Pós-Graduação em Ciência da Computação

Igor de Araújo Meira

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

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2020

Igor de Araújo Meira

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

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software
Orientador: Gustavo Henrique Porto de Carvalho

Recife
2020

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

M514v Meira, Igor de Araújo
Validating, verifying and testing timed data-flow reactive systems in Coq from controlled natural-language requirements / Igor de Araújo Meira. – 2020.
64 f.: il., fig., tab.

Orientador: Gustavo Henrique Porto de Carvalho.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências.

1. Engenharia de software. 2. Linguagem natural controlada. I. Carvalho, Gustavo Henrique Porto de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2020 - 68

Igor de Araújo Meira

Validating, Verifying and Testing Timed Data-Flow Reactive Systems in Coq from Controlled Natural-Language Requirements

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 05/03/2020.

ACKNOWLEDGEMENTS

The results presented in this work were obtained with the support of several people, therefore, I would like to thank everyone who contributed so that this moment has become reality. I begin my thanks to the Informatics Center, at Federal University of Pernambuco, for the incredible structure, and for the knowledge I acquired during my time as a student. I also thank professors Augusto Sampaio and Alexandre Mota, for the incredible classes taught.

A special thanks to my advisor, Gustavo Carvalho, who helped me whenever I needed, guided me during my trajectory, for believing in my potential and for the valuable lessons, in the classroom, at meetings and even about aviation. I will be forever grateful for everything.

I would like to thank my aunt Alexandra and my uncle César, for all the conversations, for all the advice and unique moments. I also thank all my friends, who were always by my side when I needed it. To my neighbours in Gravatá, who always lent their internet.

Finally, all the gratitude to my parents (Gibson Meira and Nelma Meira), for giving me all the support and always ensuring that I had everything to complete this stage of life. To André Meira (my lovely brother and best friend), for all the fun and advices, and Bruna Lisboa (my girlfriend and life partner), for all her patience, support and love.

ABSTRACT

The NAT2TEST strategy provides means for generating test cases from controlled natural-language requirements. It is tailored for testing timed data-flow reactive systems (DFRSs), which are a class of embedded systems whose inputs and outputs are always available as signals. Input signals can be seen as data provided by sensors, whereas the output data are provided to system actuators. In previous works, verifying well-formedness properties of DFRS models was accomplished in a programmatic way, with no formal guarantees, and test cases were generated by translating theses models into other notations. Here, we use Coq as a single framework to specify, 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, and test generation is achieved with the aid of the QuickChick tool. Our Coq-based testing strategy was integrated into the NAT2TEST tool, which is a multi-platform tool written in Java, using the Eclipse RCP framework. 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%. 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.

Keywords: NAT2TEST strategy. Controlled natural language. Timed data-flow reactive system. Coq. Property-based testing. QuickChick.

RESUMO

A estratégia NAT2TEST permite gerar casos de testes a partir de requisitos em linguagem natural controlada. Esta estratégia se destina ao teste de sistemas reativos baseados em fluxos de dados (DFRSs), uma classe de sistemas embarcados cujas entradas e saídas estão sempre disponíveis como sinais. Sinais de entrada podem ser vistos como dados providos pelos sensores, enquanto que dados de saída são encaminhados a atuadores do sistema. Em trabalhos anteriores, a verificação de propriedades de boa formação de modelos DFRS era realizada de forma programática, sem garantias formais, e casos de testes eram gerados traduzindo estes modelos em outras notações. Aqui, faz-se uso de Coq como um ambiente único para especificar, validar e verificar modelos DFRS. Adicionalmente, a especificação de DFRSs em Coq é gerada automaticamente a partir de requisitos em linguagem natural controlada, e propriedades de boa formação são formalmente verificadas sem intervenção do usuário. A validação do sistema é suportada através da exploração controlada de modelos, e testes são gerados com o apoio da ferramenta QuickChick. A estratégia baseada em Coq desenvolvida neste trabalho foi integrada à ferramenta NAT2TEST, que é uma ferramenta multiplataforma escrita em Java, usando o ambiente Eclipse RCP. Considerando exemplos tanto da literatura, como também da indústria aeroespacial (Embraer) e automotiva (Mercedes), a estratégia de testes proposta aqui foi avaliada em termos de desempenho e de habilidade em detectar defeitos gerados por mutação. Em poucos segundos, casos de testes foram gerados automaticamente a partir dos requisitos, alcançando uma taxa de detecção de mutantes de cerca de 75%. Descartando mutantes equivalentes, em um dos exemplos industriais, a taxa de detecção real é de 100%; os casos de testes gerados foram capazes de detectar todos os erros introduzidos sistematicamente.

Palavras-chaves: Estratégia NAT2TEST. Linguagem natural controlada. Sistemas reativos baseados em fluxos de dados. Coq. Testes baseados em propriedades. QuickChick.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Today's society, given its enormous dependence on computer programs, can be seen as largely dependent on, or even defined by, software. Failures in programs that support people's daily lives can have significant impacts, causing financial losses or even endangering people's lives. Therefore, it is necessary to invest in strategies that seek to increase the quality of the developed software.

One possibility is to promote error detection and subsequent correction through testing policies. However, when tests are written and run manually, they are not error-free either, which can compromise their ability to find real defects. In this scenario, it is important to invest in strategies that can automatically generate and even perform tests, ideally error free.

1.1 MODEL-BASED TESTING

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

Here, we focus on models of timed Data-Flow Reactive Systems (DFRSs): a class of embedded systems whose inputs and outputs are always available as signals. Additionally, the system behaviour might be dependent on time. Models of DFRSs are fully explained in (CARVALHO; CAVALCANTI; SAMPAIO, 2016). They have been used to model examples both from the literature and the industry. This previous work shows that these models can be seen as timed input-output transition systems, but, being more abstract, enable them to be automatically extracted from system-level specifications described in a controlled natural language, which is an important aspect as discussed in what follows.

Despite the benefits of MBT, those who are not familiar with the models syntax and semantics may be reluctant to adopt theses formalisms. 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

The demand of 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, a specialist (usually mathematicians, logicians, computer engineers and scientists) is required when such languages, and their corresponding techniques, are used in business contexts. Furthermore, most of these models are not yet available in the very beginning of the system development project.

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

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 Natural Languages (CNLs). The works that adopt the former approach usually depend on a higher level of user intervention to derive models and to generate test cases. Differently, the restrictions imposed by a CNL might allow a more automatic approach when generating models and test cases. Ideally, a compromise between these two possibilities should be sought to provide a useful degree of automation along with a natural-language specification feasible to be used in practice.

1.3 THE NAT2TEST$_{Coq}$ STRATEGY

NAT2TEST (CARVALHO et al., 2015) is a fully automatic strategy for generating test cases from natural language to timed data-flow reactive systems. Seeking for automation, it adopts a CNL for describing the system requirements. In (CARVALHO; CAVALCANTI; SAMPAIO, 2016), it is explained how models of DFRSs can be automatically derived from natural-language requirements adhering to SysReq-CNL, a CNL specially designed for editing requirements of timed data-flow reactive systems. Test generating is achieved by reusing different notations and techniques.

In our 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 inconsistency between different requirements). Additionally, to generate test cases it is necessary to translate DFRS models into other less abstract notations.

In this work (CARVALHO; MEIRA, 2019), we extend the NAT2TEST strategy, using the Coq proof assistant (BERTOT; CASTRAN, 2010) as a single framework to validate and to verify DFRS models, besides 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. Test generation is achieved with the aid of the QuickChick tool (PARASKEVOPOULOU et al., 2015), 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) industry, 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%. 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 in Coq for validating, verifying, and testing timed data-flow reactive systems from controlled natural-language requirements. In more details, we have:

- A Coq characterisation of symbolic and expanded data-flow reactive systems;
- An automatic mechanism for proving well-formedness properties;
- System validation via bounded exploration of models 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.

1.4 DISSERTATION STRUCTURE

The remainder of this work is structured as follows.

- Chapter 2 discusses the main concepts related to this work: the NAT2TEST strategy, data-flow reactive systems, the Coq proof assistant, and property-based testing.
- Chapter 3 presents our characterisation of data-flow reactive systems in Coq. Moreover, it shows how our Coq characterisation can be used to validate system specifications, besides generating test cases with the aid of the QuickChick tool.
- Chapter 4 explains how our work was integrated into the NAT2TEST tool. In this chapter, we also detail our empirical analyses.
- Chapter 5 concludes by discussing related and future work.

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

2 BACKGROUND

Here, 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 an automatic strategy for generating test cases for timed data-flow reactive systems from controlled natural-language requirements, considering different internal and hidden formalisms (see Figure 1). In Section 2.1.4, we discuss the advantages and limitations related to each possible internal formalism, besides explaining why it is worth proposing a a new possibility based on Coq.

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 other phases of the process depend on internal formalisms adopted.

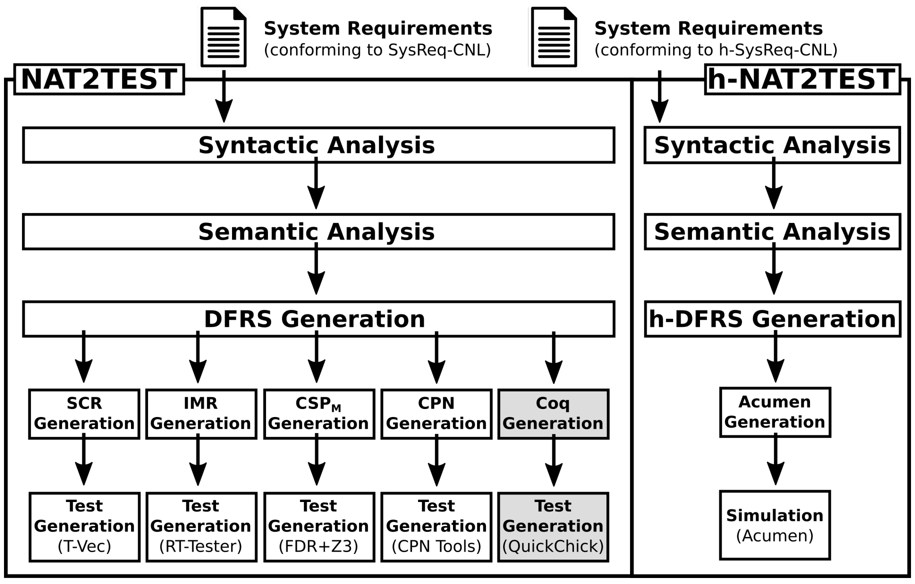


Figure 1 – NAT2TEST: a strategy for generating test cases

In what follows, we describe the first three phases of the NAT2TEST strategy, besides explaining in general terms its current support for test generation, along with some extensions that have been previously developed. It is important to emphasise that the focus of this work is the Coq-based extension (NAT2TEST$_{Coq}$), highlighted in Figure 1.

2.1.1 Syntactic analysis

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 (LARSEN; MIKUCIONIS; NIELSEN, 2005)). 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 requirement (REQ001) 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.*

2.1.2 Semantic analysis

In the second phase, the requirements are semantically analysed using the case grammar theory (FILLMORE, 1968). 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 the 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.

2.1.3 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 thematic roles are inspected to identify system variables (inputs, outputs and timers). For instance, since the *Patient* denotes an entity who is affected by some action (its value is changed), it will be considered when identifying output variables. Entities whose value are only accessed will be considered when identifying input variables. Timers are identified with the aid of the auxiliary keyword *timer*.

After identifying system variables, the system behaviour is encoded as assignments (derived from actions) guarded by conditions (derived from conditions). For a compre-

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*		

hensive explanation of how DFRS models are derived from requirement frames, we refer to (CARVALHO; CAVALCANTI; SAMPAIO, 2016).

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 Section 2.2.

2.1.4 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 (HENINGER et al., 1978) (more details in (CARVALHO et al., 2014));
- NAT2TEST$_{IMR}$: based on *Internal Model Representation* – IMR (PELESKA et al., 2011) (more details in (CARVALHO et al., 2014));
- NAT2TEST$_{CSP}$: based on *Communicating Sequential Processes* – CSP (ROSCOE, 2010) (more details in (CARVALHO; SAMPAIO; MOTA, 2013));
- NAT2TEST$_{CPN}$: based on *Coloured Petri Nets* – CPN (JENSEN; KRISTENSEN, 2009) (more details in (SILVA; CARVALHO; SAMPAIO, 2019));
- NAT2TEST$_{Coq}$: based on *Coq* (BERTOT; CASTRAN, 2010) – the main contribution of this work.

Exploring different formal notations allows test generation from several perspectives, using different languages, tools, and techniques. In NAT2TEST$_{SCR}$ and NAT2TEST$_{IMR}$, test cases are generated with the support of commercial testing tools: T-VEC[1] and RT-Tester[2], respectively. Differently, the NAT2TEST$_{CSP}$ strategy reuses a general purpose refinement checker (FDR[3] (GIBSON-ROBINSON et al., 2014)) and SMT solver (Z3[4] (MOURA; BJØRNER, 2008)) to deliver a generation supported by 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 (internal formalism derived from DFRS models), tools and techniques considered by the aforementioned specialisations in order to generate test cases.

Table 2 – Specialisations of the NAT2TEST strategy

	Language	**Technique**	**Tool**
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

Each specialisation has its own advantages, summarised as follows:

- NAT2TEST$_{SCR}$: scalability, test coverage criteria;
- NAT2TEST$_{IMR}$: scalability, test coverage criteria;
- NAT2TEST$_{CSP}$: formal testing theory, symbolic time representation, free software[5];
- NAT2TEST$_{CPN}$: scalability, perspective of formal testing theory, free software[6];
- NAT2TEST$_{Coq}$: scalability, single framework for validation, verification and testing, free software[7].

The first two specialisations (NAT2TEST$_{SCR}$ and NAT2TEST$_{IMR}$) are supported by commercial testing tools, with algorithms designed and optimised for testing generation. Important coverage criteria are supported when generating test cases, for instance

[1] T-VEC website: <https://www.t-vec.com/>
[2] RT-Tester website: <https://www.verified.de/products/rt-tester/>
[3] FDR website: <https://www.cs.ox.ac.uk/projects/fdr/>
[4] Z3 website: <https://github.com/Z3Prover/z3>
[5] Z3 is released under MIT license. FDR is freely available for academic teaching and research purposes.
[6] CPN Tools is released under GNU General Public License (GPL) version 2.
[7] Coq is released under GNU Lesser General Public License v2.1.

MC/DC, which is required by standards such as DO-178C (JACKLIN, 2012). However, the testing theory is not formal (GAUDEL, 1995) 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. This theory differs from the one defined in (CAVALCANTI; GAUDEL, 2007), since it considers a clear separation between input and output events, besides supporting partial specifications and 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.

Finally, the NAT2TEST$_{Coq}$ specialisation, proposed here, distinguishes itself by creating a scalable, unified and formal framework for validating, verifying and testing timed data-flow reactive systems; with no need to further translate our Coq characterisation of DFRSs into other notations. 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.

2.1.5 Other extensions

In (SANTOS; CARVALHO; SAMPAIO, 2018), the SysReq-CNL is extended 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 (BARZA et al., 2016), it is possible to specify in natural language 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 (CIMATTI et al., 2000), 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 (OLIVEIRA et al., 2017), it is discussed 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, this work revisits each phase of the NAT2TEST strategy, now considering 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 (TAHA et al., 2016), 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, while outputs are data provided to actuators. DFRSs may also have internal timers, which are used to trigger time-based behaviour. There are two models of DFRSs: symbolic DFRS (s-DFRS) and expanded DFRS (e-DFRS). Basically, the former comprises an initial state, along with functions that describe the system behaviour (how the system evolves, depending on the input it receives). Differently, an e-DFRS represents the system behaviour as a state-based machine. It can be seen as an expansion of the symbolic model, 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 ($button$). 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 communicated values reflect the system's possible modes ($choice \mapsto 0$, $idle \mapsto 1$, $strong\ coffee \mapsto 2$, and $weak\ coffee \mapsto 3$) and the possible outputs ($strong \mapsto 0$, and $weak \mapsto 1$). The VM has just one timer: the $request$ timer, which is used to register the moments when a coin is inserted, when the coffee request button is pressed, and when the coffee is produced.

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

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

An s-DFRS is a 6-tuple: (I, O, T, $gcvar$, s_0, F). Inputs (I) and outputs (O) are system variables, whereas timers (T) are used to model temporal behaviour. The global clock is $gcvar$, a variable whose values are non-negative numbers. The initial state is s_0, and F is

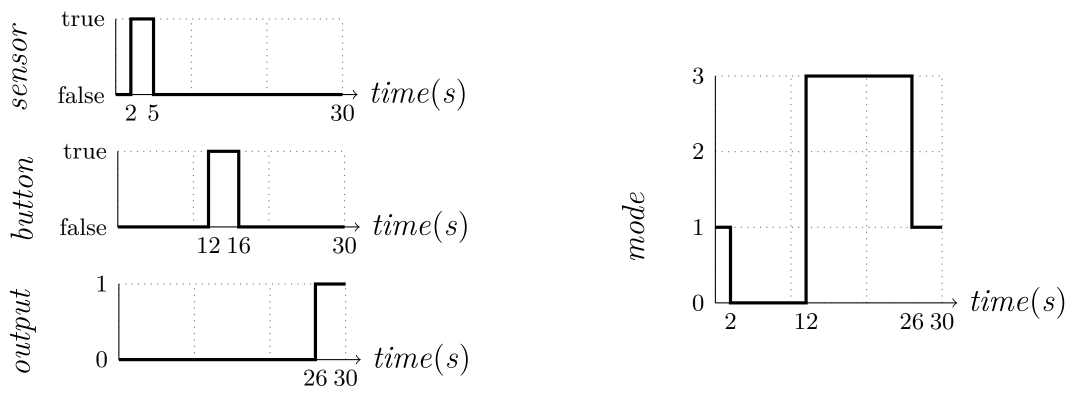


Figure 2 – Example of signals for the vending machine

a set of functions describing the system behaviour. For the VM, we have the following s-DFRS:

- $I = \{sensor, button\}$;
- $O = \{mode, output\}$;
- $T = \{timer\}$; – an internal variable
- *gcvar* denotes the system global clock
- $s_0 = \{sensor = false, button = false, mode = 1, output = 0, timer = 0, gcvar = 0\}$;
- $F = \{...\}$ – detailed later

An e-DFRS differs from the symbolic one as it encodes the system behaviour as a state-based machine, whereas an s-DFRS does that symbolically via definitions of functions. An e-DFRS represents a timed system with continuous or discrete behaviour modelled as a state-based machine. States are obtained from an s-DFRS by applying its functions to non-stable states (when a system reaction is expected), but also letting the time evolve.

Therefore, an e-DFRS is a 7-tuple: (I, O, T, $gcvar$, s_0, S, TR), where TR is a transition relation associating states in S by means of delay and function transitions. A delay transition represents the observation of the input signals' values after a given delay, whereas the function transition represents how the system reacts to the input signals: the observed values of the output signals. The transitions are encoded as assignments to input and output variables as well as timers.

Considering the example presented in Figure 2, Figure 3 shows some states of the e-DFRS representation for the vending machine. The initial state considers the initial value of all system variables. The delay transition (D) 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 (F), updating the system mode, besides resetting the request timer. Here, the reset operation is represented as assigning to the timer the current system global clock. This prevents us from updating all timers whenever time evolves. The function transition happens instantaneously (time does not evolve).

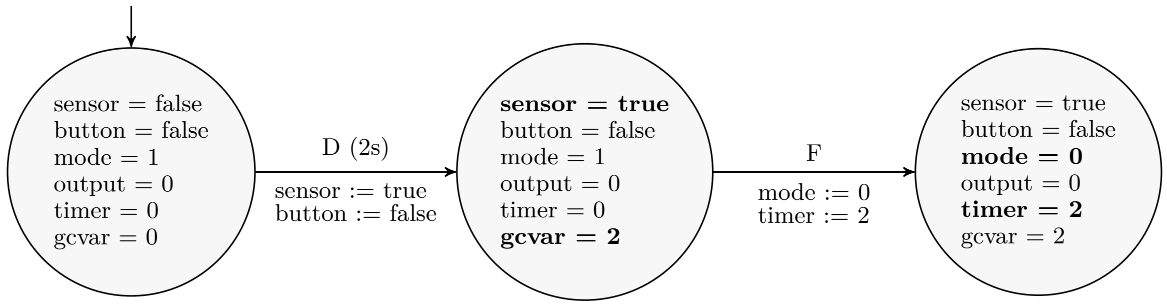


Figure 3 – Some states of the e-DFRS representation for the VM

As said before, we refer to (CARVALHO; CAVALCANTI; SAMPAIO, 2016) 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 (BERTOT; CASTRAN, 2010), which is used later in this work.

Coq[8] employs a functional language (Gallina), which is similar to Haskell, to describe algorithms. Computer-verified proofs are developed interactively using tactics[9], which have some support for automation via the tactics language (Ltac). As a logic system, Coq considers a higher-order logic. In what follows, we briefly address these three topics (Gallina, tactics, and automation). Moreover, we also explain the benefits and limitations of functional, logic, and inductive definitions in Coq.

2.3.1 The Gallina language

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

[8] Coq website: <https://coq.inria.fr>
[9] Index of built-in tactics: <https://coq.inria.fr/refman/coq-tacindex.html>

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

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

```
Fixpoint length {T : Type} (l : list T) : nat :=
  match l with
  | [] ⇒ 0
  | h :: tl ⇒ 1 + length tl
  end.
```

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

2.3.2 Building proofs with tactics

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

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

The tactics employed modifies the proof goal in order to demonstrate its truth. Table 3 shows how the proof goal evolves after processing each tactic. The command `Proof.`

starts the proof environment, loading the proof goal. After that, `intros.` performs universal instantiation, in order to prove the goal for arbitrary values of `T`, `l1`, and `l2`. The command `induction l1.` performs induction on *l1*. This creates two subgoals: base case, and inductive step. The symbol `-` (optional) tells Coq that we now focus on the next subgoal (base case).

Table 3 – Proof of theorem *length_app*

Command	**Proof state**
`Proof.`	`forall (T : Type) (l1 l2 : list T),` `length (app l1 l2) = length l1 + length l2`
`intros.`	`length (app l1 l2) = length l1 + length l2`
`induction l1.`	`____________________(1/2)` `length (app [] l2) = length [] + length l2` `____________________(2/2)` `length (app (a :: l1) l2) = length (a :: l1) + length l2`
`-`	`length (app [] l2) = length [] + length l2`
`simpl.`	`length l2 = length l2`
`reflexivity.`	`This subproof is complete`
`-`	`IHl1 : length (app l1 l2) = length l1 + length l2` `____________________` `length (app (a :: l1) l2) = length (a :: l1) + length l2`
`simpl.`	`IHl1 : length (app l1 l2) = length l1 + length l2` `____________________` `S (length (app l1 l2)) = S (length l1 + length l2)`
`rewrite IHl1.`	`IHl1 : length (app l1 l2) = length l1 + length l2` `____________________` `S (length l1 + length l2) = S (length l1 + length l2)`
`reflexivity.`	`No more subgoals.`
`Qed.`	`length_app is defined`

After simplifying the proof goal of the base case (`simpl.`), we are left to prove that `length l2 = length l2`, which is trivially true (proved by `reflexivity.`). In the inductive step, assuming the hypothesis `IHl1 : length (app l1 l2) = length l1 + length l2`, we need to prove that `length (app (a :: l1) l2) = length (a :: l1) + length l2` holds. After simplification (`simpl.`), the goal becomes: `S (length (app l1 l2)) = S (length l1 + length l2)`, where S stands for the "successor" constructor. By rewriting the goal considering *IHl1* (`rewrite IHl1.`) we get `S (length l1 + length l2) = S (length l1 + length l2)`, provable by `reflexivity.`. The command `Qed.` finishes the prove. Each command is verified by Coq, and it can only be applied if the underlying premises for its application are satisfied. This ensures the construction of a computer-verifiable proof.

2.3.3 Proof automation

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

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

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

```
Ltac trivial_hypo :=
  try (simpl ; intro H ; reflexivity).
Ltac absurd_hypo :=
  try (simpl ; intro H ; inversion H).
Theorem empty_length_0' :
  ∀ (T : Type) (l : list T),
    is_empty l ↔ length l = 0.
Proof.
  intros. destruct l ; split ;
  repeat (trivial_hypo ; absurd_hypo).
```

```
Qed.
```

In our second try (*empty_length_0'*), we define two tactics (*trivial_hypo*, and *absurd_hypo*), which tries to prove the current goal by reflexivity or contradiction, respectively. Then, the theorem is proved by trying to apply these two tactics many times (`repeat`). The command `;` applies the following commands to all subgoals, and not only to the next one.

2.3.4 Functional, logical, and inductive characterisations

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

```
Fixpoint evenb (n : nat) : bool :=
  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, particularly when proving well-formedness properties, we favour their functional characterisation to enable automatic proof of model consistency.

2.4 PROPERTY-BASED TESTING

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

Property-based testing, famous in the functional world due to the QuickCheck framework for Haskell (CLAESSEN; HUGHES, 2000), 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[10] 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. In Chapter 3, when describing our test generation strategy, we give move details about QuickChick.

[10] QuickCheck: <http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html>

3 VALIDATING, VERIFYING AND TESTING DFRSS IN COQ

In (CARVALHO; CAVALCANTI; SAMPAIO, 2016), models of data-flow reactive systems, along with well-formedness properties, are formalised in Z. In this previous work, it is also proposed 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. This verification is neither sound nor complete; the algorithms are not proved to reflect precisely all formal conditions in Z.

Here, we take one step further and formalise DFRS models using Coq. This chapter has two main parts. First, we detail our Coq characterisation of symbolic and expanded DFRSs (Section 3.1). Afterwards, we show how this characterisation can be used to validate system requirements and generate test cases (Section 3.2).

3.1 CHARACTERISATION OF DFRSS IN COQ

The general architecture of our characterisation of DFRSs in Coq[1] is presented in Figure 4. The folders *variables*, *states*, and *functions* define the constituent elements of a symbolic DFRS (*s_dfrs*). The folder *e_dfrs* contains the definitions of an expanded DFRS, which is built upon the symbolic one and the definition of a transition relation (*trans_rel*). Each folder has a main *.v file (named after the folder's name), which provides a logical characterisation. There are two other files: *_fun_rules.v (a functional characterisation), and *_fun_ind_equiv.v (proving that both characterisations are equivalent). 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. Part of these functions are used by our test generation module (defined in *quickchick*). Finally, the examples considered in this work are defined in *examples*. In what follows, we detail our Coq characterisation of DFRSs. In terms of Lines of Code (LOC), our Coq characterisation has about 2.3 KLOC of definitions and 1.5 KLOC of proof scripts.

Sections 3.1.1 and 3.1.2 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 3.1.3, this verification is performed with no need for user interaction; it is completely automatic.

[1] Available online in: <https://github.com/igormeira/DFRScoq>

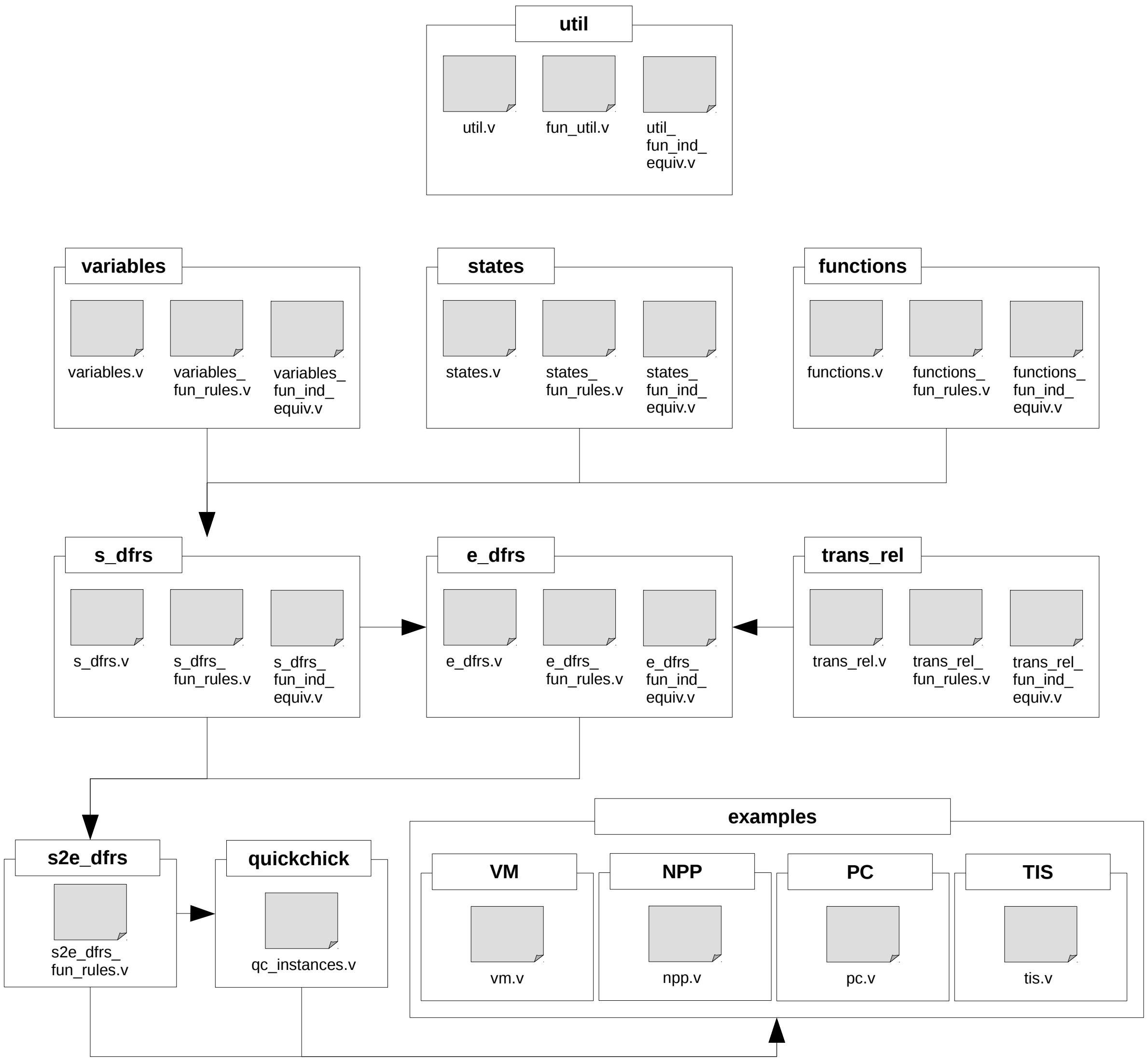


Figure 4 – Characterisation of DFRSs in Coq

3.1.1 Coq characterisation of symbolic DFRSs

An s_DFRS is a 6-tuple: $(I, O, T, gcvar, s_0, F)$, comprising (input, output, and timer) variables, 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.

3.1.1.1 Variables

In our Z characterisation of variables, let $NAME$ be a given type and gc be the name of the system global clock, a variable name ($VNAME$) is defined as any element of $NAME$ except by the value of gc (CARVALHO; CAVALCANTI; SAMPAIO, 2016).

$$[NAME]$$
$$gc : NAME$$
$$VNAME == NAME \setminus \{gc\}$$

In this work, we rewrite this definition in Coq as follows: *NAME* is defined as any string, and *gc* is defined as the string "gc". Then, we define *ind_rules_vname* as a propositional function (i.e., functions that build predicates from the given arguments) that, given a *vname : String*, yields the proposition that is true if, and only if, the value of *vname* is different from the value of the variable *gc* (*string_dec* is an auxiliary propositional function that compares strings). Finally, we define a variable name as any instance of the record *VNAME*.

```
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
}.
```

It is important to note the usage of `Record`. In Coq, differently from typical programming languages, a record might comprise data values (*vname* – the value of the variable name), but also properties (predicates) that need to hold (*rules_vname*). For the *VNAME* record, the property *rules_vname* enforces that the variable name (*vname*) is different from "gc". Therefore, when creating an instance of *VNAME* (i.e., declaring the name of a variable), it is necessary to prove that the instance is well-formed (the name of the variable is not "gc", since this name is reserved for the variable *gc*). This feature brings cohesion between structural definition and well-formedness properties.

In our Z characterisation of variables, system variables (*SVARS*), which might represent input or output variables, are 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) (CARVALHO; CAVALCANTI; SAMPAIO, 2016). Here, these types[2] are sufficient since we are dealing with embedded systems, where variables denote signals.

$$TYPE ::= bool \mid int \mid nat \mid float \mid ufloat$$
$$SVARS == \{f : VNAME \mathrel{+\!\!\!\!\twoheadrightarrow} TYPE \mid f \neq \emptyset \land \operatorname{ran} f \subseteq \{bool, int, float\}\}$$

In Coq, system variables (*SVARS*) are defined as a list of pairs of names and types (*TYPE*). All Coq functions are total and, thus, we represent partial functions as lists in conjunction with the required properties that guarantee that the lists are indeed representations of partial functions.

[2] In Z, there is default support for boolean, integer and natural values.

```
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
}.
```

Concerning the record *SVARS*, we have that *svars* needs to be a non-empty function of variable names to types (no name repetition, and each name is mapped to a single type). Besides that, we need to have type consistency. These properties are defined with the aid of 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 should be compared.

In this work, we restrict ourselves to boolean, integer and natural values. We consider that supporting floating-point numbers remains as a future work. Nevertheless, this restriction has not prevented us from considering interesting examples both from the literature and the industry. As one can note, in this work we have carefully and systematically rewritten in Coq our previous Z characterisation of DFRS models. Hereafter, we only show the Coq definitions.

Considering the previous Coq definitions, and the VM example, the following code illustrates how to declare the name of a variable (create an instance of the record *VNAME*).

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

First, we give a name for the instance (*the_coin_sensor*), and then Coq enters on proof mode. After providing the value for the instance ("the_coin_sensor") via the command `apply (mkVNAME ...).`, we need to prove that the defined 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"* → `False`. In Coq, the logical negation (¬ *P*) is modelled as *P* → `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).

It is worth noting that the proof is finished with `Defined.` instead of `Qed.`. This makes the definition transparent, and it can be unfolded later (we will be able to retrieve the string associated with *vname*). When a proof is finished with `Qed.`, it is marked as opaque (proof irrelevance).

To model system timers, we define the record *STIMERS*, which also comprises a list of variables and types, but with different well-formedness properties (*ind_rules_stimers*). *STIMERS* also represents a function from variable names to types, but here the function can be empty (the system has no timers), and neither *bool* nor *int* are allowed types (boolean and integer values cannot be assigned to system timers).

```
Record STIMERS : Set := mkSTIMERS {
  stimers : list (VNAME × TYPE)
  ; rules_stimers : ind_rules_stimers stimers
}.
```

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). For the vending machine, we have the following definitions.

```
Definition vm_I : SVARS.
Proof.
  apply (mkSVARS [(the_coin_sensor, Tbool) ; (the_coffee_request_button, Tbool)]).
  (* proof omitted *)
Defined.

Definition vm_variables : DFRS_VARIABLES.
Proof.
```

```
  apply (mkDFRS_VARIABLES vm_I vm_O vm_T vm_gcvar).
  (* proof omitted *)
Defined.
```

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

3.1.1.2 Initial state

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

The initial state for the VM is the following: both input signals are false, the system mode is idle (*i 1*), the machine output is strong coffee (*i 0*), and the request timer and the global clock are equal to 0; *i*, *n*, and *b* are constructors for integer, natural, and boolean values (within the *VALUE* definition), respectively. The initial state of the VM is defined as follows.

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

```
Definition vm_s0 : DFRS_INITIAL_STATE.
Proof.
  apply (mkDFRS_INITIAL_STATE vm_state).
Defined.
```

It is important to bear in mind that the definitions in Coq for a particular example, such as the VM, are automatically generated from the corresponding systems requirements, written according to our controlled natural language.

3.1.1.3 Functions

A DFRS might comprise multiple concurrent components. The behaviour of each component is described by a function. The behaviour of the entire s_DFRS is then defined as a list of functions (F), which cannot be empty (ensured by *ind_rules_dfrs_functions*). Each function (*function*) is a list of 4-tuples: a static guard (*EXP*), a timed guard (*EXP*), a list of assignments (*ASGMTS*), and requirement traceability information (*REQUIREMENT*). Although in our Z characterisation of functions the traceability information was not present, it was considered by the tool implementation. Now, this information is also part of the formal definition in Coq.

The first two elements define the static and timed conditions necessary to be met to react by performing the respective assignments. One of these two conditions can be empty, but not both (ensured by *ind_rules_function*).

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

In our work, the static and timed guards adhere to a Conjunctive Normal Form (CNF): $(c_1 \vee \cdots \vee c_n) \wedge \cdots \wedge (c_1 \vee \cdots \vee c_m)$. The term *DIS* refers to a list of disjunctive clauses, such as $(c_1 \vee \cdots \vee c_n)$. The term *EXP* refers to a list of conjunctive clauses and, thus, represent a static or timed guard.

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 that the actions described in REQ001 are performed when the coin sensor changes to true (its previous value was false – *REQ001_sg_disj1*, and its current value is true – *REQ001_sg_disj2*), and the system mode is idle (*i 1* – see *REQ001_sg_disj3*). As said before, 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 omitted *)
Defined.
Definition REQ001_sg_disj2 : DISJ.
Proof.
  apply (mkDISJ [mkBEXP (current (the_coin_sensor)) eq (b true)]).
  (* proof omitted *)
Defined.
Definition REQ001_sg_disj3 : DISJ.
Proof.
  apply (mkDISJ [mkBEXP (current (the_system_mode)) eq (i 1)]).
  (* proof omitted *)
Defined.
```

It is worth noting that the order of the formal definitions does not necessarily reflect the order of the conditions in the requirement (text). The order of the definitions depends on how the text is parsed and how the underlying abstract syntax tree is traversed. Nevertheless, this has no impact on the semantics, since conjunction is a commutative operator.

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. 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*). These definitions are grouped later to define a possible system reaction; see definition *the_coffee_machine_system*.

```
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 omitted *)
Defined.
```

Finally, the system behaviour is defined as the collection of 4-tuples (static guard, timed guard, assignments and requirement traceability information) for each system requirement. This is captured by the definition of *the_coffee_machine_system*. As the VM comprises a single system component, its functions component (*vm_functions*) has a single function (*the_coffee_machine_system*). Note again that the order of the 4-tuples (static guard, timed guard, assignments, and traceability information) does not necessarily reflect the requirements order. The order in the formal definition depends upon how the requirements are parsed, and not the order they are defined.

```
Definition the_coffee_machine_system : FUNCTION.
Proof.
  apply (mkFUNCTION [
          (REQ001_sg, REQ001_tg, REQ001_asgmts, "REQ001")
          ; (REQ003_sg, REQ003_tg, REQ003_asgmts, “REQ003”)
          ; (REQ005_sg, REQ005_tg, REQ005_asgmts, “REQ005”)
          ; (REQ004_sg, REQ004_tg, REQ004_asgmts, “REQ004”)
          ; (REQ002_sg, REQ002_tg, REQ002_asgmts, “REQ002”)
        ]).
  (* proof omitted *)
Defined.

Definition vm_functions : DFRS_FUNCTIONS.
Proof.
  apply (mkDFRS_FUNCTIONS [the_coffee_machine_system]).
  (* proof omitted *)
Defined.
```

In what follows, to provide a complete perspective, one can see all system requirements of the vending machine:

- 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.
- REQ002: When the system mode is choice, and the coin sensor is false, and the coin sensor was false, and the coffee request button changes to pressed, and the request timer is lower than or equal to 30.0, the coffee machine system shall: reset the request timer, assign preparing weak coffee to the system mode.
- REQ003: When the system mode is choice, and the coin sensor is false, and the coin sensor was false, and the coffee request button changes to pressed, and the request timer is greater than 30.0, the coffee machine system shall: reset the request timer, assign preparing strong coffee to the system mode.
- REQ004: When the system mode is preparing weak coffee, and the request timer is greater than or equal to 10.0, and the request timer is lower than or equal to 30.0, the coffee machine system shall: assign idle to the system mode, assign weak to the coffee machine output.
- REQ005: When the system mode is preparing strong coffee, and the request timer is greater than or equal to 30.0, and the request timer is lower than or equal to 50.0, the coffee machine system shall: assign idle to the system mode, assign strong to the coffee machine output.

The omitted proofs guarantee that all definitions meet the well-formedness conditions. As explained in Section 3.1.3, 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.

3.1.1.4 s_DFRSs

An s_DFRS is composed by the previously defined elements. Various consistency properties are enforce by *ind_rules_s_dfrs*; for instance, the initial state must provide values for all system variables, the static and timed guards, as well as the assignments, must also be type consistent with the declaration of system variables. We refer to our git repository for the definition in details of *ind_rules_s_dfrs*.

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

It is important 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, as we illustrate later.

Regarding the VM example, its symbolic DFRS is defined as follows (*vm_s_dfrs*).

```
Definition vm_s_dfrs : s_DFRS.
Proof.
  apply (mkS_DFRS vm_variables vm_s0 vm_functions).
  (* proof omitted *)
Defined.
```

It comprises the system variables (*vm_variables*), the system initial state (*vm_s0*), and the system functions (*vm_functions*), defined before.

To illustrate a possible specification error, consider the following definition. Differently from the one presented in the previous section, which is the correct one, here we try to assign a boolean value (*b false*) to *the_system_mode*.

```
Definition REQ001_asgmt2 : ASGMT.
Proof.
```

```
  apply (mkASGMT (the_system_mode, (b false))).
Defined.
```

When trying to create the VM instance (*vm_s_dfrs*), the proof fails, since we are trying to assign a value that is not compatible with the variable type. The following error message is displayed: *Unable to unify "true" with "s_dfrs_fun_rules.fun_rules_s_dfrs vm_variables vm_s0 vm_functions"*, which means that some s_DFRS rule does not hold. It remains as future work to provide the user with a more informative description, possibly tracing the error to the system requirements.

3.1.2 Coq characterisation of expanded DFRSs

An e_DFRS is a 7-tuple: (I, O, T, $gcvar$, s_0, S, TR), comprising (input, output, and timer) variables, besides a global clock, along with an initial state and a set of states. Its definition also considers a transition relation, which relates states by means of delay and function transitions. Therefore, the main difference with respect to an s_DFRS is its transition relation.

3.1.2.1 Transition relation

A transition relation (*TRANSREL*) 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 (*DFRS_TRANSITION_RELATION*) is defined as a transition relation, along with its well-formedness properties. A number of consistency rules are enforced by *rules_TR*. For instance, the source state of a function transition must not be stable (there should be at least one function entry in the symbolic DFRS whose static and timed guards evaluate to true in the source state), and the target state of the function transition should reflect the corresponding assignments applied to the source state. Analogously, 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 (Figure 3).

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

3.1.2.2 e_DFRSs

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 3.2.1).

3.1.3 Functional characterisation of well-formedness properties

In Section 3.1.1.1, 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 the original 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.

The equivalence proof for *ind_rules_vname* (a variable name shall be different from "gc") is shown below. Since it is an equivalence proof, we need to prove both sides of the implication. We do that by reaching (via different ways) a contradiction in the proof context (`inversion H'`).

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

Now it is possible to define *the_coin_sensor* as follows. The theorem *theo_rules_vname* allows rewriting the proof goal considering the functional characterisation of the well-formedness property. 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 of this chapter follow this pattern. Therefore, all well-formedness proofs are discharged automatically, with no 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 3.1, we have written about 1.5 KLOC of proof scripts. All proof scripts can be found in <https://github.com/igormeira/DFRScoq>.

3.2 APPLICATIONS OF OUR COQ CHARACTERISATION OF DFRSS

As said before, our Coq characterisation of DFRS models supports requirement validation and generation of test cases, as detailed in Sections 3.2.1 and 3.2.2, respectively. This achieves our goal of providing a single, unified and formal framework for validation, verification and testing of timed data-flow reactive systems.

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

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

The variable *combinations* considers all possible value combinations for input. For example, 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). The parameter *possibilities* lists all possible values for input variables. This information is necessary, for instance, when dealing with integers, since we do not want to consider all possible integer values. In the NAT2TEST tool (see Section 4.1), the user can inform such lists for input variables.

Figure 5 shows an application of *genTransitions* considering the initial state of the VM, a time step of 1s, and a bound *num = 3*. The initial state is the topmost one. Since this state 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 these four new states, *genTransitions* recurses considering the first generated state (second row, leftmost state) and a bound *num = 2*. Since this state is also stable, 4 new states are generated (bottommost ones). Now, *generateTransitions* recurses again, considering the second generated state (second row, second state from left to right) 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 the state presented in the third row. Finally, *genTransitions* recurses again considering the third state (from left to right) of the second row and a bound *num = 0*. Since the bound is equal to 0, the function terminates.

Now, we define a function *buildTR*, which expands dynamically an s_DFRS, bounded by the value of *num*. More precisely, it 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 3.1.2); each transition relates two states by means of a transition label (delay or function transition).

```
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.
```

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 – see Figure 5. Then, the current state is removed from *toVisit*, and the reached

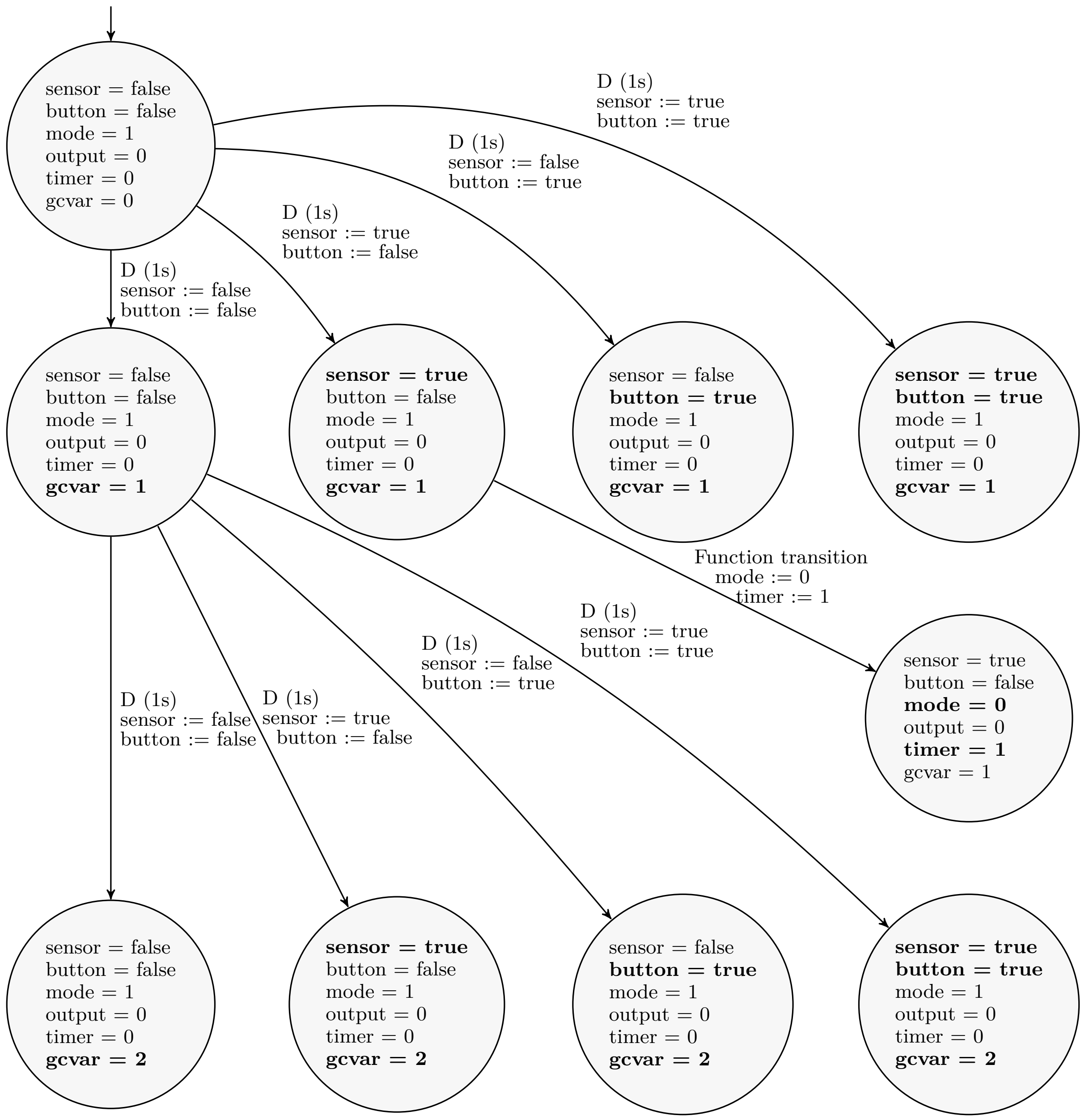


Figure 5 – Bounded exploration of the vending machine e-DFRS

states are added to *toVisit*. Afterwards, this function recurses considering the predecessor of *num*. When it reaches 0, the function stops yielding an empty list of transitions to be appended to the previously computed results. 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. 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 and should be avoided.

Considering the functions previously defined, *expandedDFRS* yields an e_DFRS from a given s_DFRS whose transition relation is bounded by the value of *limiter*. First, it computes the transition relation considering the provided bound (*call_buildTR* is just a wrapper function to *buildTR*). Then, the e_DFRS states are extracted from the states reached by the transition relation, and also the initial state. Duplicates are removed, since we are considering lists. The variables of the e_DFRS are the same of the given s_DFRS. Finally, an e_DFRS is built from the previously computed elements, namely: variables, states and the transition relation.

```
Definition expandedDFRS (sdfrs : s_DFRS) (limiter : nat)
  (possibilities : list (VNAME × list VALUE)) : e_DFRS :=
  let TR := mkTRANSREL (call_buildTR ... ) in
  let
     states := removeDuplicateStates (
                    (get_list_states TR.(transrel) []) ++
                    [sdfrs.(s_dfrs_initial_state).(s0)])
  in
  let dfrs_variables := (mkDFRS_VARIABLES ...) in
  let dfrs_states := (mkDFRSSTATES (mkSTATES states ...) ...) in
  let dfrs_tr := (mkDFRSTRANSITIONREL (mkTRANSREL TR ...) ...) in
     mkE_DFRS dfrs_variables dfrs_states dfrs_tr ... .
```

Although this is beyond the scope of this work, 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. In this direction, we would also need to prove that the functions presented in this section are correct; for any given input, they always yield elements whose corresponding well-formedness properties hold. Since this is not within the scope of this work, these proofs were not carried out here.

3.2.2 Generating test cases with QuickChick

Besides validation via bounded simulation, our Coq characterisation of DFRSs also supports generation of test data with the support of QuickChick. It is achieved by sampling valid traces.

As discussed in Section 2.4, property-based testing (supported in Coq by 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 3.2.1). 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 3.2.1), 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[3].

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

The definition *vm_initial_state* refers to the initial state of the VM, *vm_s_dfrs* refers to the s_DFRS that describes the VM behaviour, *vm_possibilities* details the possible values for the system inputs, and the number 600 is the maximum allowed number of transitions for each generated trace. This number was empirically chosen; greater values lead to no enhancement in the performed empirical analyses.

```
Definition vm_initial_state := vm_state.
Definition vm_possibilities := [(the_coin_sensor, [b false; b true]);
                                (the_coffee_request_button, [b false; b true])].
```

Table 4 shows, in a tabular form, a fragment of an output (test data) generated via QuickChick. The labels *time*, *sensor*, *button*, *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 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,

[3] QuickChick manual: <softwarefoundations.cis.upenn.edu/qc-current/>

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

time	**sensor**	**button**	**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

when defining functions (see Section 3.1.1 – Functions), we can also extract requirement coverage information from the generated test data.

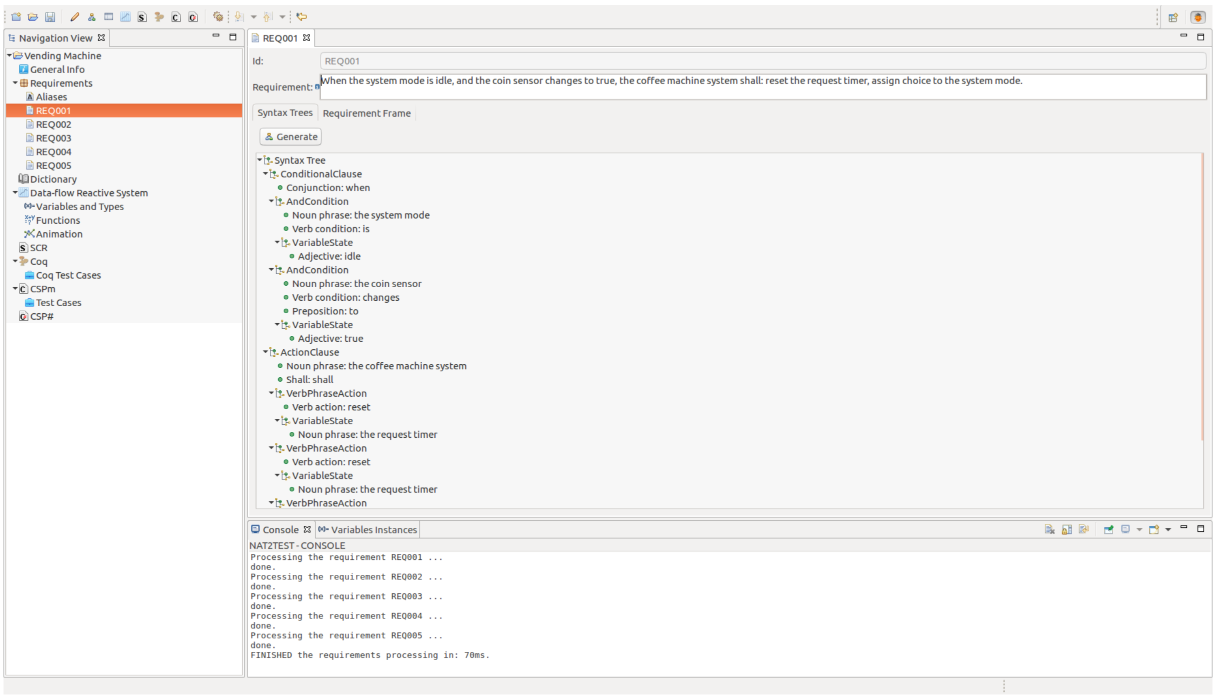
The time step needs to be set carefully, since a low granularity (high value) might prevent from seeing the exact moment the system reacts to some stimuli. However, a granularity too high (low value) might be responsible for generating excessive long test sequences, where we see the time advancing in small steps. In our empirical analyses (Section 4.2, 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 TOOL SUPPORT AND EMPIRICAL ANALYSES

The testing strategy presented in the previous chapter (Section 3.2.2) has been integrated into the NAT2TEST tool (CARVALHO et al., 2015). In Section 4.1 we detail this integration, and in Section 4.2 we discuss the empirical analyses performed.

4.1 INTEGRATION WITH THE NAT2TEST TOOL

The NAT2TEST tool is a multi-platform tool written in Java, using the Eclipse RCP framework[1]. It supports writing system requirements adhering to the SysReq-CNL grammar, presenting the corresponding syntax tree if the requirement is valid according to this grammar (see Figure 6).



Id: REQ001

Requirement: 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.

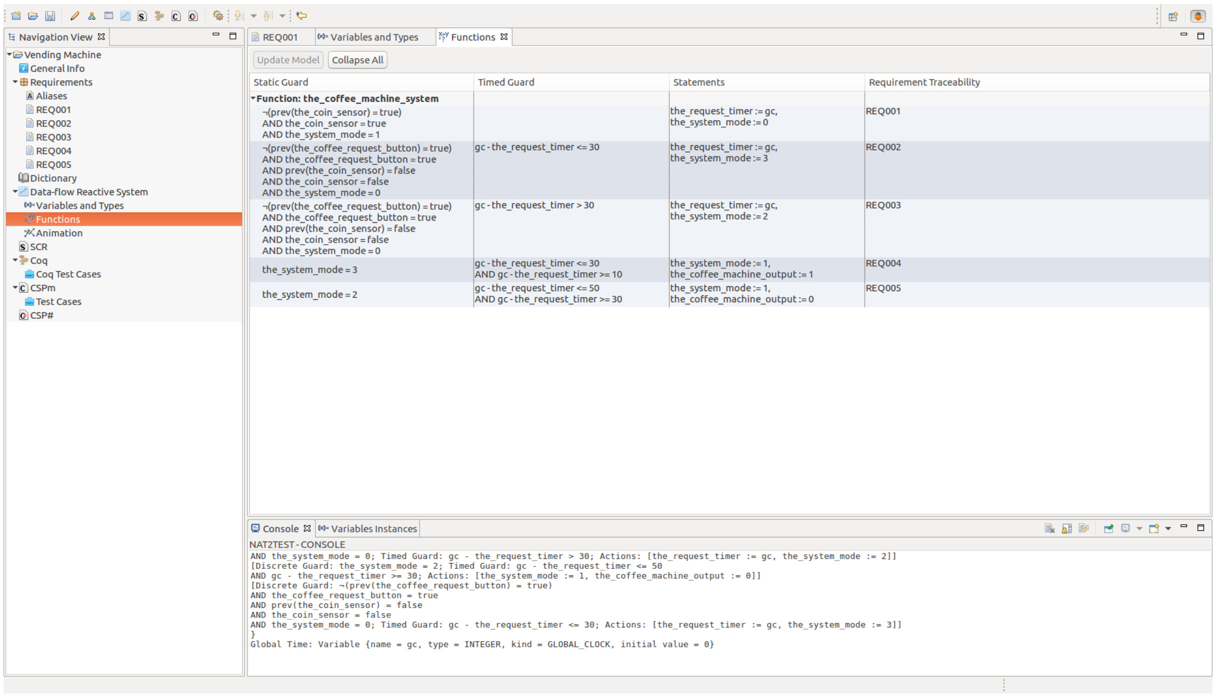
```
Syntax Tree
  ConditionalClause
    Conjunction: when
    AndCondition
      Noun phrase: the system mode
      Verb condition: is
      VariableState
        Adjective: idle
    AndCondition
      Noun phrase: the coin sensor
      Verb condition: changes
      Preposition: to
      VariableState
        Adjective: true
  ActionClause
    Noun phrase: the coffee machine system
    Shall: shall
    VerbPhraseAction
      Verb action: reset
      VariableState
        Noun phrase: the request timer
    VerbPhraseAction
      Verb action: reset
      VariableState
        Noun phrase: the request timer
    VerbPhraseAction
```

```
NAT2TEST - CONSOLE
Processing the requirement REQ001 ...
done.
Processing the requirement REQ002 ...
done.
Processing the requirement REQ003 ...
done.
Processing the requirement REQ004 ...
done.
Processing the requirement REQ005 ...
done.
FINISHED the requirements processing in: 70ms.
```

Figure 6 – NAT2TEST tool: editing system requirements

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 Figure 7, 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 instance, the first line presents a textual representation of the function entry for the 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*).

[1] Eclipse RCP: <http://wiki.eclipse.org/index.php/Rich_Client_Platform>

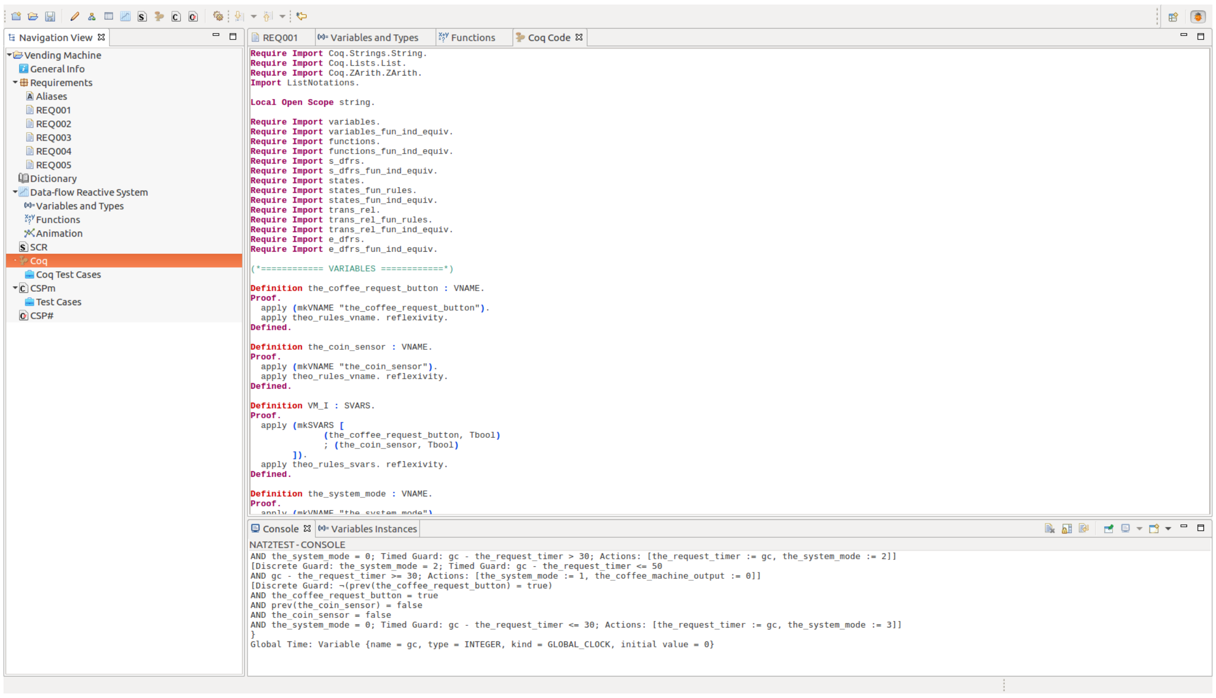


Static Guard	Timed Guard	Statements	Requirement Traceability
Function: the_coffee_machine_system			
¬(prev(the_coin_sensor) = true) AND the_coin_sensor = true AND the_system_mode = 1		the_request_timer := gc, the_system_mode := 0	REQ001
¬(prev(the_coffee_request_button) = true) AND the_coffee_request_button = true AND prev(the_coin_sensor) = false AND the_coin_sensor = false AND the_system_mode = 0	gc - the_request_timer <= 30	the_request_timer := gc, the_system_mode := 3	REQ002
¬(prev(the_coffee_request_button) = true) AND the_coffee_request_button = true AND prev(the_coin_sensor) = false AND the_coin_sensor = false AND the_system_mode = 0	gc - the_request_timer > 30	the_request_timer := gc, the_system_mode := 2	REQ003
the_system_mode = 3	gc - the_request_timer <= 30 AND gc - the_request_timer >= 10	the_system_mode := 1, the_coffee_machine_output := 1	REQ004
the_system_mode = 2	gc - the_request_timer <= 50 AND gc - the_request_timer >= 30	the_system_mode := 1, the_coffee_machine_output := 0	REQ005

```
NAT2TEST - CONSOLE
AND the_system_mode = 0; Timed Guard: gc - the_request_timer > 30; Actions: [the_request_timer := gc, the_system_mode := 2]]
[Discrete Guard: the_system_mode = 2; Timed Guard: gc - the_request_timer <= 50
AND gc - the_request_timer >= 30; Actions: [the_system_mode := 1, the_coffee_machine_output := 0]]
[Discrete Guard: ¬(prev(the_coffee_request_button) = true)
AND the_coffee_request_button = true
AND prev(the_coin_sensor) = false
AND the_coin_sensor = false
AND the_system_mode = 0; Timed Guard: gc - the_request_timer <= 30; Actions: [the_request_timer := gc, the_system_mode := 3]]
}
Global Time: Variable {name = gc, type = INTEGER, kind = GLOBAL_CLOCK, initial value = 0}
```

Figure 7 – NAT2TEST tool: s-DFRS function for the VM

In this work, we first 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 3.1.1. This was implemented as a separate library (*CoqGenerator*) and then integrated into the NAT2TEST code. In Figure 8, one can see 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*.



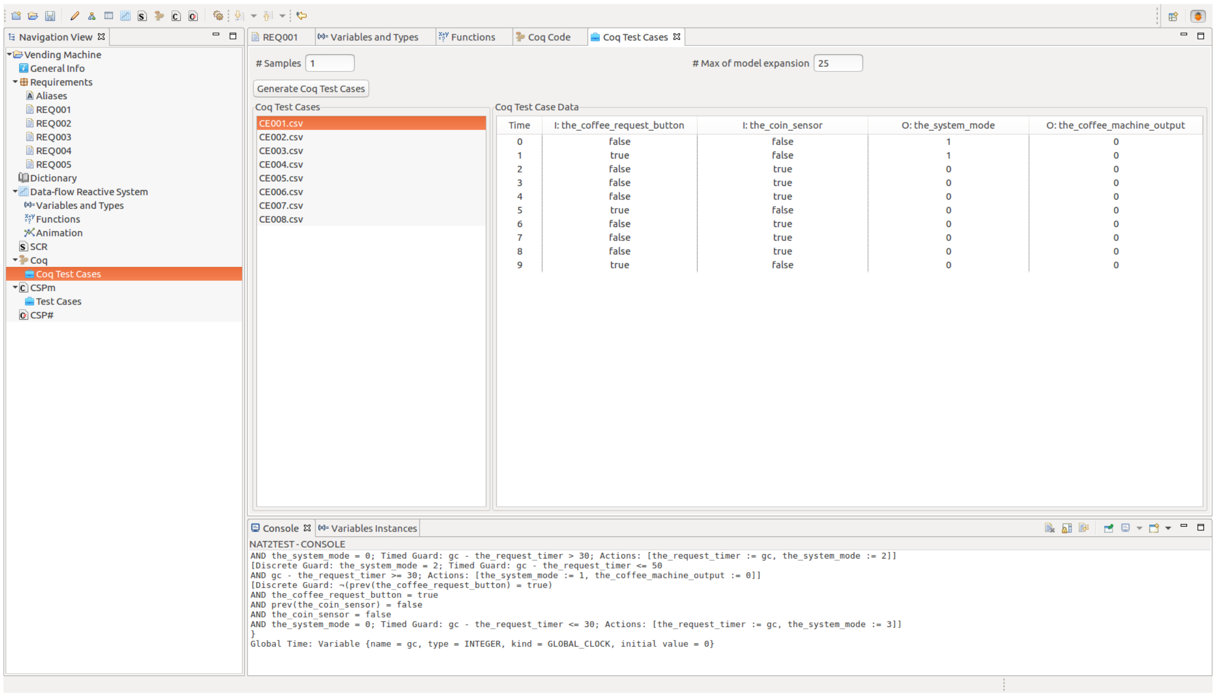
```
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.
Require Import Coq.ZArith.ZArith.
Import ListNotations.

Local Open Scope string.

Require Import variables.
Require Import variables_fun_ind_equiv.
Require Import functions.
Require Import functions_fun_ind_equiv.
Require Import s_dfrs.
Require Import s_dfrs_fun_ind_equiv.
Require Import states.
Require Import states_fun_rules.
Require Import states_fun_ind_equiv.
Require Import trans_rel.
Require Import trans_rel_fun_rules.
Require Import trans_rel_fun_ind_equiv.
Require Import e_dfrs.
Require Import e_dfrs_fun_ind_equiv.

(*============ VARIABLES ============*)

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

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

Definition VM_I : SVARS.
Proof.
  apply (mkSVARS [
             (the_coffee_request_button, Tbool)
             ; (the_coin_sensor, Tbool)
         ]).
  apply theo_rules_svars. reflexivity.
Defined.

Definition the_system_mode : VNAME.
Proof.
  apply (mkVNAME "the_system_mode").
```

```
NAT2TEST - CONSOLE
AND the_system_mode = 0; Timed Guard: gc - the_request_timer > 30; Actions: [the_request_timer := gc, the_system_mode := 2]]
[Discrete Guard: the_system_mode = 2; Timed Guard: gc - the_request_timer <= 50
AND gc - the_request_timer >= 30; Actions: [the_system_mode := 1, the_coffee_machine_output := 0]]
[Discrete Guard: ¬(prev(the_coffee_request_button) = true)
AND the_coffee_request_button = true
AND prev(the_coin_sensor) = false
AND the_coin_sensor = false
AND the_system_mode = 0; Timed Guard: gc - the_request_timer <= 30; Actions: [the_request_timer := gc, the_system_mode := 3]]
}
Global Time: Variable {name = gc, type = INTEGER, kind = GLOBAL_CLOCK, initial value = 0}
```

Figure 8 – NAT2TEST tool: generated Coq specification for the VM

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 (about 30 properties) hold. We emphasise that this verification is fully automatic. Therefore, if an error related to these properties is introduced into the Coq definition (due to an specification error or a translation error), some proof obligation would not be discharged; indicating, thus, a property violation. The violation will be shown as a Coq proof script error. It remains as future work to provide the user with a more informative description, possibly tracing the error to the system requirements.

Afterwards, a second library (*CoqTCGenerator*) was developed (about 0.6 KLOC) to perform the logical integration of the NAT2TEST tool with Coq and QuickChick in order to generate test cases. This library receives from the tool all necessary information, such as the Coq model generated for the considered example, besides test generation parameters: number of `Sample` calls, the maximum allowed number of delay/function transitions (*size*), and the possible values for input variables (*possibilities*) – see Section 3.2.2). Then, this library copies all necessary files, including our Coq characterisation of DFRSs and some Python integration scripts, to the project folder. After that, some changes are automatically performed in some files to consider the test generation parameters aforementioned. Finally, all Coq files are compiled and test cases are generated with the aid of QuickChick (see Figure 9).



Samples 1

Max of model expansion 25

Coq Test Case Data

Time	I: the_coffee_request_button	I: the_coin_sensor	O: the_system_mode	O: the_coffee_machine_output
0	false	false	1	0
1	true	false	1	0
2	false	true	0	0
3	false	true	0	0
4	false	true	0	0
5	true	false	0	0
6	false	true	0	0
7	false	true	0	0
8	false	true	0	0
9	true	false	0	0

NAT2TEST - CONSOLE

```
AND the_system_mode = 0; Timed Guard: gc - the_request_timer > 30; Actions: [the_request_timer := gc, the_system_mode := 2]]
[Discrete Guard: the_system_mode = 2; Timed Guard: gc - the_request_timer <= 50
AND gc - the_request_timer >= 30; Actions: [the_system_mode := 1, the_coffee_machine_output := 0]]
[Discrete Guard: ¬(prev(the_coffee_request_button) = true)
AND the_coffee_request_button = true
AND prev(the_coin_sensor) = false
AND the_coin_sensor = false
AND the_system_mode = 0; Timed Guard: gc - the_request_timer <= 30; Actions: [the_request_timer := gc, the_system_mode := 3]]
}
Global Time: Variable {name = gc, type = INTEGER, kind = GLOBAL_CLOCK, initial value = 0}
```

Figure 9 – NAT2TEST tool: test cases generated for the VM

Besides creating these two libraries, we also developed two new Graphical User Interfaces (GUIs) for the NAT2TEST tool. The first one is shown in Figure 8; it displays to the user the Coq code generated for the considered example. The second one is shown in Figure 9; it displays to the user the generated test data in a tabular form.

It is important to bear in mind that test data are generated in a fully automatic way from the system requirement described in natural language. Default values are considered for a number of parameters (e.g., the number of `Sample` calls, the maximum allowed number of delay/function transitions, among others), but the user can modify them according to his needs. This test data can be further used to simulate and to verify other system models (potentially more concrete, such as Simulink[2] 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 their correctness. 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, for instance, checking that enough test data was generated in order to test the system in accordance with the required coverage criteria.

4.2 EMPIRICAL ANALYSES

We evaluate our testing strategy based on Coq considering examples from the literature, but also from the aerospace and the automotive industry:

- VM: a vending machine (our running example),
- NPP: a simplified control system for safety injection in a nuclear power plant (publicly available in (LEONARD; HEITMEYER, 2003)),
- PC: a priority command function that decides whether the pilot or copilot side stick will have priority in controlling the airplane (provided by Embraer),
- TIS: part of the turn indicator system of Mercedes vehicles (publicly available[3]). In this example, there are two parallel components responsible for controling 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.

[2] Simulink: <https://www.mathworks.com/products/simulink.html>
[3] Turn indicator model: <http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/turn_indicator/index_e.html>

As detailed in the following sections, our evaluation considers performance (Section 4.3) and quality aspects (Section 4.4), the latter considering mutant-based strength analysis.

4.3 PERFORMANCE ANALYSIS

All data presented in this section and in the following one, whenever relevant, consider multiple executions. Table 5 presents general data about the considered examples. The largest example is the TIS, with 17 system requirements, in a 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 5.

Table 5 – General empirical data

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

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 5. 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 3.1), with about 2.3 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 3.2.2). 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 5 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 occurence of invalid test cases.

Table 6 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 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 more a 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).

Table 6 – 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

The time to generate test cases is linearly proportional to the number of sampling calls (see Figure 10) 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.

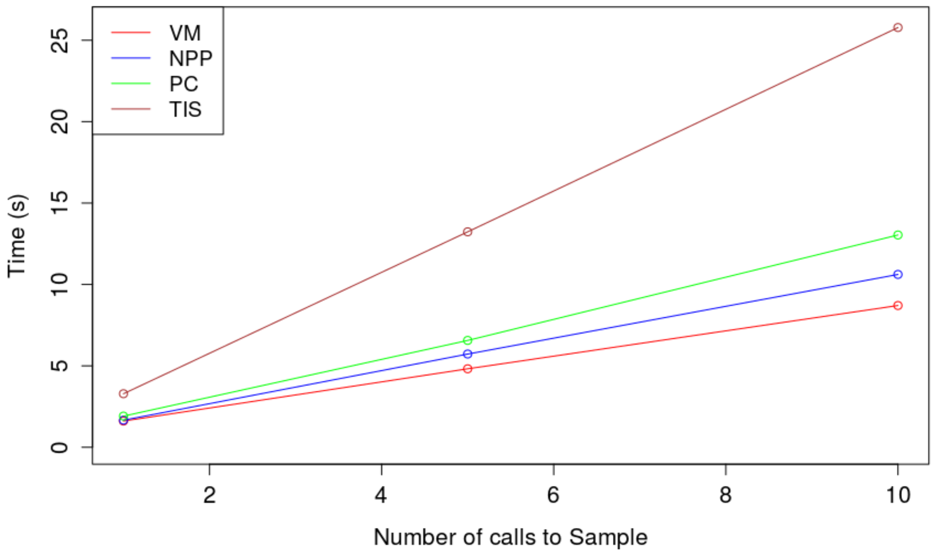


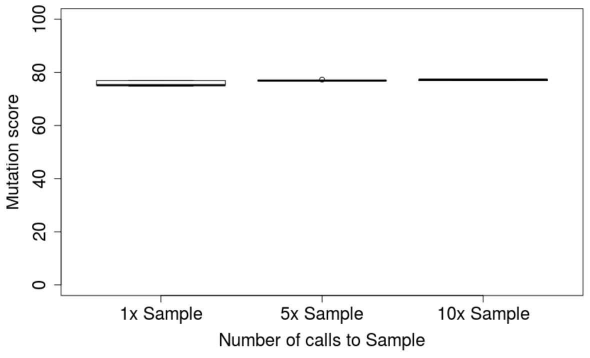
Figure 10 – Linearity between number of calls to `Sample` and test generation time

4.4 MUTANT-BASED STRENGTH ANALYSIS

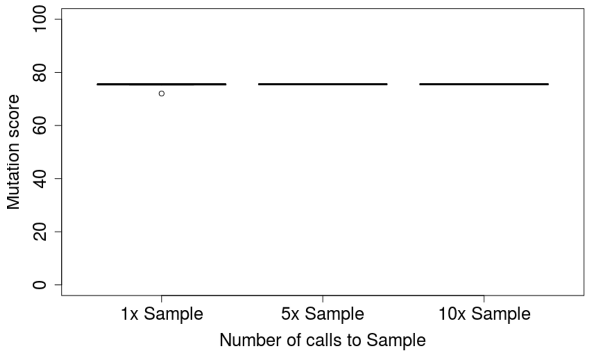
A mutant-based strength analysis was used to assess the quality of the generated tests. Mutation operators yield a trustworthy comparison of test cases strength because they create erroneous programs in a controlled and systematic way (ANDREWS; BRIAND; LABICHE, 2005). A good test suite should be able 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: a priori, 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 with SRCIROR (HARIRI; SHI, 2018), 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 (comparing integers, natural numbers and boolean values). Figure 11 shows the mutation score (ratio of killed/generated mutants) for the VM and PC. Figure 12 shows the same data for the NPP and the TIS.

Considering all examples and all datasets, we achieved an average mutation score of

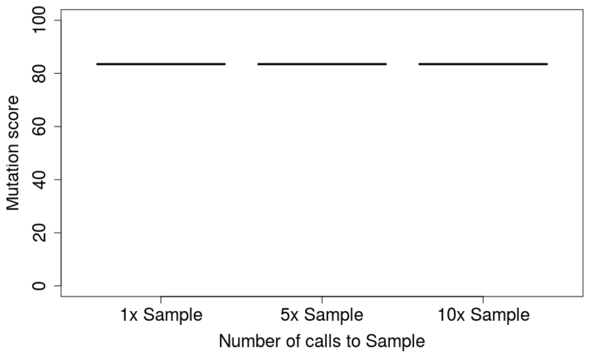


(a) Mutation score for the VM example

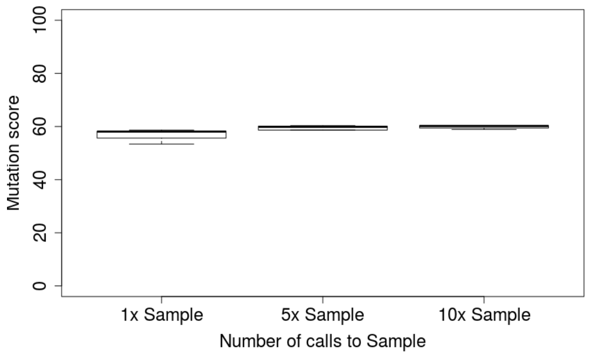


(b) Mutation score for the NPP example

Figure 11 – Mutant-based strength analysis (examples from the literature)



(a) Mutation score for the PC example



(b) Mutation score for the TIS example

Figure 12 – Mutant-based strength analysis (examples from industry)

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.

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. However, in general, increasing the number of `Sample` calls tends to increase the chances of finding new errors, since new test cases are potentially generated.

Regarding the PC example, we also have 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 weather `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 test suite with only randomly generated test cases should be enhanced by other means; for instance, taking into account 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[4].

[4] All empirical data and scripts for the VM, NPP and TIS are available online: see Footnote 1. The files regarding the PC example cannot be made publicly available.

5 CONCLUSIONS

This work presents a scalable, 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. Well-formedness properties of the models are automatically verified with no user intervention.

The generation of test data is also supported with the aid of the QuickChick tool. For now, we support test generation by sampling random valid traces. 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.

To carry out the empirical analyses, besides our Coq characterisation of DFRSs, the libraries developed and integrated into the NAT2TEST tool, we also created Python scripts to fully automate these analyses. 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.

5.1 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 (FAA, 2009). 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. The NAT2TEST strategy distinguishes itself by providing a flexible writing structure to describe timed data-flow reactive systems in controlled natural language, but also achieving automation. We refer to (CARVALHO; CAVALCANTI;

SAMPAIO, 2016) for a detailed comparison of the NAT2TEST strategy with others concerning the adoption of controlled natural languages.

The adoption of s-DFRS and e-DFRS models, opposed to other modelling notations, to formally represent the behaviour of timed data-flow reactive systems is a 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 SCADE/Lustre[1] code. Verification could be supported by tools such as the Simulink Design Verifier[2] or the SCADE Suite. Moreover, as reported in (MILLER; WHALEN; COFER, 2010), 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. In this context, one could, for instance, use Uppaal to automate the generation of test cases (LARSEN; MIKUCIONIS; NIELSEN, 2005). Besides test generation, Uppaal also supports checking invariants and reachability properties, which is also desired by us to the future (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 (WIMMER; LAMMICH, 2018), the authors constructs a verified model checker for timed automata using Isabelle/HOL (NIPKOW; WENZEL; PAULSON, 2002). Scalability should also be an issue here, since the underlying verification technique is defined in terms of state-space exploration.

An extension of our current work could rely on induction principles to prove properties (formalised in Coq) 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 the context of our work, it would be possible to apply the idea described in Section 3.2.1 and, then, verify the desired system properties via state-space exploration. However, scalability would be an issue again.

In (PAULIN-MOHRING, 2001), 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, to allow for test generation, besides being able to prove system properties via the

[1] SCADE suite: <https://www.ansys.com/products/embedded-software/ansys-scade-suite>
[2] Simulink Design Verifier: <https://www.mathworks.com/products/simulink-design-verifier.html>

development of proof scripts instead of using state-space exploration. However, a possible drawback is that DFRS models were particularly designed to facilitate their automatic generation from controlled natural-language requirements.

DFRS models are tailored for embedded systems whose inputs and outputs are always available, as signals. An advantage of a DFRS model is the fact that, as opposed to classical timed reactive systems notations such as timed automata, it is a state-rich notation that embeds and enforces a number of properties that are required in our context. For example, DFRS models enforce the principle of delayable transitions; they use delay transitions to represent environment stimuli and, thus, they cannot range over the output signals and timers of the system; they include no self-transitions; they ensure that there are no delay and function transitions emanating from the same state. 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 (FELIACHI et al., 2013), 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 (HONG et al., 2018; BRUCKER et al., 2016; WAN et al., 2009; AHRENDT; GLADISCH; HERDA, 2016), 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 (BRUCKER; WOLFF, 2009; BRUCKER; WOLFF, 2013; BERGHOFER; NIPKOW, 2004). In general, similar functionalities are provided among these different options.

5.2 FUTURE WORK

As future work, we envisage the following tasks.

- *Consider symbolic time representation.* As explained in the end of Section 3.2.2, when expanding an e-DFRS, as part of a requirement validation or of a test generation task, the time step considered by delays transitions needs to be set carefully. In a previous work (CARVALHO; SAMPAIO; MOTA, 2013), 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 the proof of general and domain-specific system properties.* Our Coq characterisation of DFRSs can be used to formalise and prove general (such as determinism, absence of time-lock) and domain-specific system properties (such as typical safety and liveness properties). Moreover, it would be relevant to devise proof tactics to (partially) automate the verification of such properties.

- *Support test generation guided by test purposes.* A test purpose can be seen as a test scenario defined manually by a domain specialist. In the context of this work, a test purpose would describe a scenario in terms of the observed values for the input and output signals. For instance, a scenario where the coin sensor becomes true and later (not necessarily just after the previous step) the system mode changes to *weak*. Then, one could use QuickChick to find (by sampling random valid traces) a test case where this behaviour is observed. This would ensure that the user-defined scenario is covered by the generated test cases.

- *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.

- *Generate correct-by-construct simulators for e-DFRSs.* As explained in the end of Section 3.2.1, the definitions created for validating system requirements via bounded exploration of models could be used to support the development of simulators for e-DFRSs, since Haskell and OCaml code can be extract from functional definitions in Gallina. Currently, the NAT2TEST tool has a simulator for e-DFRSs, which was implemented in Java, with no formal guarantees that it reflect the formal characterisation of DFRS models.

- *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 in Coq these phases in order to be able to prove properties about the underlying algorithms, achieving a formal process (completely in Coq) to generate test cases from controlled natural-language requirements.

REFERENCES

AHRENDT, W.; GLADISCH, C.; HERDA, M. Proof-based test case generation. In: ______. *Deductive Software Verification – The KeY Book: From Theory to Practice*. Cham: Springer International Publishing, 2016. p. 415–451. ISBN 978-3-319-49812-6.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is Mutation an Appropriate Tool for Testing Experiments? In: *Proceedings of International Conference on Software Engineering*. New York: ACM, 2005. p. 402–411. ISBN 1-58113-963-2.

BARZA, S.; CARVALHO, G.; IYODA, J.; SAMPAIO, A.; MOTA, A.; BARROS, F. Model Checking Requirements. In: RIBEIRO, L.; LECOMTE, T. (Ed.). *Formal Methods: Foundations and Applications*. Cham: Springer International Publishing, 2016. p. 217–234. ISBN 978-3-319-49815-7.

BERGHOFER, S.; NIPKOW, T. Random testing in Isabelle/HOL. In: *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004*. [S.l.: s.n.], 2004. p. 230–239.

BERTOT, Y.; CASTRAN, P. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. [S.l.: s.n.], 2010.

BRUCKER, A. D.; HAVLE, O.; NEMOUCHI, Y.; WOLFF, B. Testing the IPC Protocol for a Real-Time Operating System. In: GURFINKEL, A.; SESHIA, S. A. (Ed.). *Verified Software: Theories, Tools, and Experiments*. Cham: Springer, 2016. p. 40–60. ISBN 978-3-319-29613-5.

BRUCKER, A. D.; WOLFF, B. hol-TestGen. In: CHECHIK, M.; WIRSING, M. (Ed.). *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 417–420. ISBN 978-3-642-00593-0.

BRUCKER, A. D.; WOLFF, B. On theorem prover-based testing. *Formal Aspects of Computing*, v. 25, n. 5, p. 683–721, Sep 2013. ISSN 1433-299X.

CARVALHO, G.; BARROS, F.; CARVALHO, A.; CAVALCANTI, A.; MOTA, A.; SAMPAIO, A. NAT2TEST Tool: From Natural Language Requirements to Test Cases Based on CSP. In: CALINESCU, R.; RUMPE, B. (Ed.). *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2015. p. 283–290.

CARVALHO, G.; BARROS, F.; LAPSCHIES, F.; SCHULZE, U.; PELESKA, J. Model-Based Testing from Controlled Natural Language Requirements. In: ARTHO, C.; ÖLVECZKY, P. C. (Ed.). *Formal Techniques for Safety-Critical Systems*. Cham: Springer International Publishing, 2014. p. 19–35. ISBN 978-3-319-05416-2.

CARVALHO, G.; CAVALCANTI, A.; SAMPAIO, A. Modelling timed reactive systems from natural-language requirements. *Formal Aspects of Computing*, v. 28, n. 5, p. 725–765, Sep 2016. ISSN 1433-299X. Disponível em: <https://doi.org/10.1007/s00165-016-0387-x>.

CARVALHO, G.; FALCÃO, D.; BARROS, F.; SAMPAIO, A.; MOTA, A.; MOTTA, L.; BLACKBURN, M. NAT2TEST$_{SCR}$: Test case generation from natural language requirements based on SCR specifications. *Science of Computer Programming*, v. 95, Part 3, n. 0, p. 275 – 297, 2014. ISSN 0167-6423.

CARVALHO, G.; MEIRA, I. Modelling and testing timed data-flow reactive systems in coq from controlled natural-language requirements. In: *Proceedings of Brazilian Symposium on Formal Methods*. [S.l.: s.n.], 2019.

CARVALHO, G.; SAMPAIO, A.; MOTA, A. A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. In: GROVES, L.; SUN, J. (Ed.). *Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 148–164. ISBN 978-3-642-41202-8.

CAVALCANTI, A.; GAUDEL, M.-C. Testing for refinement in CSP. In: BUTLER, M.; HINCHEY, M. G.; LARRONDO-PETRIE, M. M. (Ed.). *Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 151–170.

CIMATTI, A.; CLARKE, E.; GIUNCHIGLIA, F.; ROVERI, M. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, v. 2, n. 4, p. 410–425, Mar 2000. ISSN 1433-2779. Disponível em: <https://doi.org/10.1007/s100090050046>.

CLAESSEN, K.; HUGHES, J. Quickcheck: A lightweight tool for random testing of haskell programs. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. USA: ACM, 2000. (ICFP '00), p. 268–279. ISBN 1-58113-202-6.

FAA. *Requirements Engineering Management Findings Report*. USA, 2009.

FELIACHI, A.; GAUDEL, M.-C.; WENZEL, M.; WOLFF, B. The Circus Testing Theory Revisited in Isabelle/HOL. In: GROVES, L.; SUN, J. (Ed.). *Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer, 2013. p. 131–147. ISBN 978-3-642-41202-8.

FILLMORE, C. J. The Case for Case. In: BACH; HARMS (Ed.). *Universals in Linguistic Theory*. [S.l.]: New York: Holt, Rinehart, and Winston, 1968. p. 1–88.

GAUDEL, M.-C. Testing can be formal, too. In: MOSSES, P. D.; NIELSEN, M.; SCHWARTZBACH, M. I. (Ed.). *TAPSOFT '95: Theory and Practice of Software Development*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. p. 82–96. ISBN 978-3-540-49233-7.

GIBSON-ROBINSON, T.; ARMSTRONG, P.; BOULGAKOV, A.; ROSCOE, A. W. FDR3 — A Modern Refinement Checker for CSP. In: ÁBRAHÁM, E.; HAVELUND, K. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 187–201. ISBN 978-3-642-54862-8.

HARIRI, F.; SHI, A. 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*. New York, NY, USA: ACM, 2018. (ASE 2018), p. 860–863. ISBN 978-1-4503-5937-5.

HENINGER, K.; PARNAS, D.; SHORE, J.; KALLANDER, J. *Software Requirements for the A-7E Aircraft - TR 3876*. [S.l.], 1978.

HONG, W.; NAWAZ, M. S.; ZHANG, X.; LI, Y.; SUN, M. Using Coq for Formal Modeling and Verification of Timed Connectors. In: CERONE, A.; ROVERI, M. (Ed.). *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2018. p. 558–573. ISBN 978-3-319-74781-1.

JACKLIN, S. *Certification of safety-critical software under DO-178C and DO-278A*. [S.l.], 2012.

JENSEN, K.; KRISTENSEN, L. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. [S.l.: s.n.], 2009. ISBN 978-3-642-00283-0.

LARSEN, K. G.; MIKUCIONIS, M.; NIELSEN, B. Online testing of real-time systems using uppaal. In: GRABOWSKI, J.; NIELSEN, B. (Ed.). *Formal Approaches to Software Testing: 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*. [S.l.]: Springer Berlin Heidelberg, 2005. p. 79–94.

LEONARD, E.; HEITMEYER, C. Program Synthesis from Formal Requirements Specifications Using APTS. *Higher Order Symbol. Comput.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 16, p. 63–92, 2003. ISSN 1388-3690.

MILLER, S. P.; WHALEN, M. W.; COFER, D. D. Software model checking takes off. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 53, n. 2, p. 58–64, fev. 2010. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/1646353.1646372>.

MOURA, L. de; BJØRNER, N. Z3: An efficient smt solver. In: RAMAKRISHNAN, C. R.; REHOF, J. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 337–340. ISBN 978-3-540-78800-3.

NIPKOW, T.; WENZEL, M.; PAULSON, L. C. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN 3540433767.

OLIVEIRA, B.; CARVALHO, G.; MOUSAVI, M.; SAMPAIO, A. Simulation of hybrid systems from natural-language requirements. In: *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*. [S.l.: s.n.], 2017. p. 1320–1325. ISSN 2161-8089.

PARASKEVOPOULOU, Z.; HRIŢCU, C.; DÉNÈS, M.; LAMPROPOULOS, L.; PIERCE, B. C. Foundational property-based testing. In: URBAN, C.; ZHANG, X. (Ed.). *Interactive Theorem Proving*. Cham: Springer International Publishing, 2015. p. 325–343. ISBN 978-3-319-22102-1.

PAULIN-MOHRING, C. Modelisation of Timed Automata in Coq. In: KOBAYASHI, N.; PIERCE, B. C. (Ed.). *Theoretical Aspects of Computer Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 298–315. ISBN 978-3-540-45500-4.

PELESKA, J.; VOROBEV, E.; LAPSCHIES, F.; ZAHLTEN, C. *Automated Model-Based Testing with RT-Tester*. [S.l.], 2011.

ROSCOE, A. W. *Understanding Concurrent Systems*. [S.l.]: Springer, 2010.

SANTOS, T.; CARVALHO, G.; SAMPAIO, A. Formal Modelling of Environment Restrictions from Natural-Language Requirements. In: MASSONI, T.; MOUSAVI, M. R. (Ed.). *Formal Methods: Foundations and Applications*. Cham: Springer International Publishing, 2018. p. 252–270. ISBN 978-3-030-03044-5.

SILVA, B. C. F.; CARVALHO, G.; SAMPAIO, A. CPN simulation-based test case generation from controlled natural-language requirements. *Science of Computer Programming*, v. 181, p. 111 – 139, 2019. ISSN 0167-6423. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167642319300516>.

TAHA, W.; DURACZ, A.; ZENG, Y.; ATKINSON, K.; BARTHA, F. A.; BRAUNER, P.; DURACZ, J.; XU, F.; CARTWRIGHT, R.; KONEČNÝ, M.; MOGGI, E.; MASOOD, J.; ANDREASSON, P.; INOUE, J.; SANT'ANNA, A.; PHILIPPSEN, R.; CHAPOUTOT, A.; O'MALLEY, M.; AMES, A.; GASPES, V.; HVATUM, L.; MEHTA, S.; ERIKSSON, H.; GRANTE, C. Acumen: An Open-Source Testbed for Cyber-Physical Systems Research. In: MANDLER, B.; MARQUEZ-BARJA, J.; CAMPISTA, M. E. M.; CAGÁŇOVÁ, D.; CHAOUCHI, H.; ZEADALLY, S.; BADRA, M.; GIORDANO, S.; FAZIO, M.; SOMOV, A.; VIERIU, R.-L. (Ed.). *Internet of Things. IoT Infrastructures*. Cham: Springer International Publishing, 2016. p. 118–130. ISBN 978-3-319-47063-4.

WAN, H.; CHEN, G.; SONG, X.; GU, M. Formalization and Verification of PLC Timers in Coq. In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. [S.l.: s.n.], 2009. v. 1, p. 315–323. ISSN 0730-3157.

WIMMER, S.; LAMMICH, P. Verified model checking of timed automata. In: BEYER, D.; HUISMAN, M. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2018. p. 61–78. ISBN 978-3-319-89960-2.