. For example, one requirement might assign a boolean value to a variable, whereas other requirement assigns an integer to the same variable. If such well-formedness problems occur, the proof scripts will fail. The fail will indicate which property does not hold.

3.2. Characterisation of expanded DFRSs in Coq

An e_DFRS is defined in terms of a transition relation (*TRANSREL*), which comprises a list of transitions. Each transition (*TRANS*) 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.

```
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
}.
```

The transition relation of an e_DFRS is defined by *DFRS_TRANSITION_RELATION*, also considering its well-formedness properties.

```
Record DFRS_TRANSITION_RELATION := mkDFRSTRANSITIONREL {
  TR : TRANSREL ;
  rules_TR : ind_rules_TR TR
}.
```

A number of consistency rules are enforced by *rules_TR*. For instance: the source state of a function transition must not be stable, and the target state of the function transition should reflect the corresponding assignments applied to the source state. A state is said to be stable if, and only if, there is no system reaction (represented as a function transition) expected on this state. The propositional function *is_stable* formalises this concept. Recall that a symbolic DFRS comprises a list of functions, and each function comprises a list of entries. In *is_stable*, we define *entries* as being all function entries among all functions. Then, stability is defined with the aid of the recursive function *is_stable_entry*.

```
Definition is_stable (s : STATE) (IO T : list (VNAME × TYPE))
  (F : list (list FUNCTION)) : bool :=
  let entries := union_lists (map (fun f : FUNCTION ⇒ f.(function)) (union_lists F))
  in is_stable_entry s IO T entries.
```

The function *is_stable_entry* iterates over the list of function entries trying to find an entry whose static and timed guard are met, besides leading to a different state when performing the corresponding assignments (i.e., by performing the function transition some system variable is updated). If one entry meeting these criteria is found, the state is not stable. Otherwise, it is said that the state is stable.

```
Fixpoint is_stable_entry (s : STATE) (IO T : list (VNAME × TYPE))
  (le : list (EXP × EXP × ASGMTS × REQUIREMENT)) : bool :=
  match le with
  | [] ⇒ true
  | h :: t ⇒ if (negb (static_guards_true s (fst3 (fst h)).(conjs) IO T))
                || (negb (timed_guards_true s (snd3 (fst h)).(conjs) T))
                || beq_state_current s (nextState s T (trd3 (fst h)))
             then is_stable_entry s IO T t else false end.
```

Therefore, when firing a function transition, there should be at least one function entry in the symbolic DFRS whose static and timed guards evaluate to true in the source state. Differently, the source state of a delay transition must be stable, and the target state of the delay transition should reflect the time elapsed and new values for system inputs. See our previous example (Fig. 3).

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 typically comprises an infinite number of states. If considering real-time delays, it would become even uncountably branching. Therefore, it is not possible to create an instance of *e_DFRS* by enumerating all of its states. A function that expands a symbolic DFRS (by computing its function and delay transitions), yielding the obtained e_DFRS, would never terminate its execution and, thus, cannot be defined in Coq in a straightforward way. Although it is not possible to perform a full computation of an e_DFRS from its corresponding s_DFRS, it is possible to perform bounded exploration of the former. This is formalised and described later (Section 4.1).

3.3. Functional characterisation of well-formedness properties

In Section 3.1.1, when defining the element *the_coin_sensor*, it was necessary to prove that the variable name is not "gc". When more complex well-formedness rules need to be proved, the proof script becomes equally more complex, which inhibits automation. A workaround consists in providing functionally-defined well-formedness rules, which are logically equivalent to their logical/inductive counterparts.

In this work, we first defined all properties in logical terms, since it is closer to our Z formalisation. Afterwards, we defined computable procedures (functions) that check whether the same properties hold. Finally, we proved that, for any valid DFRS (regardless of the example), these functions are semantically equivalent to the logical characterisation; the function yields true if, and only if, the corresponding predicate is also true.

Now it is possible, for instance, to define *the_coin_sensor* as follows. We apply the theorem *theo_rules_vname* to rewrite the proof goal considering the functional characterisation of the well-formedness properties. 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. Therefore, all well-formedness proofs are discharged automatically, with no user of user intervention, regardless of the example. Although proving *theo_rules_vname* is quite straightforward, other equivalence proofs are more complex. As said in the beginning of Section 3, we have written about 1.5 KLOC of proof scripts.

4. Validating, verifying and testing DFRSs

In this section, we get into details about three applications of our Coq characterisation of DFRSs. First, in Section 4.1, we describe how an expanded DFRS can be dynamically built up to a given bound from a symbolic DFRS, which can be used as a validating mechanism to assess whether the requirements indeed capture the intended system behaviour. Then, in Section 4.2, we discuss how our formal characterisation of data-flow reactive systems in Coq can be used to prove general and domain-specific properties. We illustrate this application considering two general properties: determinism and absence of time lock. Finally, in Section 4.3, we define two mechanisms for generating test cases from our formal characterisation: by sampling valid traces and guided by user-defined test purposes.

4.1. Validating system requirements via bounded exploration of models

An important development task is requirement validation: assessing whether the written requirements (and created models) indeed reflect the desired system behaviour. Model simulation is a relevant validation technique, since it enables the analysis of whether the created models properly capture the expected system behaviour. In this work, by performing bounded exploration of an e-DFRS, one might observe an undesired system reaction due to a miswritten requirement. In such a situation, the requirements should be rewritten, the models regenerated, and the simulation/analysis should be performed once more.

To support such a task, we define the function *buildTR*, which dynamically expands an s-DFRS, bounded by the value of *num*, which limits the number of recursive calls (typically, an e-DFRS comprises an infinite number of states). More precisely, the function *buildTR* yields part of the transition relation of an e-DFRS. Recall that a transition relation (*TRANSREL*) is defined as a list of transitions (see Section 3.2); each transition relates two states by means of a transition label (delay or function transition).

Initially, *toVisit* has a single element (the initial state of the s-DFRS), and *visited* is empty. With the aid of *genTransitions*, we generate all emanating transitions from the initial state; if it is stable, we will have delay transitions, otherwise, only function transitions. State stability has been formally defined in Section 3.2. Then, the current state is removed from *toVisit*, and the reached states are added to *toVisit*. Afterwards, this function recurses considering the predecessor of *num*. When it reaches 0, the function stops.

```
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.
```

We note that we need to check whether the current state (the head of *toVisit*) has already been visited in the past. This might occur in the presence of time locks, when we have a loop of function transitions (see Section 4.2.2).

The parameter *possibilities* list the values of input variables that will be considered during automatic bounded exploration. In the NAT2TEST tool (see Section 5.1), the user can inform such lists for input variables. This information is necessary, for instance, when dealing with integers, since considering a large number of values would rapidly reach the exploration bound; recall that *genTransitions* generates all emanating transitions from a given state. When the state is stable, the generated delay transitions consider all possible combinations of input values. When simulating/testing a system, typically, it is not necessary to consider the full range of possible values; it is reasonable to restrict ourselves to a subset of values applying a technique known as input space partitioning [24].

The auxiliary function *genTransitions* is defined as follows. It assesses whether a given state is stable. 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).
```

The variable *combinations* considers all possible value combinations for input variables, restricted to the values defined in *possibilities*. For instance, the VM has two boolean inputs (*the_coin_sensor* and *the_coffee_request_button*) and, thus, *combinations* will have 4 different assignment possibilities: (false, false), (false, true), (true, false), and (true, true).

Fig. 5 shows an application of *buildTR* considering the initial state of the VM (s1), a time step of 1s, and a bound *num = 3*. Since s1 is stable, we have only delay transitions (abbreviated as *D*) departing from it. We have one transition for each possible combination of input values; 4 in total, since the system inputs are two boolean variables. After generating four new states (s2, s3, s4, and s5), *buildTR* recurses considering the first generated state (s2) and a bound *num = 2*. Since s2 is also stable, 4 new states are generated (s6, s7, s8, and s9). Now, *buildTR* recurses again, considering the second generated state (s3) and a bound *num = 1*. This state is not stable: when the system mode is idle (*mode = 1*), and the sensor changes to true, the system reacts by resetting the request timer and moving to the choice mode (*mode = 0*). Therefore, a function transition is generated reaching s10. Finally, *buildTR* recurses again considering the third generated state (s4) and a bound *num = 0*. Since the bound is equal to 0, the function terminates.

This bounded and functional exploration can support the development of simulators for e_DFRSs, since it is possible to extract Haskell and OCaml code directly from functional definitions in Gallina.

4.2. Verifying general and domain-specific properties

In some situations, mainly when dealing with safety-critical systems, it is desired to have a definite evidence about the satisfiability of some general (e.g., determinism, absence of time lock) or domain-specific property (e.g., some safety/liveness statement). For instance, when developing safety-critical functions, the implementation is usually expected to be deterministic; the system shall always exhibit the same behaviour when provided with the same input stimuli. Similarly, a DFRS is expected to be time-lock free; it shall not have a loop of function transitions that prevent it from reaching a stable state.

Our Coq characterisation also supports the verification of general and domain-specific properties, since proof scripts can be developed to prove desired properties. Here, we show how our characterisation can be used to prove two general properties: determinism (Section 4.2.1) and absence of time lock (Section 4.2.2).

4.2.1. Proving determinism

In this work, non-determinism can be seen as the possibility of reaching two or more different stable states from the same non-stable state; in other words, more than one admissible system reaction for the input stimuli. Fig. 6a illustrates such a situation. Let the initial state be stable, and that after receiving a new input stimuli (delay transition – assigning 1 to `in1` after 1s), the system reaches a non-stable state, where two system reactions are admissible: assigning 1 or 2 to the variable `out1`. In this example, we also consider that the two bottom-most states are stable.

In Fig. 6b, we consider that only the right-most state of the second row is stable. Therefore, although there are two different function transitions emanating from the non-stable state, both paths converge to the same stable state and, thus, only one admissible system behaviour will be observed. Recall that function transitions occur instantaneously (i.e., time does not evolve when performing function transitions).

Before formalising in Coq the notion of deterministic DFRSs, we first need to define the concept of traces of function transitions (*ind_func_trace*). In the following definitions consider that *I*, *O*, and *T* denote the system variables (inputs, outputs,

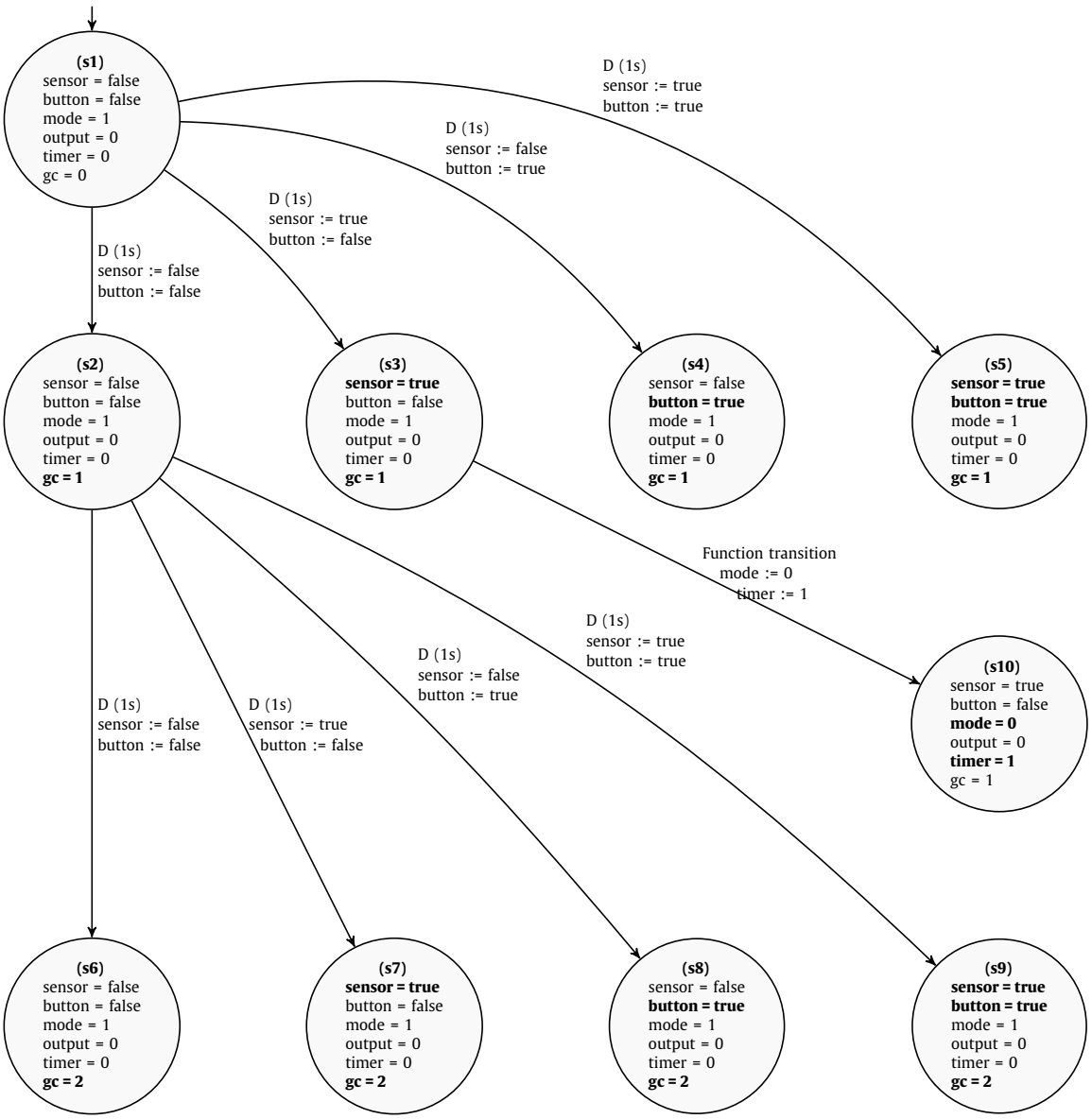


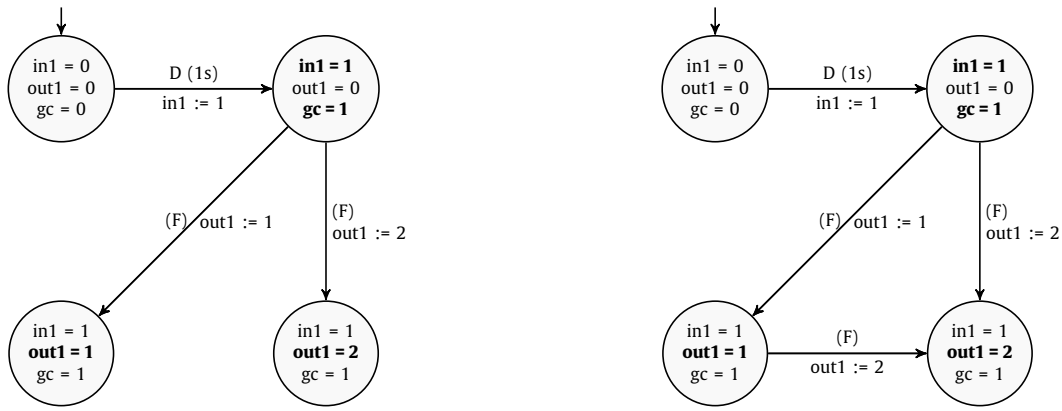
Fig. 5. Bounded exploration of the vending machine e-DFRS.

and timers, respectively), *IOT* denotes the collection of inputs, outputs, and timers, whereas *All* denotes all system variables (*IOT* and the system global clock). The symbol *F* represents the functions describing the system behaviour, and *entries* is the collection of function entries of all system functions (i.e., a flat representation of *F*).

We define function traces (a sequence that only comprises function transitions) inductively. The first case (*func_traces_empty*) states that the empty sequence is a valid function trace for any state *s*, assuming that *s* is well typed with respect to *All*, and that *s* is not stable. By being well typed we mean that it provides a valuation for all system variables, and the values are also valid with respect to the type of the variables.

The second case (*func_traces_step*) allows the definition of other function traces considering an inductive principle. Let *t* be a valid function trace for *s* (*ind_func_trace sdfrs s t*), let *s'* be the non-stable state reached by performing *t* from *s*, appending to *t* any label *l* emanating from *s'* also leads to a valid function trace for *s*. We do not present here the auxiliary definition of *reached_func_state*.

```
Inductive ind_func_trace : s_DFRS → STATE → list TRANS_LABEL → Prop :=
  | func_traces_empty :
    ∀ (sdfrs : s_DFRS) (s : STATE),
      let I := sdfrs.(s_dfrs_variables).(I).(svars) in
```



(a) Non-deterministic behaviour

(b) Deterministic behaviour

Fig. 6. Example of non-deterministic and deterministic behaviour.

```
    let O := sdfrs.(s_dfrs_variables).(O).(svars) in
    let T := sdfrs.(s_dfrs_variables).(T).(stimers) in
    let IOT := List.app (List.app I O) T in
    let All := List.app ([sdfrs.(s_dfrs_variables).(gcvar)])
                        (map (fun x : (VNAME × TYPE)
                          ⇒ ((fst x).(vname), snd x)) IOT) in
    let F := sdfrs.(s_dfrs_functions).(F) in
      well_typed_state s All ∧ (is_stable s (List.app I O) T [F]) = false
      → ind_func_trace sdfrs s []
| func_traces_step :
  ∀ (sdfrs : s_DFRS) (s : STATE) (t : list TRANS_LABEL),
    let I := sdfrs.(s_dfrs_variables).(I).(svars) in
    let O := sdfrs.(s_dfrs_variables).(O).(svars) in
    let T := sdfrs.(s_dfrs_variables).(T).(stimers) in
    let IOT := List.app (List.app I O) T in
    let All := List.app ([sdfrs.(s_dfrs_variables).(gcvar)])
                        (map (fun x : (VNAME × TYPE)
                          ⇒ ((fst x).(vname), snd x)) IOT) in
    let F := sdfrs.(s_dfrs_functions).(F) in
    let entries := union_lists
                     (map (fun f : FUNCTION ⇒ f.(function)) (union_lists [F])) in
      well_typed_state s All ∧ (is_stable s (List.app I O) T [F]) = false
      ∧ ind_func_trace sdfrs s t
      → ∀ (s' : STATE), s' = reached_func_state sdfrs s t
        ∧ (is_stable s' (List.app I O) T [F]) = false
        → let L := map (fun (t : TRANS) ⇒ snd3 t.(STS))
                       (make_trans_func s' (I ++ O) T entries) in
          ∀ (l : TRANS_LABEL), In l L → ind_func_trace sdfrs s (t ++ [l]).
```

Now, we formalise the concept of determinism (*deterministic*) as follows. We say that a symbolic DFRS is deterministic if, and only if, for any non-stable state *s*, which is reachable from the initial state of the symbolic DFRS, and two function traces *t1* and *t2*, the stable states reached by *t1* and *t2* from *s* must be the same. The propositional function *reachable_state_dfrs sdfrs s* yields a predicate stating that there is a trace from the initial state of *sdfrs* leading to *s*. The omitted (auxiliary) definitions can be seen in our GitHub repository.

```
Definition deterministic (sdfrs: s_DFRS) : Prop :=
  let I := sdfrs.(s_dfrs_variables).(I).(svars) in
  let O := sdfrs.(s_dfrs_variables).(O).(svars) in
  let T := sdfrs.(s_dfrs_variables).(T).(stimers) in
  let F := sdfrs.(s_dfrs_functions).(F) in
  ∀ (s : STATE) (t1 t2 : list TRANS_LABEL),
    let s1 := reached_func_state sdfrs s t1 in
```

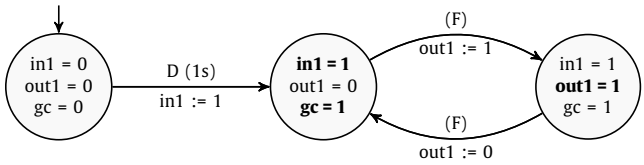


Fig. 7. Example of time lock.

```
let s2 := reached_func_state sdfrs s t2 in
reachable_state sdfrs s ∧
ind_func_trace sdfrs s t1 ∧ ind_func_trace sdfrs s t2 ∧
(is_stable s1 (List.app I O) T [F]) = true ∧
(is_stable s2 (List.app I O) T [F]) = true
→ s1 = s2.
```

Regarding the VM, it is deterministic since the predicate *deterministic vm_s_dfrs* reduces to true. Briefly speaking, for any non-stable state, there is only one requirement whose conditions are met. After performing the assignments associated with these conditions, a single stable state is reached. Therefore, the specification is deterministic.

4.2.2. Proving absence of time lock

In this work, time lock can be seen as a loop of function transitions. Such a loop would prevent the system from reaching a stable state and, thus, time elapsing. This is obviously an undesired property due to a specification error. Fig. 7 illustrates such a situation. After performing the delay transition *D (1s)*, the system reaches a loop of function transitions: alternating the assignment of 1 and 0 to *out1*.

In Coq, this concept is formalised as follows (*no_time_lock*). Let *visited_func_states* be an auxiliary function that given a function trace t and a state s yields the non-stable states visited by performing t from s (including the source s), an s-DFRS is free of time lock if, and only if, for any non-stable state s, which is reachable from the initial state of the symbolic DFRS, and function trace t of s, the number of occurrences of s in the list of visited states by t is equal to 1; it is not possible to come back to s (i.e., no loop of function transitions).

```
Definition no_time_lock (sdfrs : s_DFRS) : Prop :=
  ∀ (s : STATE) (t : list TRANS_LABEL),
    let S := visited_func_states sdfrs s t in
    reachable_state sdfrs s ∧ ind_func_trace sdfrs s t → count_occ_states S s = 1.
```

The VM is also free of time lock: the predicate *no_time_lock vm_s_dfrs* holds. The proof is similar (in structure) to the one presented in the previous section. Briefly speaking, when a state is not stable, there is only one requirement whose conditions are met. After performing the assignments associated with these conditions, a stable state is reached and, thus, no loop of function transitions occur.

4.3. Generating test cases with QuickChick

Besides validation via bounded simulation, and property verification via proof development, our Coq characterisation of DFRSs also supports generation of test data with the support of QuickChick. In general, test cases are generated or written according to some test adequacy criterion [23], such as the following ones: node, edge, and path coverage [24].

Nevertheless, it has been shown that, in some situations, the differences in effectiveness between random testing and test cases generated/written according to some test adequacy criterion are small [25,26]. For instance, in Section 5.2, considering a mutant-based strength analysis, we show that, for the priority command function (PC), randomly generated test scenarios outperform test cases written by domain specialists; the former was capable of detecting all non-equivalent mutants, this was not the case regarding the latter. Additionally, we also show that, across multiple runs, there is a little variation regarding the number of mutants detected by different sets of randomly generated test cases (see Figs. 12 and 13). Additionally, random testing has the benefit of possibly exploring scenarios that, although admissible, might not be considered by other testing approaches since they are not likely to happen in practice.

Despite of what has been said, typically, it is still useful to consider some test adequacy criterion in order to guarantee that important test scenarios are always considered by the test campaign. In this work, we support both approaches: one can generate test data by sampling valid traces (Section 4.3.1), which is better than arbitrary random traces that might not be valid, but also guided by user-defined test purposes (Section 4.3.2).

4.3.1. Test generation via sampling valid traces

As discussed in Section 2.4, property-based testing (supported by tools such as QuickChick) 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.

The test generation strategy presented in this section makes uses of (ii) random generators of valid traces, and (iii) printers for presenting the generated traces. A *trace* is formally defined as a list of transition labels (delay or function transition).

```
Definition trace := list TRANS_LABEL.
```

The definition of printers for traces and transition labels is straightforward. For instance, for transition labels (a delay transition has two elements – a delay and a list of assignments; a function transition has two elements – a list of assignments and traceability information), we define the function *string_of_trans_label*. Given a transition label, it yields the corresponding string representation. Then, we create an instance of the `Show` typeclass considering transition labels (*TRANS_LABEL*). Therefore, whenever it is necessary to show a string representation of a transition label, the function *string_of_trans_label* is called.

```
Definition string_of_trans_label (tl : TRANS_LABEL) : string :=
  match tl with
  | func tl' ⇒ "func (" ++ show (fst tl') ++ ", " ++ (snd tl') ++ ")"
  | del tl' ⇒ "del (" ++ show (fst tl') ++ ", " ++ show (snd tl') ++ ")"
  end.
Instance showTransLabel : Show TRANS_LABEL :=
  {
    show := string_of_trans_label
  }.
```

To generate valid traces, we define a function (*genValidTrace*) that, given an s_DFRS, yields random valid traces. To generate valid sequences of transitions, this function relies upon *genTransitions*, previously defined (see Section 4.1). From the initial state, it generates all possible transitions emanating from this state. 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 due to the combinator *freq*, which takes a list of generators, each one tagged with a natural number that serves as the weight of that generator. For instance, if we had *freq [(1, G1), (3, G2)]*, the generator *G2* would have three more chances to occur than the generator *G1*.

```
Fixpoint genValidTrace (st : STATE) (dfrs : s_DFRS)
(possibilities : list (VNAME × list VALUE)) (size : nat) : G trace :=
  match size with
  | 0 ⇒ ret []
  | S size' ⇒ let
      tr := (genTransitions st dfrs.(s_dfrs_variables).(I).(svars)
               dfrs.(s_dfrs_variables).(O).(svars) dfrs.(s_dfrs_variables).(T).(stimers)
               [dfrs.(s_dfrs_functions).(F)] possibilities).(transrel)
    in freq [ (1, ret []) ;
              (size, x ← next_label st tr ;;
                   xs ← genValidTrace (nextState st x tr) dfrs possibilities size' ;; ret (x :: xs)) ]
```

To generate a number of random valid traces, we sample the function *genValidTrace*. For the VM, we have the following command. When we sample *genValidTrace* with QuickChick, by default, it performs 11 calls to the sampled function. For a comprehensive explanation of typeclasses, generators, and combinators in the context of QuickChick, we refer to the QuickChick volume of the Software Foundation series.[9]

```
Sample (genValidTrace vm_initial_state vm_s_dfrs vm_possibilities 600).
```

Table 3 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.

In this example, a coin is inserted and the coffee request button is pressed at the same 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

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

Table 3
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

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.

The time step needs to be set carefully, since a low granularity (high value) might prevent from seeing the exact moment the system reacts to some stimuli. However, a granularity too high (low value) might be responsible for generating excessive long test sequences, where we see the time advancing in small steps. In our empirical analyses (Section 5.2, as we only have integer-valued time constraints, we opted for the smallest possible (integer) time step, which is 1, to prevent us from not seeing some system reaction.

4.3.2. Test generation guided by test purposes

The test generation strategy presented in this section is built upon the one presented just before. Here, besides random generators and printers, we also define an executable property P, which will guide the generation process. This property will be defined in terms of a test purpose. A test purpose (*test_purpose*) is a list of test purpose steps (*test_purpose_step*). There are two possible types of steps: any observed value is valid (*any_values*) or some specific values (mapped to variable names) are expected to be seen (*match_values*).

```
Inductive test_purpose_step : Type :=
  | any_values : test_purpose_step
  | match_values : list (NAME × VALUE) → test_purpose_step.
Definition test_purpose := list test_purpose_step.
```

Regarding the VM, the following test purpose (*vm_tp1*) defines that, when generating test data, after the initial state, it is expected to observe a state where the value of *the_coin_sensor* is equal to *true*. Then, after an arbitrary number of states (matching any values), the test should drive the system to a state where the value of *the_system_mode* is equal to 3 (*weak* mode – a weak coffee is going to be produced).

```
Definition vm_tp1 := [
  match_values [ ("the_coin_sensor", b true) ]
  ; any_values
  ; match_values [ ("the_system_mode", i 3) ]
].
```

The property P is defined in terms of whether there is a trace t that drives the system through a sequence of states that match the definition of a given test purpose. To define such a property, we rely on the function *match_purpose_trace*. Since the test purpose step *any_values* might match with an arbitrary number of states, we need to define a bound (*size*) to guarantee that the function will always terminate. When calling this function, we provide the same value provided to *genValidTrace*, which is a natural choice, since a test purpose is limited by the size of the generated trace.

```
Fixpoint match_purpose_trace (sdfrs : s_DFRS) (s : STATE)
  (t : trace) (tp : test_purpose) (size : nat) : bool :=
  let T := sdfrs.(s_dfrs_variables).(T).(stimers) in
  let s' := e_dfrs.nextState s T (get_fst_asgmts t) in
  match tp,size with
  | [], _ ⇒ true
  | _, 0 ⇒ false
  | tp_h :: tp_tl, S size' ⇒
      match tp_h with
      | any_values ⇒
          let next_values := next_match_values tp_tl in
          match next_values with
          | None ⇒ true
          | Some values ⇒ if match_state_values s' values
```

```
                    then match_purpose_trace sdfrs s' (List.tl t) (values_tl tp_tl) size'
                    else match_purpose_trace sdfrs s' (List.tl t) tp size'
        end
    | match_values values ⇒ if match_state_values s' values
                      then match_purpose_trace sdfrs s' (List.tl t) tp_tl size'
                      else false
    end
end.
```

If the test purpose steps defined by *tp* are all fulfilled by the given trace (i.e., the recursive call of *match_purpose_trace* leads to an empty test purpose), we say that we found a trace that matches the provided test purpose. If *size* reaches 0, the given trace does not match the given test purpose.

When the test purpose is not empty (there is at least one test purpose step), we match on the type of the first test purpose step (*tp_h* – the test purpose head). If we have *match_values*, we check whether performing the assignments associated with the first transition label of *t* leads to a state *s':= e_dfrs.nextState s T (get_fst_asgmts t)* where we observe the values defined in the current test purpose step. This analysis is performed by means of the auxiliary function *match_state_values*). If there is a match, we continue analysing the remainder of the test purpose (*tp_tl* – test purpose tail) and of the trace (*List.tl t* – the tail of *t*). Otherwise, the function terminates yielding *false*: the given trace does not match the provided test purpose.

If the current test purpose step is *any_values*, we interpret this as a wild card. We retrieve the values associated with the next test purpose step that is not *any_values* (*next_values*), if the reached state *s'* matches these values, we discard all *any_values* up to this test purpose; performed by the auxiliary function *values_tl*. Otherwise, we recurse consuming the first element of the trace *t*, but we keep intact the test purpose; note that *tp* (the entire test purpose) is passed as argument to the recursive call.

Regarding the VM, considering the definition of *match_purpose_trace* and the previously defined test purpose (*vm_tp1*), we can define an executable property as follows. Given a trace *t*, this definition (*m_tp1_property*) states that this trace drives the system through a sequence of states that does not match the test purpose *vm_tp1*. The symbol ? is related to the typeclass `Dec`, which defines what it means for a proposition *P* to be decidable (computable). Again, we refer to the QuickChick documentation for more information.

```
Definition vm_tp1_property (t : trace) :=
  match_purpose_trace vm_s_dfrs vm_s0.(s0) t vm_tp1 600 = false ?.
```

Now, we can invoke QuickChick, as follows, to find a counterexample for *vm_tp1_property* by generating random valid traces. If a counterexample is found, we have obtained a trace *t* that drives the system through a sequence of states that matches the given test purpose. In this way, with knowledge about the domain, one can guide the generation of test cases by user-defined test purposes, and, thus, guarantee the generation of test cases to verify the system behaviour against particular and relevant scenarios.

```
QuickChick (forAll (genValidTrace vm_initial_state vm_s_dfrs vm_possibilities 600) vm_tp1_property).
```

We conclude here the explanation of applications of our Coq characterisation of DFRSs; particularly, highlighting the benefit of relying on a single computer-verifiable framework for validating, verifying, and testing timed data-flow reactive systems.

5. Tool support and empirical analyses

The testing strategy presented in this paper has been integrated into the NAT2TEST tool [3]. In Section 5.1 we detail this integration, and in Section 5.2 we discuss the empirical analyses performed.

5.1. Tool support

The NAT2TEST tool is a multi-platform tool written in Java, using the Eclipse RCP framework.[10] It supports writing the system requirements adhering to the SysReq-CNL grammar, presenting the corresponding syntax tree if the requirement is valid according to this grammar (see Fig. 8).

After processing all system requirements, and inferring the corresponding requirement frames (see Section 2.1), a symbolic DFRS is generated and informally represented as a Java object. In Fig. 9, one can see a tabular representation of the s-DFRS function of the VM. The first two columns show static and timed guards, the third column shows the corresponding assignments, and the last one has traceability information. For example, the first line presents the function entry for the

[10] Eclipse RCP: http://wiki.eclipse.org/index.php/Rich_Client_Platform.

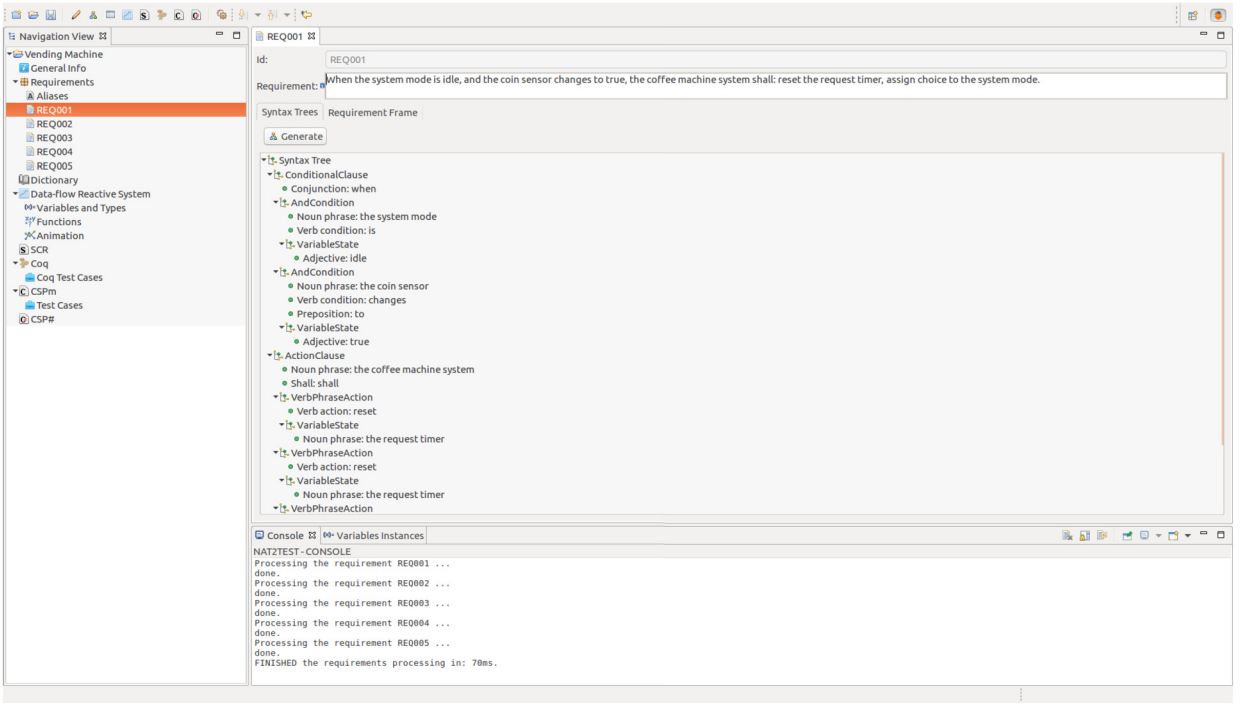


Fig. 8. NAT2TEST tool – writing requirements adhering to the SysReq-CNL grammar.

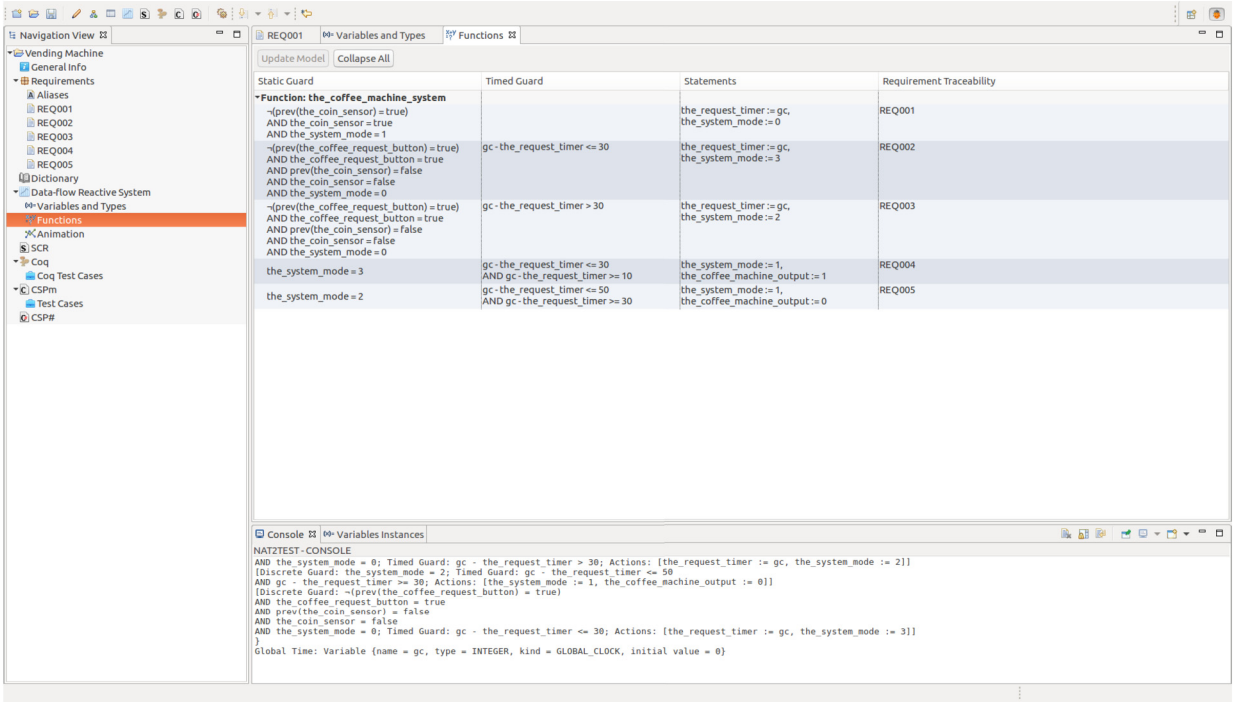


Fig. 9. NAT2TEST tool – s-DFRS function for the VM.

requirement REQ001 (*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*).

In this work, we integrated into this tool a Coq code generator (about 1 KLOC). The informal Java representation of a symbolic DFRS is translated to a Coq specification, considering the definitions detailed in Section 3. In Fig. 10, one can see

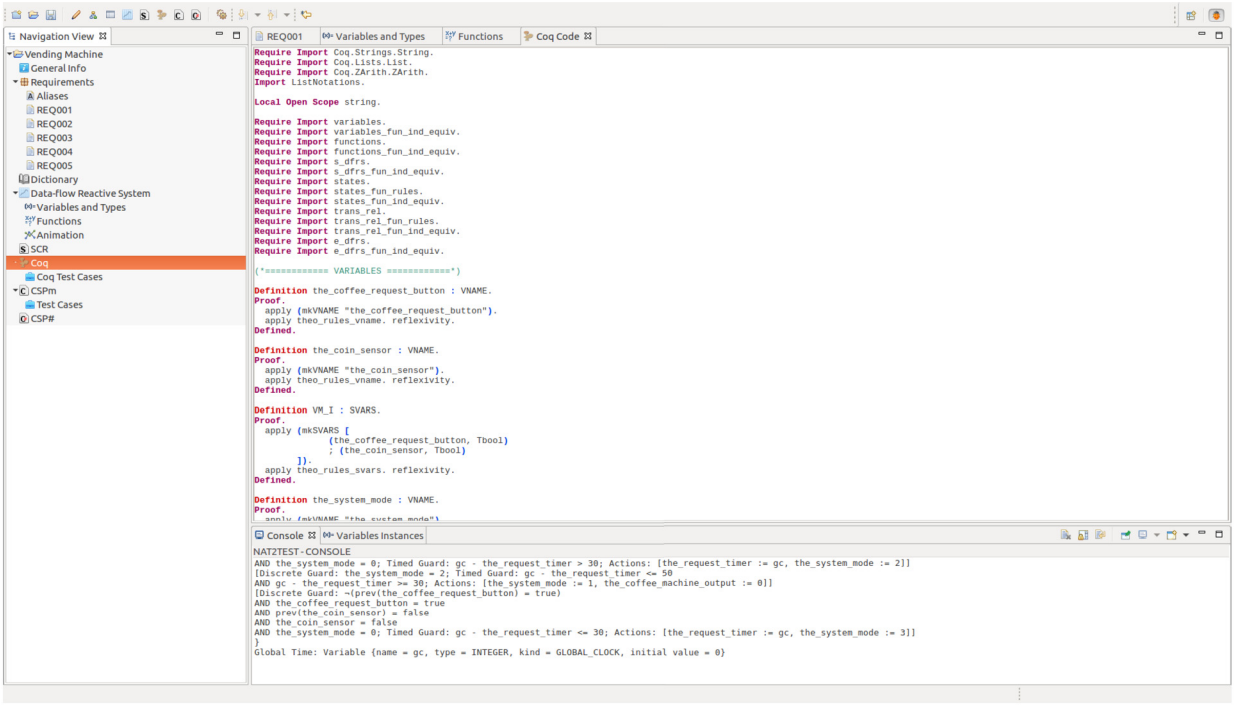


Fig. 10. NAT2TEST tool – generated Coq specification for the VM.

the beginning of the Coq specification for the VM example. For instance, it is possible to see the definition of the variable *the_coin_sensor*. The translation from Java to Coq is straightforward (almost 1:1), since our Java classes reflect the structure defined for s-DFRSs.

Nevertheless, we emphasise that the generated Coq code is then formally checked to assess whether all well-formedness properties hold. Therefore, if an error related to these properties is introduced into the Coq definition (due to an specification error or translation error), some proof obligation would not be discharged; indicating, thus, a property violation.

Finally, after deriving the Coq code, one can rely on the testing strategy described in Section 4.3 to generate test cases. It is important to bear in mind that test cases are generated in a fully automatic way from the system requirement described in natural language. Fig. 11 shows in tabular form a test case generated for the VM. This test data can be further used to simulate and to verify other system models (potentially more concrete, such as Simulink diagrams) or even as test input for hardware-in-the-loop testing. The NAT2TEST strategy does not provide automatic support for the writing of the necessary test procedures, since they are dependent upon the particular use of the generated test data.

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 favour 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 analysis of whether the system has been properly tested.

5.2. Empirical analyses

We evaluate our work considering examples from the literature, but also from the aerospace and the automotive industries: (i) a vending machine (VM – our running example); (ii) a simplified control system for safety injection in a nuclear power plant (NPP – publicly available [27]), (iii) the priority command function (PC – provided by Embraer); and (iv) part of the turn indicator system of Mercedes vehicles (TIS – publicly available[11]). In the PC, we have a priority command function that decides whether the pilot or copilot side stick will have priority in controlling the airplane. In the TIS, there are two parallel components responsible for controlling the car turn lights, taking into account information such as the position of the turn indicator lever, if the emergency button has been pressed, and the car battery voltage.

Our evaluation considers performance and quality aspects. All data presented here consider multiple executions. Table 4 presents general data about the considered examples. The largest example is the TIS, with 17 system requirements, in a

[11] Turn indicator model: http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/turn_indicator/index_e.html.

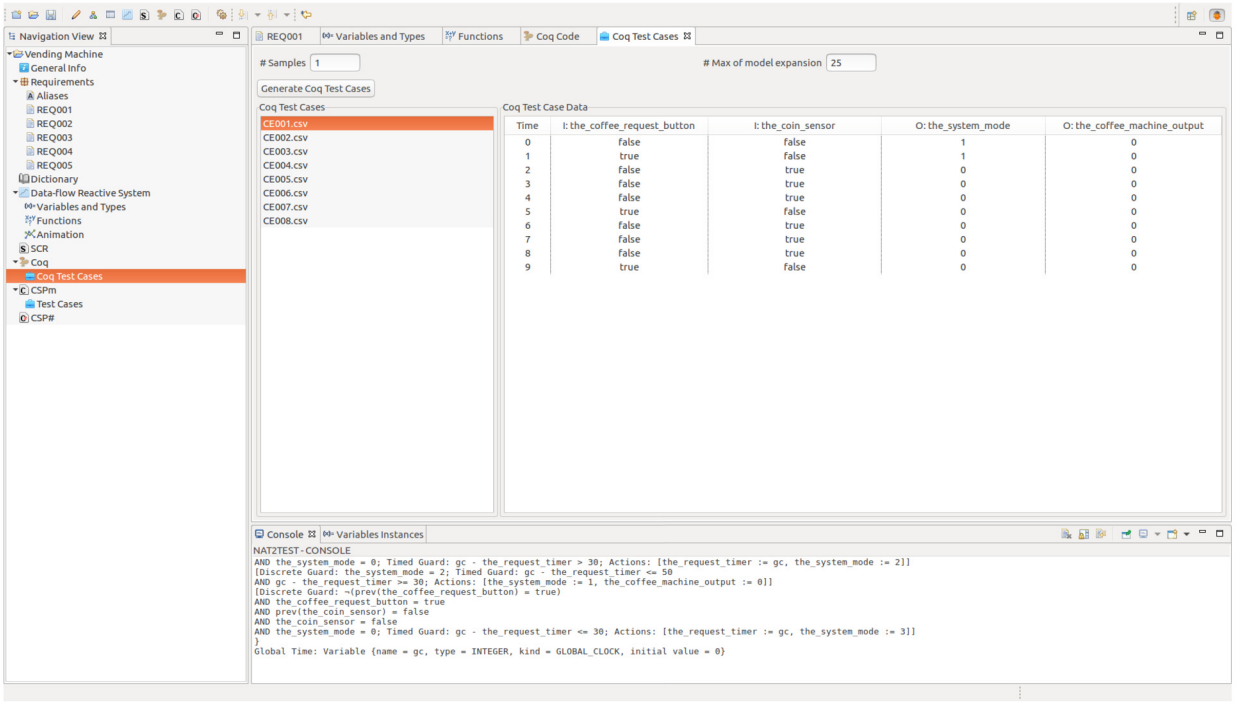


Fig. 11. NAT2TEST tool – test cases generated for the VM.

Table 4
General empirical data.

	VM	NPP	PC	TIS
# of requirements/words:	5/232	11/268	8/294	17/580
# of conditions/actions/TRs:	18/10/130	21/11/149	26/8/162	54/34/406
# of I/O/T:	2/2/1	3/3/0	4/1/0	3/2/1
# of LOC of Coq specification:	382	680	593	1,228
μ # of test cases (1x `Sample`):	9.80 (89.09%)	1.80 (16.36%)	6.60 (60.00%)	5.40 (49.09%)
μ # of test cases (5x `Sample`):	51.00 (92.73%)	10.20 (18.55%)	37.20 (67.64%)	26.60 (48.36%)
μ # of test cases (10x `Sample`):	101.20 (91.17%)	17.60 (15.86%)	67.60 (60.90%)	54.60 (49.19%)

total of 580 words. The smallest one is our running example (VM): 5 system requirements, in a total of 232 words. The number of conditions, actions, and thematic roles automatically identified for each example can also be seen in Table 4.

The system requirements of NPP and PC are not time dependent and, thus, the corresponding s-DFRSs do not have timers. The number of input and output variables are also shown in Table 4. The largest Coq specification is the one derived for the TIS example (near 1,300 LOC), whereas the smallest one is the one generated for our running example (VM – almost 400 LOC). Besides these lines, we have the ones related to our characterisation of DFRSs in Coq (see Section 3), with about 2.5 KLOC; however, these are the same regardless of the example.

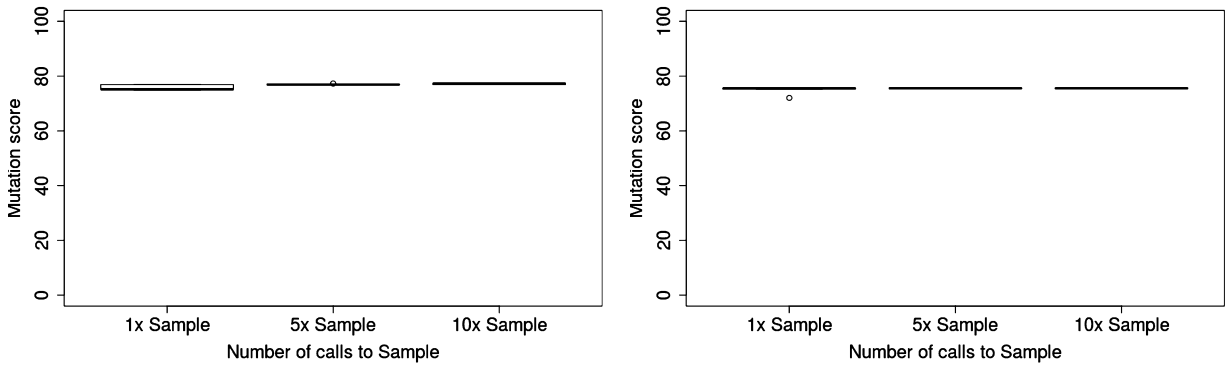
Regarding test generation, we generated (multiple times) three datasets: performing 1, 5, and 10 calls to the QuickChick sampling function, respectively. For instance, considering the VM, we would have 1, 5 and 10 calls of the following command: `Sample (genValidTrace vm_initial_state vm_s_dfrs vm_possibilities 600)`. Each `Sample` call performs 11 calls (the default value of QuickChick) to *genValidTrace* (see Section 4.3). Therefore, each dataset contained 11, 55, and 110 test cases, respectively. The size of each test is bounded to $size = 600$ (up to 600 delay/function transitions).

Some test cases are not valid and, thus, they are discarded: it does not end in a stable state. This happens in two situations. The first one is when the generation stops after performing a delay transition, but before the following function transitions. This test case is not considered since we would have the system input (delay transition), but not the expected system reaction (function transition). The second situation happens when the generation stops after performing a function transition, but before other function transitions. This test case is also not considered since we would have observed only part of the system reaction. Table 4 shows the average number of valid test cases for each dataset (1, 5 and 10 `Sample` calls). A lower percentage of valid test cases is achieved for the NPP and TIS examples, since they have situations where we have a sequence of function transitions, which make more common the occurrence of invalid test cases.

Table 5 presents performance data, considering an i7 @ 2.40GHz x 4, with 8 GB of RAM running Ubuntu 16.04 LTS. The time to generate the Coq specification from the controlled natural-language requirements is marginal (less than 1s even

Table 5
Performance data.

	VM (μ)	NPP (μ)	PC (μ)	TIS (μ)
Syntactic analysis:	0.069s	0.038s	0.080s	0.222s
Semantic analysis:	0.043s	0.239s	0.068s	0.201s
DFRS generation:	0.003s	0.004s	0.003s	0.006s
Coq generation:	0.004s	0.004s	0.007s	0.006s
Compile Coq infra (once):	10.561s	9.789s	9.890s	9.797s
Test generation (1xS):	1.610s	1.664s	1.914s	3.288s
Test generation (5xS):	4.820s	5.726s	6.562s	13.232s
Test generation (10xS):	8.706s	10.614s	13.038s	25.778s



(a) Mutation score for the VM example

(b) Mutation score for the NPP example

Fig. 12. Mutant-based strength analysis (examples from the literature).

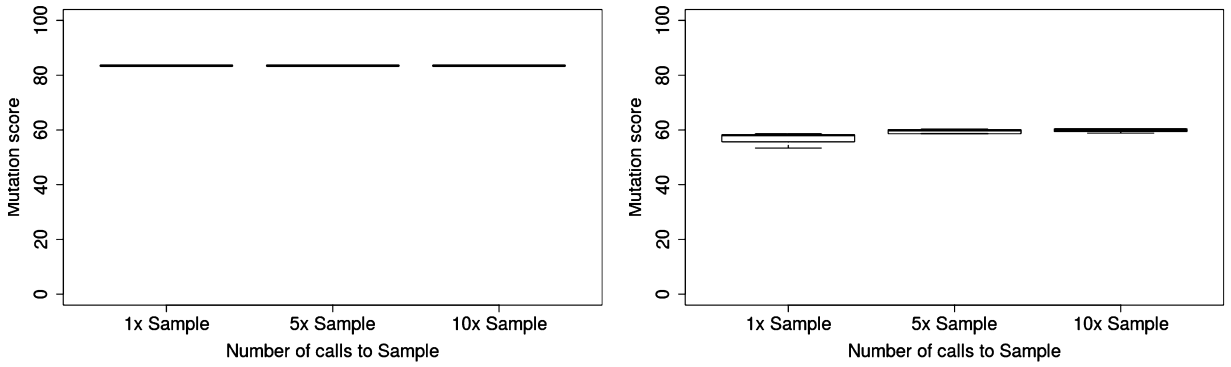
for the most complex example – TIS). The time to compile our Coq infra-structure (particularly, the Coq characterisation of DFRSs, considering definitions or records, well-formedness properties and equivalence proofs of logical and functional definitions) is a more less constant between examples (since this code is not dependent on the considered example) and about 10s. It is important to say that the compilation of this infra-structure is required just once (or when updating the version of the Coq Proof Assistant).

The time to generate test cases is linearly proportional to the number of sampling calls and also relatively small: ranging from 1.61s (1 `Sample` call for the VM example) to 25.77s (10 `Sample` calls for the TIS example). Therefore, the total time required for automatically generating test cases from controlled natural-language requirements would be about 30s for the most complex considered scenario (TIS with 10 `Sample` calls). This data provides a promising argument in favour of the scalability of our approach.

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 [28]. 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 [29], considering all of its default mutant operators; it comprises the typical mutant operators: AOR (arithmetic operator replacement), LCR (logical connector replacement), ROR (relational operator replacement), and ICR (integer constant replacement). We generated 287, 373, 213, and 861 mutants for each considered example (VM, NPP, PC, and TIS, respectively). We wrote C test drivers to run all mutants against all generated datasets. The driver also embeds the test oracle: it provides to the C implementation the input values described by each step of the test case, and then it assesses whether the observed output is precisely the same described by the test step. We are not considering here margin of errors and, thus, the test oracle is trivial. Fig. 12 shows the mutation score (ratio of killed/generated mutants) for the VM and PC. Fig. 13 shows the same data for the NPP and the TIS.

Considering all examples and all datasets, we achieved an average mutation score of about 75%. For the VM, 1, 5 and 10 `Sample` calls lead to a mutation score of 75.80%, 76.99% and 77.13%, respectively. For the NPP, we had a mutation score of 74.78%, 75.54% and 75.54%, respectively. For the PC, we had a mutation score of 83.49% for all datasets (actually, the same value for each replication of each dataset – we comment on this result later). For the TIS, we had a mutation score of 56.80%, 59.53% and 59.84%, respectively (the lowest scores for all examples). As said before, these results are conservative, since we consider by default that all non-killed mutants are not equivalent. By random inspection, we have identified equivalent mutants in all examples. Therefore, we are sure that the true mutation score is actually higher (better) than the ones presented before.



(a) Mutation score for the PC example

(b) Mutation score for the TIS example

Fig. 13. Mutant-based strength analysis (examples from industry).

One interesting result shown by the data is the effect of the number of `Sample` calls on the mutation score. Although some increase is always observed when we increase the number of `Sample` calls, it is not as high as we first expected. By analysing the data, we understood that this happens due to the size (number of test steps) of the generated test cases. Recall that we invoke `Sample` with a bound of $size = 600$, which results in reasonably long test cases (more than 50 steps in many cases). As a consequence, even with a small number of test cases, we are capable of testing many different scenarios and, thus, unveiling many different errors.

Regarding the PC example, we also had test cases written (by hand) by domain specialists. The mutation score achieved by these tests is 81.04% against the score of 83.49% achieved by our testing strategy, which is slightly higher. Concerning this example, we further analysed our results by inspecting manually all non-killed mutants, since the same mutation score (83.49%) was achieved in all runs of the three datasets.

We identified many mutations that lead to equivalent mutants. For instance, we had in the reference code `x != true`, where `x` is a boolean variable. In the mutated code, we had `x < true` (an example of an ROR mutation). Since, `true` is internally represented as 1 and `false` as 0, and the code only assigns one of these two values to `x`, evaluating whether `x != true` (i.e., `x == false` or `x == 0`) is equivalent to check whether `x < true` (i.e., `x < 1` or `x == 0`).

After a careful analysis, we conclude that the actual mutation score, achieved by our testing strategy, for the PC example is of 100%. In other words, all non-equivalent mutants were killed by our automatically generated test cases (even considering just one `Sample` call).

The lowest mutation score was achieved concerning the TIS, which is the most complex considered example. In this situation, the automatically generated test cases should be enhanced by test cases generated with the guide of test purposes (written by domain specialists). This would increase the chances of testing non-covered relevant scenarios, particularly, those associated with the non-killed and non-equivalent mutants.

The main threat to the validity of our analyses is external. The conclusions reached are intrinsically related to the four considered examples. Nevertheless, we believe that the results presented in this section are promising, considering the observed performance, besides being fully automatic.[12]

6. Conclusion

This paper presents a unified and formal framework based on the Coq proof assistant for validating and verifying models of timed data-flow reactive systems, which are automatically derived from controlled natural-language requirements. For all considered examples, both from the literature and the industry, the time required to derive models from requirements is marginal (less than 1 second), and the time required to generate test cases is linearly proportional to the number of sampling calls and also relatively small. This data provides a promising argument in favour of the scalability of our approach.

Well-formedness properties of the models are automatically verified with no user intervention. Proving of general and domain-specific system properties is achieved by developing proof scripts. The generation of test data is supported with the aid of the QuickChick tool. We support two test generation strategies: by sampling random valid traces and guided by user-defined test purposes. The contributions discussed here are integrated into the NAT2TEST tool, which is developed using the Eclipse RCP framework. Our random test generation strategy was evaluated in terms of performance and mutant-based strength analysis, considering examples both from the literature and the industry.

[12] All empirical data and scripts for the VM, NPP and TIS are available online: see Footnote 8. The files regarding the PC example cannot be made publicly available.

Within seconds, test cases were generated automatically from the controlled natural-language requirements, achieving an average mutation score of about 75%. Discarding equivalent mutants, in one of the industrial examples, the actual mutation score is 100%; the generated test cases were capable of detecting all systematically introduced errors.

6.1. Related work

A report by the Federal Aviation Administration (FAA), which discusses practices concerning requirements engineering, states that the overwhelming majority of survey respondents indicated that requirements are being capture as English text [30]. This supports the thesis that, at the very beginning of system development, typically only natural-language requirements are documented. Although this report is some years old, our experience with industrial and academic partners shows that this is still a current practice. Therefore, a central aspect of our research agenda is that we want to enable formal verification from natural-language requirements. Moreover, since it is not realistic (in general) to expect a good knowledge of formal modelling by domain specialists, we also pursue automation whenever possible.

Previous works have already investigated and proposed formal models for describing natural-language requirements and possibly generating test cases. As mentioned in Section 1.2, there is a trade-off between the adoption of controlled natural languages and the degree of user intervention. Since less user intervention is desired by us, as explained before, we compare our work with studies that also adopt some standardisation for writing natural-language requirements and pursue automation.

Unlike our work, the testing strategies proposed in [31–35] consider use case diagrams as specification, which might not be suitable for specifying embedded reactive systems (our focus). In [36], the requirements adhere to a strict *if-then* sentence template, achieving a less flexible (and natural) writing structure than our CNL. In summary, our work distinguishes itself by providing a flexible structure (more natural) to describe timed data-flow reactive systems in controlled natural language, but also achieving automation. We refer to [2] for a more detailed comparison of our work with others concerning the adoption of controlled natural languages.

The adoption of s-DFRS and e-DFRS models to formally represent the behaviour of timed data-flow reactive systems is another central aspect of this work. There are other formal notations that can also be used to model data-flow reactive systems, such as Simulink block diagrams and Lustre code. Verification could be supported by tools such as the Simulink Design Verifier[13] or the SCADE Suite.[14] Moreover, as reported in [37], these representations could even be translated into other notations, enabling the use of other tools (model checkers and theorem provers).

The challenge of this verification perspective is that our strategy is supposed to be used in the very beginning of the project to validate and verify high-level system requirements. Other models, such as Simulink diagrams and Lustre code, are more concrete; they are developed also considering design decisions and other information that is typically not yet available at this stage (project beginning). Actually, our strategy can be used to generate test data, from the high-level system requirements, in order to simulate and to verify other more concrete system models, such as the ones mentioned before.

Another candidate notation would be a timed automaton, which is a classical notation for modelling timed reactive systems. Indeed, one could trace a parallel between timed automata and symbolic DFRSs, and a parallel between timed input-output transition systems and expanded DFRSs. In [2], we show that an expanded DFRS can be represented as a timed input-output transition system. However, it is important to bear in mind that an expanded DFRS is obtained by expanding its symbolic counterpart and, thus, a number of well-formedness properties are enforced since they are part of any valid symbolic DFRS. Therefore, if considering a timed automaton or a timed input-output transition system to directly represent timed data-flow reactive systems, one would need to revise the definitions of these models to take into account these specific properties, since they are not part of the standard theory. We get back to this issue later.

Nevertheless, in the context of using timed automata to represent timed data-flow reactive systems, one could, for instance, use Uppaal to automate the generation of test cases [7]. Besides test generation, Uppaal also supports checking invariant and reachability properties, which is also a goal of our work (system property verification), by means of exploring the state-space of the system. However, in this scenario, one might face scalability issues, when a huge state-space needs to be covered. In [38], the authors construct a verified model checker for timed automata using Isabelle/HOL [39]. Scalability should also be an issue here, since the underlying verification technique is defined in terms of state-space exploration.

In our approach (see Section 4.2), we can rely on induction principles to prove properties with no need to cover the system state-space. A drawback of this approach is that the verification is not automatic; someone needs to develop the proof script. However, if seeking automation, in our work, it would be possible to apply the idea described in Section 4.1 and, then, verify the desired system properties via state-space exploration. However, scalability would be an issue again.

In [40], the authors develop a formalisation in Coq of a special class of timed automata to support the specification, validation and test of critical algorithms. In principle, one could use this formalisation to represent timed data-flow reactive systems. However, we would not gain very much by doing that, and the effort would be comparable to the one carried out

[13] Simulink Design Verifier: https://www.mathworks.com/products/simulink-design-verifier.html.
[14] SCADE Suite: https://www.ansys.com/products/embedded-software/ansys-scade-suite.

here. There are similarities between both representations, such as the notions of variables, states/locations, transitions, and universal/global clock, but they are mostly limited to the structural level of the representation.

It is important to bear in mind that we are representing here a specific class of embedded systems whose inputs and outputs are always available as signals. As a consequence, many concepts are not directly supported in [40], and one would need to encode them; for instance, we have a clear separation between input and output variables, and there is a distinction between delay and function transitions. Furthermore, related to these and other specific concepts, we have many well-formedness properties that are enforced by our representation. System states are partitioned into stable and non-stable ones. A delay transition, whose source is always a stable state, encodes the notion of time evolving and receiving new input signals. Therefore, output variables cannot be updated by delay transitions. Moreover, each delay transition considers all input variables, since they represent signals that are always available. Analogously, a function transition, whose source is always a non-stable state, encodes the notion of system reaction. Therefore, input variables cannot be updated by function transitions. Additionally, time does not evolve when performing such transitions. The collection of all these notions and properties establishes our semantics for timed data-flow reactive systems. Most of our Coq characterisation is about formalising these notions in logical and functional terms, besides proving the equivalence of both formulations. In summary, formalising all these notions in the Coq representation of timed automaton described in [40] would require an effort comparable to the one carried out here, since they are not part of the standard theory of timed automata.

Furthermore, our DFRS models were particularly designed to facilitate their automatic generation from controlled natural-language requirements. If we were to use general purpose notations such as timed automata to capture our controlled natural-language requirements, the translation would be more complicated and costly. This argument also applies to the work developed in [41], where Circus is used to model system-level requirements.

Differently from our work, which focuses on models of system-level requirements, modelling and/or testing timed systems using (interactive) theorem provers is addressed, for example, in [42–45], considering timed connectors, real-time operating systems, programmable logic controllers, and Java code, respectively. Finally, the integration of (interactive) theorem provers and testing strategies is also 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 [46–48]. In general, similar functionalities are provided among these different options.

In summary, the main goal our work is to provide early means for validation, verification and testing, when typically only natural-language requirements are available and, thus, anticipate the discovery of problems. To the best of our knowledge, we are not aware of other unified and formal frameworks for validating, verifying, and testing models of timed data-flow reactive systems, which are automatically derived from controlled natural-language requirements.

6.2. Future work

We envisage the following tasks as future work.

- Consider symbolic time representation. As explained in the end of Section 4.3.1, when expanding an e-DFRS, as part of a validation or of a test generation task, the time step considered by delays transitions needs to be set carefully. In a previous work [12], when representing DFRSs in the CSP process algebra, we proposed a symbolic representation of time. As a consequence, symbolic test cases are generated, with no concrete time information, and then the time delays are made concrete with the aid of an SMT solver. Therefore, there is no need to define a concrete time step. As future work, we should investigate how we could port this result to our Coq characterisation of DFRSs.
- Explore other general and domain-specific system properties. In this work, we explored two general system properties: determinism and absence of time lock. As future work, we should take into account of other general and domain-specific properties; for instance, typical safety and liveness properties.
- Devise proof tactics to automate verification of system properties. In this work, the proof of general system properties is accomplished by developing proof scripts manually. As a future work, we should devise specific proof tactics and/or decision procedures to automate part of these proofs.
- Develop a formal testing theory for DFRSs in Coq. In Section 2.1, we comment on the benefits of defining a formal testing strategy, in particular, proving that the strategy is sound: 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. As a future work, we should pursue the definition of the missing parts that would support a formal testing strategy for our Coq characterisation of DFRSs, namely: an implementation relation, a test execution procedure, besides taking into account the details on how QuickChick performs property-based testing.
- Provide support in Coq for the common phases of the NAT2TEST strategy. The first three phases of the NAT2TEST strategy (syntactic and semantic analyses, besides generation of DFRSs from requirement frames) are implemented as Java programs. As future work, we should try to specify them in Coq. In more details, to define in Coq: (i) a Generalised LR (GLR [49]) parser for our CNL, (ii) the process that derives requirement frames from the abstract syntax trees yielded by the parser, and (iii) the generator of DFRSs from requirement frames. In such a situation, we would no longer need to translate the informal representation of DFRSs in Java to the formal Coq characterisation, since the generator of DFRSs (defined in Coq) would already consider the formal characterisation of DFRSs.

Declaration of competing interest

Gustavo Carvalho: The following researchers also work at my institution (UFPE) or I have close collaboration with them: Alexandre Mota, UFPE, Brazil Augusto Sampaio, UFPE, Brazil Juliano Iyoda, UFPE, Brazil Marcio Cornelio, UFPE, Brazil Sidney Nogueira, UFRPE, Brazil I also have some informal collaboration with: Marcel Oliveira, UFRN, Brazil Thierry Lecomte, CLEARSY Systems Engineering, France.

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Brasil (CAPES) – Finance Code 001. It has also been partially supported by Centro de Informática (CIn) at Universidade Federal de Pernambuco (UFPE).

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, Form. Asp. Comput. 28 (5) (2016) 725–765, https://doi.org/10.1007/s00165-016-0387-x.
[3] 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.
[4] 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.
[5] A.W. Roscoe, Understanding Concurrent Systems, Springer, 2010.
[6] Y. Bertot, P. Castran, Interactive Theorem Proving and Program Development: Coq'Art the Calculus of Inductive Constructions, 1st edition, Springer Publishing Company, Incorporated, 2010.
[7] K.G. Larsen, M. Mikucionis, B. Nielsen, Online Testing of Real-Time Systems Using Uppaal, Springer, Berlin Heidelberg, 2005, pp. 79–94.
[8] C.J. Fillmore, The case for case, in: Harms Bach (Ed.), Universals in Linguistic Theory, Holt, Rinehart, and Winston, New York, 1968, pp. 1–88.
[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, Sci. Comput. Program. 95 (Part 3) (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] 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.
[13] K. Jensen, L. Kristensen, Coloured petri nets: modelling and validation of concurrent systems, https://doi.org/10.1007/b95112, 2009.
[14] B. Cesar, F. Silva, G. Carvalho, A. Sampaio, CPN simulation-based test case generation from controlled natural-language requirements, Sci. Comput. Program. 181 (2019) 111–139, https://doi.org/10.1016/j.scico.2019.04.001.
[15] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A.W. Roscoe, FDR3 — a modern refinement checker for CSP, in: E. Ábrahám, K. Havelund (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 187–201.
[16] L. de Moura, N. Bjørner, Z3: an efficient smt solver, in: C.R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 337–340.
[17] M.-C. Gaudel, Testing can be formal, too, in: P.D. Mosses, M. Nielsen, M.I. Schwartzbach (Eds.), TAPSOFT'95: Theory and Practice of Software Development, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 82–96.
[18] 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.
[19] 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.
[20] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NUSMV: a new symbolic model checker, Int. J. Softw. Tools Technol. Transf. 2 (4) (2000) 410–425, https://doi.org/10.1007/s100090050046.
[21] 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.
[22] 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.
[23] 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.
[24] P. Ammann, J. Offutt, Introduction to Software Testing, 2nd edition, Cambridge University Press, USA, 2016.
[25] J.W. Duran, S.C. Ntafos, An evaluation of random testing, IEEE Trans. Softw. Eng. 10 (4) (1984) 438–444, https://doi.org/10.1109/TSE.1984.5010257.
[26] D. Hamlet, R. Taylor, Partition testing does not inspire confidence (program testing), IEEE Trans. Softw. Eng. 16 (12) (1990) 1402–1411, https://doi.org/10.1109/32.62448.
[27] E. Leonard, C. Heitmeyer, Program synthesis from formal requirements specifications using APTS, High.-Order Symb. Comput. 16 (2003) 63–92.
[28] 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.
[29] 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.
[30] F.A.A. Requirements, Engineering Management Findings Report, Tech. Rep, U.S. Department of Transportation - Federal Aviation Administration, USA, 2009.
[31] A. Sinha, S. Sutton Jr, A. Paradkar, Text2Test: automated inspection of natural language use cases, in: Proceedings of the Third International Conference on Software Testing, Verification and Validation, 2010, pp. 155–164.
[32] F.A. Barros, L. Neves, E. Hori, D. Torres, The ucsCNL: a controlled natural language for use case specifications, in: Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering, 2011, pp. 250–253.

[33] S. Nogueira, A. Sampaio, A. Mota, Test generation from state based use case models, Form. Asp. Comput. 26 (3) (2014) 441–490.
[34] T. Yue, S. Ali, L. Briand, Automated transition from use cases to uml state machines to support state-based testing, in: R.B. France, J.M. Kuester, B. Bordbar, R.F. Paige (Eds.), Modelling Foundations and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 115–131.
[35] S.S. Somé, Supporting use case based requirements engineering, Inf. Softw. Technol. 48 (1) (2006) 43–58.
[36] M. Esser, P. Struss, Obtaining models for test generation from natural-language like functional specifications, in: Proceedings of International Workshop on Principles of Diagnosis, 2007, pp. 75–82.
[37] S.P. Miller, M.W. Whalen, D.D. Cofer, Software model checking takes off, Commun. ACM 53 (2) (2010) 58–64, https://doi.org/10.1145/1646353.1646372.
[38] S. Wimmer, P. Lammich, Verified model checking of timed automata, in: D. Beyer, M. Huisman (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham, 2018, pp. 61–78.
[39] T. Nipkow, M. Wenzel, L.C. Paulson, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Springer-Verlag, Berlin, Heidelberg, 2002.
[40] 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.
[41] 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.
[42] 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.
[43] 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.
[44] 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.
[45] W. Ahrendt, C. Gladisch, M. Herda, Proof-Based Test Case Generation, Springer International Publishing, Cham, 2016, pp. 415–451.
[46] 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.
[47] A.D. Brucker, B. Wolff, On theorem prover-based testing, Form. Asp. Comput. 25 (5) (2013) 683–721.
[48] 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.
[49] M. Tomita, Efficient Parsing for Natural Language, Kluwer Academic Publishers, 1986.