Article:
Carvalho, G., Cavalcanti, A. L. C. orcid.org/0000-0002-0831-1976 and Sampaio, A. C. A. (2016) Modelling Timed Reactive Systems from Natural-Language Requirements. Formal Aspects of Computing. pp. 725-765. ISSN: 1433-299X

https://doi.org/10.1007/s00165-016-0387-x

Under consideration for publication in Formal Aspects of Computing

Modelling Timed Reactive Systems from Natural-Language Requirements

Gustavo Carvalho[1], Ana Cavalcanti[2] and Augusto Sampaio[1]

[1]Universidade Federal de Pernambuco - Centro de Informática, 50740-560, Brazil
[2]University of York - Department of Computer Science, YO10 5GH, UK

Abstract. At the very beginning of system development, typically only natural-language requirements are documented. As an informal source of information, however, natural-language specifications may be ambiguous and incomplete; this can be hard to detect by means of manual inspection. In this work, we present a formal model, named Data-Flow Reactive System (DFRS), which can be automatically obtained from natural-language requirements that describe functional, reactive and temporal properties. A DFRS can also be used to assess whether the requirements are consistent and complete. We define two variations of DFRS: a symbolic and an expanded version. A symbolic DFRS (s-DFRS) is a concise representation that inherently avoids an explicit representation of (possibly infinite) sets of states and, thus, the state space-explosion problem. We use s-DFRS as part of a technique for test-case generation from natural-language requirements. In our approach, an expanded DFRS (e-DFRS) is built dynamically from a symbolic one, possibly limited to some bound; in this way, bounded analysis (e.g., reachability, determinism, completeness) can be performed. We adopt the s-DFRS as an intermediary representation from which models, for instance, SCR and CSP, are obtained for the purpose of test generation. An e-DFRS can also be viewed as the semantics of the s-DFRS from which it is generated. In order to connect such a semantic representation to established ones in the literature, we show that an e-DFRS can be encoded as a TIOTS: an alternative timed model based on the widely used IOLTS and ioco. To validate our overall approach, we consider two toy examples and two examples from the aerospace and automotive industry. Test cases are independently created and we verify that they are all compatible with the corresponding e-DFRS models generated from symbolic ones. This verification is performed mechanically with the aid of the NAT2TEST tool, which supports the manipulation of such models.

Keywords: natural-language; formal model; model mapping; TIOTS

1. Introduction

According to the Federal Aviation Administration (FAA), which published a report [FAA09] that discusses current practices concerning requirements engineering management, "*... the overwhelming majority of the*

Correspondence and offprint requests to: Gustavo Carvalho, Universidade Federal de Pernambuco - Centro de Informática, 50740-560, Brazil, e-mail: ghpc@cin.ufpe.br

survey respondents indicated that requirements are being captured as English text...". This supports the thesis that, at the very beginning of system development, typically only natural-language requirements are documented. As an informal source of information, natural-language specifications may be ambiguous and incomplete; this problem can be difficult to identify by means of manual inspection.

If formal models are derived from the requirements, it is possible to reason about the specification and its implementation formally. For instance, animation of these models might be a useful tool for validating the system requirements. Moreover, given a formal definition of ambiguity and incompleteness, a formal model can be used to check consistency and completeness of the specification to prove something that can be hard to achieve by means of manual inspection. Generation of test cases is also made possible by the availability of a formal model via Model-Based Testing (MBT) techniques. To achieve practical impact, however, automation is essential, since requiring knowledge of formal modelling by practitioners is often not feasible.

The research problem addressed here is the automatic generation of timed state-rich formal models from natural-language specifications to support test generation using a variety of techniques and tools, besides analysis of the specification. Formal modelling of requirements described using natural languages is not a new research topic. However, previous results do not cover time and state-richness simultaneously [ADS14, BCMW15, BGMC04, NSM14, ES07, SHG10] or rely on user intervention [AG06, Ili07, LHHR94, MTWH06, Sch09, SJV12], in which case it is necessary to identify and classify entities.

We define here a timed and state-rich automata-based notation for representation of natural-language requirements: Data-Flow Reactive System (DFRS). Besides writing the system requirements according to the grammar of a controlled natural language (CNL), and defining a dictionary, there is no need of user involvement to generate the proposed models. They are derived automatically from the requirements.

Moreover, we provide a technique to check properties of the formal models, including whether the natural-language requirements are consistent and complete. Opposed to other works, we consider here definitions of consistency and completeness tailored to our domain (reactive embedded systems). Requirements are said to be inconsistent if, in the natural language description, there are two or more requirements describing different system reactions for the same condition. In addition, we consider a specification to be complete if it describes how the system should react (which outputs are produced) for every possible situation (any given inputs). For instance, our interpretation of consistency contrasts to the one adopted in [BGFT10, FLGS14], where the focus is detecting the use of ambiguous terms and sentences that might lead to different interpretations, or different interpretations arising from the background knowledge of the reader, respectively.

The proposed model is part of a broader strategy, NAT2TEST (see Fig. 1), which generates test cases fully automatically from natural-language requirements based on different internal and hidden formalisms. The first phase of NAT2TEST is a syntactic analysis to determine whether the requirements are written in accordance with the CNL we proposed (SysReq-CNL). For each syntactically correct requirement, we obtain the corresponding syntax tree.

Although the adoption of this CNL by practitioners is an interesting and still open question, we already have indication that the NAT2TEST strategy is of practical relevance. We have designed and tested the SysReq-CNL considering industrial examples (provided by Embraer[1]). We have later also considered different examples provided by different companies. These include an example provided by Mercedes, discussed in this paper, as well as others not discussed here: a consolidation function, also in the aerospace domain, and the Ford car-alarm system reported in [ABJ+15]. In none of these case studies, there has been a need to adapt the CNL to cope with the new examples. So, we have some evidence that, although the CNL has a controlled structure, it is not limited to particular examples; rather, it seems flexible enough to express requirements of reactive systems more generally. Another aspect that also contributes to the general application of our CNL is the associated dictionary. While the CNL defines a writing structure, it does not restrict the vocabulary being used. The dictionary is defined by the user and it is domain specific.

The second phase of the NAT2TEST strategy is a semantic analysis, which maps the syntax trees into an informal semantic representation based on the Case Grammar theory [Fil68]. We have previously generated formal representations of the system requirements directly from this informal semantic representation. We have been able to construct models using CSP# [SLDP09] (the CSP [Ros10] dialect processed by the Process Analysis Toolkit – PAT[2]), the Intermediate Model Representation (IMR [PVLZ11] – NAT2TEST$_{IMR}$ [CBL+14]) of the RT-Tester tool[3], and the Software Cost Reduction (SCR [BBF97] – NAT2TEST$_{SCR}$

[1] www.embraer.com.br
[2] http://pat.comp.nus.edu.sg/
[3] https://www.verified.de/products/rt-tester/

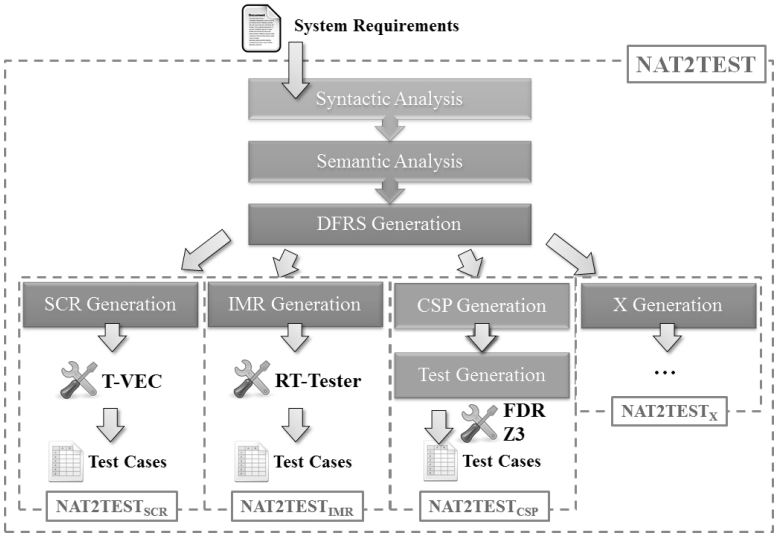

Fig. 1. The NAT2TEST strategy

[CFB+14]) version accepted by the T-VEC tool[4]. This has allowed us to take advantage of a variety of testing tools and techniques.

With those results, we have realised that, in spite of the particularities of each notation (e.g., IMR/RT-Tester has native support for dealing with time, whereas in T-VEC we need to manually encode the time evolving), we have used different notations to represent a common abstract behaviour. Based on this observation and on the experience of generating test cases using different formal representations, and with the perspective of instantiating our approach to several other target notations, we have proposed the DFRS model from which the more concrete notations can be generated. Translating the natural-language requirements to an intermediate formal notation is a promising alternative. This makes it easier to generate models in several target notations, since the direct translation from natural-language is a more elaborate task. A new architecture for our strategy that incorporates this new approach is presented in Fig. 1. Now, the third phase of the NAT2TEST strategy concerns the generation of DFRS models. Afterwards, this model is used to generate more concrete notations. Our account and implementation of NAT2TEST with CSP [CSM13] already takes advantage of DFRS representations.

Our focus in this paper is exactly the third step of NAT2TEST (*DFRS Generation*) and the DFRS model that it generates. A DFRS allows exploring the original requirements from different perspectives, besides being independent of a specific tool. In [CSM13], where we present a CSP timed input-output conformance relation, we discuss our preliminary ideas for the DFRS model. In [CCR+14], we formalise the model using Z [ISO02], besides describing how to use it to represent reactive systems. We also prove that a DFRS can be characterised as a Timed Input-Output Transition System (TIOTS)—a labelled transition system extended with time, which is widely used to characterise conformance relations for timed reactive systems. Being more abstract than a TIOTS, a DFRS comprises a more concise representation of timed requirements.

The present paper is an extension of [CCR+14]. Here, we propose a symbolic representation of DFRSs (s-DFRSs), define algorithms for incrementally generating s-DFRSs from natural-language requirements, and revise and extend the expanded DFRS (e-DFRS) discussed in [CCR+14]. An s-DFRS inherently avoids an explicit representation of possibly infinite sets of states and, thus, the state space explosion problem. An e-DFRS is built dynamically from its symbolic counterpart, possibly limited to some bound, and then used in bounded analyses such as requirements consistency, completeness, and reachability. To avoid confusion, we consider that, hereafter, s-DFRS and e-DFRS refer to symbolic and expanded DFRSs, respectively, while DFRS refers to both of them. In summary, this work enhances our previous efforts by:

- Proposing a symbolic representation of DFRSs (s-DFRSs);

[4] https://www.t-vec.com/solutions/tvec.php

- Defining six algorithms for incrementally generating s-DFRSs from natural-language requirements;
- Revising and extending the expanded DFRS (e-DFRS) definition;
- Defining a translation function from s-DFRSs to e-DFRSs;
- Showing how to check via e-DFRS whether the requirements are consistent, complete and reachable;
- Presenting our tool support for manipulating DFRS models (s-DFRSs and e-DFRSs).

Regarding related work, the formal model proposed here (DFRS) stands out for its richness and for the possibility of fully automatic generation of models from natural-language requirements. A DFRS can represent input-output variables, besides discrete and continuous time information.

To evaluate the expressiveness of DFRSs, we consider examples from four domains: a vending machine (VM a- toy example); a control system for safety injection in a nuclear power plant (NPP — toy example); a priority command (PC) control provided by Embraer; and the turn indicator system (TIS) of Mercedes vehicles. Test cases are independently generated for each example, and we assess whether they are compatible with the corresponding DFRS models.

Section 2 provides the formal definition of an s-DFRS. Section 3 describes how an s-DFRS can be automatically obtained from natural-language requirements; this strategy is implemented by the NAT2TEST tool. Section 4 first revisits the definition of an e-DFRS, then explains how it can be obtained from its symbolic counterpart. Finally, it shows how an e-DFRS model can be used to check whether its corresponding natural-language requirements are consistent and complete. Section 5 presents a theoretical and a practical validation of DFRS models. Finally, Sections 6 and 7 address related work and present our conclusions.

2. Definition and properties of an s-DFRS

In this section, first, we give an informal overview of the definition of DFRSs, and then we provide a formal definition for its symbolic representation. It is important to say that all definitions in Z presented here are syntactically correct and typed checked with the CZT plug-in for Eclipse[5].

2.1. Overview of DFRSs

A DFRS models an embedded system whose inputs and outputs are always available, as signals. The input signals can be seen as data provided by sensors, whereas the outputs as data provided to actuators. Each signal carries a binary value that represents boolean and numerical values. Hereafter, we directly refer to boolean (*true* and *false*, represented as *1* and *0*, respectively) and numerical values, instead of their binary representation.

It is assumed that a DFRS can also have internal timers, which might be used to trigger timed-based behaviour. An e-DFRS represents a timed system with continuous or discrete behaviour modelled as a state-based machine. Each state comprises a valuation for each element of the system: its inputs, outputs, and timers, as well as its global clock.

The states of an e-DFRS are connected by *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.

As a running example, we consider a Vending Machine (VM), which is an adaptation of the coffee machine presented in [LMN04]. Despite being a toy example, the vending machine comprises many different operating situations, which are described by five natural-language requirements.

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. Three seconds

[5] `http://czt.sourceforge.net/eclipse/`

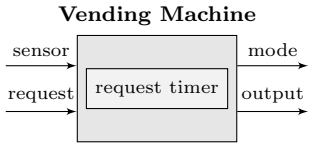

Fig. 2. The vending machine specification – abstract representation

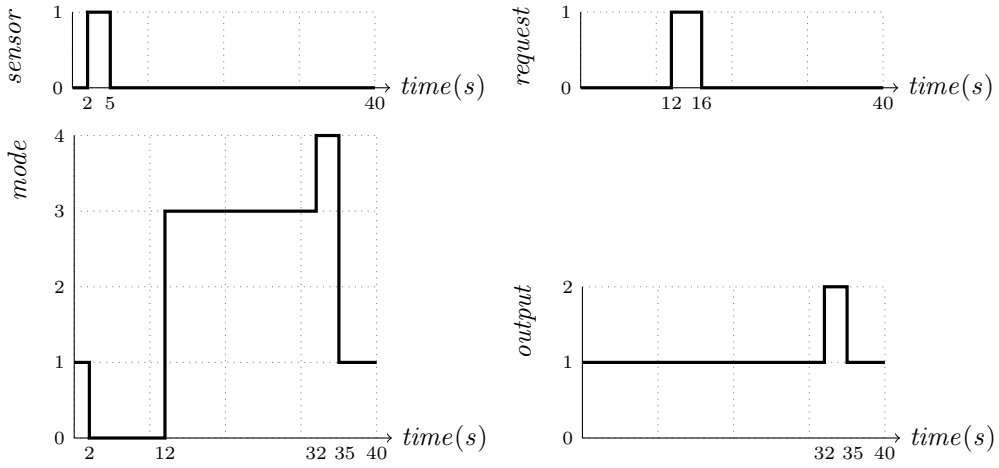

Fig. 3. Example of signals for the vending machine

after having produced weak or strong coffee, the system goes to the *reset* state, where it resets the type of the produced coffee to *undefined*, and then returns to the idle state.

As shown in Fig. 2, in this example we have two input signals related to the coin sensor (*sensor*) and the coffee request button (*request*). A *true* value means that a coin was inserted and the coffee request button was pressed. There are two output signals related to the system mode (*mode*) and the vending machine output (*output*). The values communicated by these signals reflect the system possible states (*idle*, *choice*, *weak*, *strong*, and *reset*) and the possible outputs (*undefined*, *weak*, and *strong*).

The VM has just one timer: the *request* timer, which is used to register the moments when a coin is inserted, when the coffee request button is pressed, and when the coffee is produced. Fig. 3 illustrates a scenario where there is 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 Fig. 3, a coin is inserted 2 seconds after starting the vending machine (the signal *sensor* changes to 1 – *true*). Immediately, the system state changes from *idle* to *choice*. Here, the system states are encoded as follows: *idle* $\mapsto$ 1, *choice* $\mapsto$ 0, *weak* $\mapsto$ 3, *strong* $\mapsto$ 2, and *reset* $\mapsto$ 4. Therefore, this change is represented by changing the value of the signal *mode* from 1 to 0. In this example, the signal *sensor* remains true for 3 seconds.

When 10 seconds have elapsed since the coin was inserted, which happens 2 seconds after starting the vending machine, the user requests a coffee (the signal *request* becomes true when the system global clock is equal to 12). At this moment, the system state changes to *weak* coffee (the signal *mode* becomes 3). In this example, the signal *request* remains true for 4 seconds.

As the coffee request occurs within 30 seconds of the coin being inserted, the system produces a weak coffee, which is represented as the value 2 of the signal *output*, 20 seconds after receiving the coffee request. We recall that a weak coffee is produced within 10 and 30 seconds after the coffee request. Then, as stated by the system description, the system goes to the *reset* state (the value of signal *mode* becomes 4), and 3 seconds later it goes back to the *idle* state, besides resetting the *output* to *undefined* (the signal *output* becomes 1).

As a state-based notation, the example illustrated in Fig. 3 is represented in an e-DFRS as a set of states and transitions (see Fig. 4). The states are related by *delay* (D) and *function* (F) transitions. The former represents time elapsing along with input stimuli, whereas the latter describes an instant reaction of

the system. It is important to emphasize that the diagram presented in Fig. 4 is just an illustration of an e-DFRS based on the particular scenario depicted in Fig. 3.

The first state is the top and left-most one: s, r, m, o, t and gc represent the current value of the coin sensor, the coffee request button, the system mode, the system output, the request timer and the system global clock, respectively. The delay transition emanating from this state denotes that after 2 seconds a coin is inserted and, thus, the value of s changes to *1*.

Afterwards, the system reaction is illustrated by a function transition that changes the system mode to *choice* (0), besides resetting the request timer. This reset operation is performed to register the moment when the coin is inserted, as this information is required when deciding if the system should produce a weak or a strong coffee.

The reset of a timer is represented by assigning 0 to it, but it is encoded as assigning the current value of the system global clock to the corresponding timer. This is possible as we have a single and global clock source (the system global clock). Otherwise, we would need to update the value of all timers every time a delay transition is performed. We provide more details about this design decision when formalising the e-DFRS elements.

We note that the changes produced by the transitions are highlighted in bold in Fig. 4. Moreover, each delay transition comprises the values of all input signals, whereas each function transition considers a subset of the output signals, besides the internal request timer, when appropriate.

When the user requests a coffee (third delay transition), as it is requested 10 seconds after inserting the coin ($gc - t = 12 - 2 = 10$), the system goes to the *weak* (3) state, and resets again the request timer. Later (20s), it changes the system output to 2 to denote the production of *weak* coffee. Finally, 3 seconds later it returns to the idle (1) state.

The main difference between an s-DFRS and its expanded version, characterised as just explained and illustrated in Fig. 4, is that the characterisation of an s-DFRS defines the initial state and means of calculating the next states via a set of functions. Differently, an e-DFRS comprises the set of all states and how they are related by delay and function transitions. Now, after presenting an informal discussion of DFRS models, we define the s-DFRS precisely.

2.2. Formal model of an s-DFRS

Formally, 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 a distinct kind of variable, which can be used to model temporal behaviour. The global clock is *gcvar*, a variable whose values are non-negative numbers representing a discrete or a dense (continuous) time. The initial state is s_0, and F is a set of functions. In what follows, we describe in Z the constituent components of an s-DFRS.

2.2.1. Inputs, Outpus and Timers

We use a given set $NAME$ to represent the set of all valid variable names, and define gc to be the name of the system global clock; as specified in the sequel, the component *gcvar* is a pair that maps gc to its type. Also $VNAME$ is the set of all system variables except for the global clock.

$$[NAME]$$

$$gc : NAME$$

$$VNAME == NAME \setminus \{gc\}$$

Based on these definitions, we define $SVARS$ and $STIMERS$ to represent inputs and outputs as different mappings of the same type, and timers, respectively, as finite partial functions from $VNAME$ to $TYPE$. We assume that the system has a finite number of inputs, outputs and timers; timers only hold non-negative values (*nat* or *ufloat*).

$$SVARS == \{f : VNAME \pfun TYPE \mid f \neq \emptyset \land \operatorname{ran} f \subseteq \{bool, int, float\}\}$$
$$STIMERS == \{f : VNAME \pfun TYPE \mid \operatorname{ran} f = \{nat\} \lor \operatorname{ran} f = \{ufloat\}\}$$

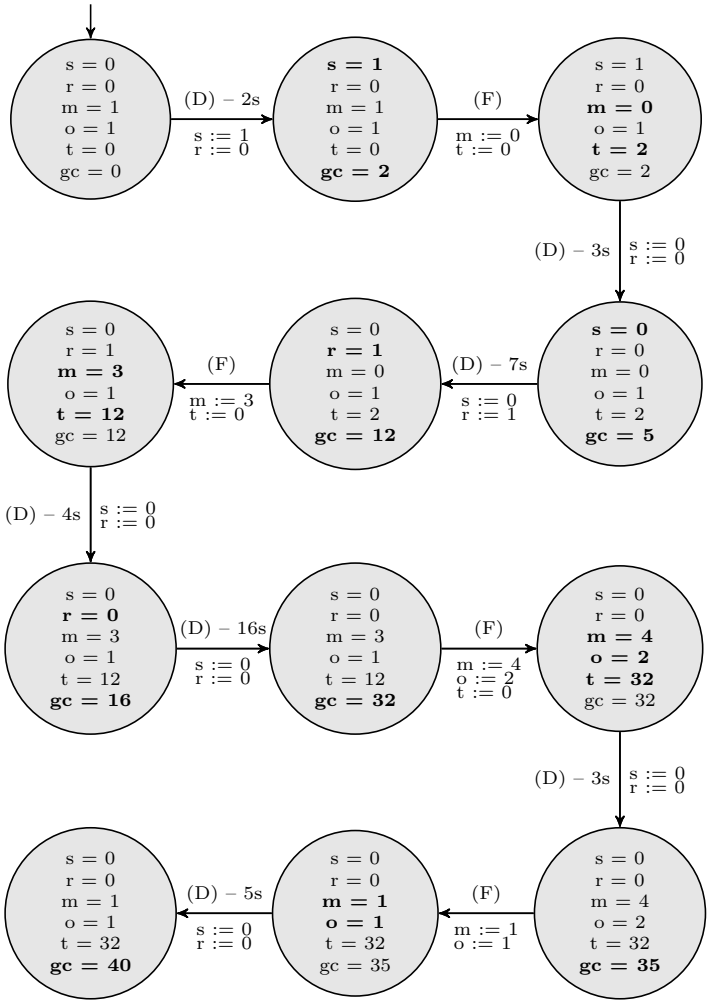

Fig. 4. The vending machine specification – e-DFRS representation

We consider as valid types boolean, integer and float types (*bool, int, nat, float, ufloat*). The type *ufloat* stands for unsigned float numbers.

$$TYPE ::= bool \mid int \mid nat \mid float \mid ufloat$$

More complex types are not needed since we are dealing with systems whose inputs and outputs are signals. As float numbers are not part of Standard Z, we provide an axiomatisation that fulfils our needs. For a more comprehensive axiomatisation, we refer, for instance, to ProofPower-Z[6].

The schema *DFRS_VARIABLES* defines the variables of a DFRS as a set of inputs (I), outputs (O), timers (T) and a global clock (*gcvar*). In Z, a schema is a named element used to structure and encapsulate definitions for reuse. As I and O are distinct and non-empty sets, we have that a DFRS has at least one input and one output variable. Differently, one can have a system with no timers: a DFRS whose behaviour is not dependent on time elapsing. These three sets (I, O and T) are disjoint.

[6] `http://www.lemma-one.com/ProofPower/index/index.html`

$$
\begin{array}{|l}
\textit{DFRS\_VARIABLES} \\
I, O : \textit{SVARS} \\
T : \textit{STIMERS} \\
\textit{gcvar} : \textit{NAME} \times \textit{TYPE} \\
\hline
\textit{gcvar} = (\textit{gc}, \textit{nat}) \lor \textit{gcvar} = (\textit{gc}, \textit{ufloat}) \\
\mathsf{disjoint}\ \langle \mathrm{dom}\, I, \mathrm{dom}\, O, \mathrm{dom}\, T \rangle \\
\mathrm{ran}\, T \subseteq \{\textit{gcvar}.2\} \\
\end{array}
$$

We note that our model can represent discrete, $gcvar = (gc, nat)$, or continuous time, $gcvar = (gc, ufloat)$, systems. Besides that, the type of all timers must be the same ($\mathrm{ran}\, T \subseteq \{gcvar.2\}$): one can analyse the behaviour of the system discretely or continuously, but not in both ways simultaneously.

Example 1 Besides the system global clock, five variables are identified in the VM example (see Fig. 4): two system inputs (*the_coin_sensor*—s, *the_coffee_request_button*—r), two system outputs (*the_system_mode*—m, *the_coffee_machine_output*—o), and one timer (*the_request_timer*—t). The sensor and the button are modelled by booleans that indicate whether a coin has been inserted or the button has been pressed. The system mode and the output of the VM are non-negative numbers. The request timer is modelled as a non-negative natural number since the temporal properties of the VM are defined in terms of discrete values (e.g., "... 30 seconds ..." instead of "... 30.0 seconds ..."). □

2.2.2. Initial state

A state is a relation between names and values, which include boolean and numerical values. The letter R refers to $\mathbb{R}$, and R^+ to the positive elements of $\mathbb{R}$.

$$\textit{BOOL\_VALUES} ::= \textit{TRUE} \mid \textit{FALSE}$$

$$\textit{VALUE} ::= b\langle\langle \textit{BOOL\_VALUES} \rangle\rangle \mid i\langle\langle \mathbb{Z} \rangle\rangle \mid n\langle\langle \mathbb{N} \rangle\rangle \mid f\langle\langle R \rangle\rangle \mid uf\langle\langle R^+ \rangle\rangle$$

Each name within a state is mapped to two values: the first one represents the previous value, and the second one the current value. Therefore, Fig. 4 shows a simplified and not the actual representation of states. For instance, in the first state, $s = 0$ should be $s \mapsto (b(\textit{false}), b(\textit{false}))$, and, in the second state, $s = 1$ should be $s \mapsto (b(\textit{false}), b(\textit{true}))$. Note that in Fig. 4 we use numbers to represent boolean values.

$$\textit{STATE} == \textit{NAME} \pfun (\textit{VALUE} \times \textit{VALUE})$$

Keeping the previous value of variables allows us to trigger system reactions to more complex behaviour. For example, the system goes to the *choice* state at the exact moment when the coin sensor changes from false to true; in other words, when the previous value of s is 0 and the current one is 1.

To simplify the access to current and previous values of a state, we consider two projection functions that yield the set of previous and current values of a given state: *previousValues* and *currentValues*, respectively. Their definition is not provided here as they are straightforward. Here, we concentrate on the most important definitions, but all omitted ones can be found in [CCS15][7].

The initial state of an s-DFRS is then defined as one possible state.

$$\textit{DFRS\_INITIAL\_STATE} == [\, s_0 : \textit{STATE} \,]$$

A variable n, whose type is t, is well typed in a state s if, and only if, n belongs to the domain of s, and the previous and current values associated with n in s belong to the set of possible values of t. This property of well typedness for variables in the context of a state is captured by the following predicate. Here, we use sets to denote predicates. The underlying idea is that *well_typed_var* is a set composed by all well typed variables. Therefore, we represent the fact of being well typed as belonging to *well_typed_var*.

$$
\begin{array}{|l}
\textit{well\_typed\_var} : \mathbb{P}(\textit{STATE} \times \textit{NAME} \times \textit{TYPE}) \\
\hline
\forall\, s : \textit{STATE};\ n : \textit{NAME};\ t : \textit{TYPE};\ v1, v2 : \textit{VALUE} \mid n \in \mathrm{dom}\, s \land (s(n)).1 = v1 \land (s(n)).2 = v2 \bullet \\
\qquad (s, n, t) \in \textit{well\_typed\_var} \Leftrightarrow v1 \in \textit{values}(t) \land v2 \in \textit{values}(t) \\
\end{array}
$$

[7] Available for download in `http://www.cin.ufpe.br/~ghpc/TR_DFRS.pdf`

The function *values* yields all possible values of a specific type t. Although we could directly access the range of a type, we use this auxiliary function to avoid legibility issues on bigger predicates.

Now, we lift the definition of well typedness for a state. Considering a set f of variables (names related to types), a state s is well typed if, and only if, it provides a value for each variable (that is, its domain is that of the function f) and those variables are well typed in s.

$$
\begin{array}{|l}
well\_typed\_state : \mathbb{P}(STATE \times (NAME \nrightarrow TYPE)) \\
\hline
\forall\, s : STATE;\ f : NAME \nrightarrow TYPE \bullet (s, f) \in well\_typed\_state \Leftrightarrow \\
\quad \operatorname{dom} s = \operatorname{dom} f \land (\forall\, n : \operatorname{dom} f;\ t : TYPE \mid f(n) = t \bullet (s, n, t) \in well\_typed\_var)
\end{array}
$$

Example 2 Considering the example shown in Fig. 4, its initial state is:

$$
\begin{array}{l}
\{(\mathrm{s} \mapsto (b(false), b(false)), \mathrm{r} \mapsto (b(false), b(false)), \\
\quad (\mathrm{m} \mapsto (n(1), n(1)), \mathrm{o} \mapsto (n(1), n(1)), \\
\quad (\mathrm{t} \mapsto (n(0), n(0)), \mathrm{gc} \mapsto (n(0), n(0))\}
\end{array}
$$

Regarding the variables m (*the_system_mode*) and o (*the_coffee_machine_output*), as previously said, the natural numbers represent elements of an enumeration of possible values: $\{0 \mapsto$*choice*, $1 \mapsto$*idle*, $2 \mapsto$*preparing strong coffee*, $3 \mapsto$*preparing weak coffee*, $4 \mapsto$*reset*$\}$, and $\{0 \mapsto$*strong coffee*, $1 \mapsto$*undefined output*, $2 \mapsto$*weak coffee*$\}$, respectively. □

2.2.3. Functions

The system behaviour is defined as a non-empty finite set of functions (see schema *DFRS_FUNCTIONS*) that describe how the system reacts in a given context. There is one function per system component; if the system comprises parallel components, we are going to have one function describing the behaviour of each component.

$$DFRS\_FUNCTIONS == [\, F == \mathbb{F}_1\, FUNCTION \,]$$

A function is a set of tuples. Each one models how the system reacts in a given context, which is characterised by a pair of static (*sGuard*) and timed (*tGuard*) guards, each one being a set (conjunction) of boolean expressions. The system reaction is denoted as a set of assignments (*asgmts*). Note that one of the guards can be empty, but not both. As formalised later, the static guards range over input and output variables, whereas timed guards are restricted to timers.

$$FUNCTION == \{sGuard, tGuard : EXP;\ asgmts : ASGMTS \mid sGuard \cup tGuard \neq \emptyset\}$$

When both guards evaluate to true in a given state, the system reacts instantly performing the corresponding assignments. These reactions are the *function transition* (F) shown in Fig. 4. An s-DFRS does not capture this dynamic behaviour (occurrence of reactions explicitly), but only includes the definition of the function that symbolically characterises the reactions.

The guards are expressions (*EXP*) whose structure adheres to a Conjunctive Normal Form (CNF): a finite set of conjunctions (*CONJ*) of disjunctions (*DISJ*), where each disjunction has at least one binary expression (*BEXP*).

$$
\begin{array}{l}
EXP == CONJ \\
CONJ == \mathbb{F}\, DISJ \\
DISJ == \mathbb{F}_1\, BEXP
\end{array}
$$

A binary expression relates a variable (*VAR*) with a literal (*VALUE*) by means of an operator (*OP*), which can be *less than or equal to* (*le*), *less than* (*lt*), *equal to* (*eq*), *greater than* (*gt*), and *greater than or equal to* (*ge*).

$$
\begin{array}{l}
BEXP == \{v : VAR;\ op : OP;\ literal : VALUE\} \\
OP ::= le \mid lt \mid eq \mid ne \mid gt \mid ge
\end{array}
$$

The element *VAR* refers to the current or previous value of the corresponding variable. By previous value we mean the last value received as input, if it refers to an input variable, or the last value produced as output, otherwise.

$$VAR ::= current\langle\!\langle VNAME \rangle\!\rangle \mid previous\langle\!\langle VNAME \rangle\!\rangle$$

Timers are variables continuously evolving in a discrete or dense fashion, depending on their type and, thus, the notion of previous value does not necessarily apply. For instance, what would be the previous value of a timer whose current value is 3.52 seconds? So, although the model syntactically permits retrieving the previous values of timers, we prohibit this usage (see the following definition of *var_consistent_be*).

A binary expression is said to be consistent with respect to a set of variables (f) and a set of timers (T) if, and only if, v (the first element of a binary expression) refers to a variable name (n) within f, and the third element (*literal*) is consistent with the type of the corresponding variable (it is one of the possible values of this variable). Moreover, if one of the operators *le*, *lt*, *gt* or *ge* is used, *literal* must not be a boolean value. Finally, as explained in the last paragraph, if n is the name of a timer, then the binary expression must consider the current value of this variable. This consistency property is formalised by the following predicate.

$$
\begin{array}{|l}
var\_consistent\_be : \mathbb{P}(BEXP \times (VNAME \pfun TYPE) \times (VNAME \pfun TYPE)) \\
\hline
\forall\, be : BEXP;\ f, T : VNAME \pfun TYPE;\ n : VNAME \mid varName(be) = n \bullet \\
\quad (be, f, T) \in var\_consistent\_be \Leftrightarrow \\
\quad (n \in \dom f) \land be.3 \in values(f(n)) \land \\
\quad (be.2 = le \lor be.2 = lt \lor be.2 = gt \lor be.2 = ge \Rightarrow be.3 \notin values(bool)) \land \\
\quad (n \in \dom T \Rightarrow be.1 \in \ran current)
\end{array}
$$

To get the name referenced by a binary expression, we rely on the auxiliary function *varName*, which projects the *VNAME* within the constructors *current* or *previous*. This concept of consistency is lifted to guards, which are said to be consistent if, and only if, all of its binary expressions are consistent. The last component of a function entry is a finite and non-empty set of assignments (*ASGMTS*). The right-hand side of an assignment (*ASGMT*) is a value (*VALUE*), and the left-hand side is the name of a variable (*VNAME*).

$$
\begin{array}{l}
ASGMT == VNAME \times VALUE \\
ASGMTS == \\
\quad \{asgmts : \mathbb{F}_1\, ASGMT \mid (\forall\, asgmt1, asgmt2 : asgmts \mid asgmt1.1 = asgmt2.1 \bullet asgmt1 = asgmt2)\}
\end{array}
$$

Note that it is not possible to define a set of assignments that considers different values to the same variable (e.g., $\{(x, n(0)), (x, n(1))\}$). If such a scenario were allowed, it would not be clear what would be the value of x after the assignments.

This restriction does not prevent us from dealing with non-deterministic requirements. For example, it is possible to say that the system can non-deterministically assign 0 or 1 to x in a certain situation. In this case, we would have two entries within the function, and the set of assignments of one would be $\{(x, n(0))\}$, whereas $\{(x, n(1))\}$ would be the assignment of the other one. Note that the property defined here with respect to assignments can be statically verified, whereas the verification of non-deterministic requirements demands a dynamic analysis.

As defined for expressions, the names mentioned by assignments should refer to one of the system variables, and the assigned value should be consistent with the type of this variable. The following predicate (*well_typed_asgmts*) formalises these assumptions.

$$
\begin{array}{|l}
well\_typed\_asgmts : \mathbb{P}(ASGMTS \times (NAME \pfun TYPE)) \\
\hline
\forall\, asgmts : ASGMTS;\ f : NAME \pfun TYPE \bullet (asgmts, f) \in well\_typed\_asgmts \Leftrightarrow \\
\quad \forall\, asgmt : asgmts \bullet asgmt.1 \in \dom f \land asgmt.2 \in values(f(asgmt.1))
\end{array}
$$

Example 3 Considering the VM, the requirement that states that the system goes to the *choice* mode, and resets the *request timer*, when a coin is inserted while in the state *idle*, is formalised as follows:

$$
\begin{array}{c}
\{\ (\{\ \{(current(m), eq, n(1))\},\ \{(current(s), eq, b(true))\},\ \{(previous(s), eq, b(false))\}\ \}, \emptyset, \\
\{(m, n(0), (r, n(0))\})\ \}
\end{array}
$$

The static condition is a conjunction of three binary expressions. The first denotes that the system mode is 1 (*idle*), the second that the current value of s is *true*, and the third that the previous value of s was *false* (a coin was inserted). The time guard is empty, and when the static guard evaluates to true, the system shall assign 0 to m (go to the *choice* state), and assign 0 to r (reset the request timer). □

2.2.4. Complete definition of an s-DFRS

Considering the schemas *DFRS_VARIABLES*, *DFRS_INITIAL_STATE*, and *DFRS_FUNCTIONS*, previously defined, an s-DFRS is defined as follows.

s_DFRS

$$
\begin{array}{l}
DFRS\_VARIABLES \\
DFRS\_INITIAL\_STATE \\
DFRS\_FUNCTIONS \\
\hline
(s_0, I \cup O \cup T \cup \{gcvar\}) \in well\_typed\_state \\
\forall f : F \bullet \forall entry : f \bullet \\
\quad (entry.1, I \cup O, T) \in var\_consistent\_exp \land \\
\quad (entry.2, T, T) \in var\_consistent\_exp \land \\
\quad (entry.3, O \cup T) \in well\_typed\_asgmts
\end{array}
$$

The schema *s_DFRS* defines a type that comprises the set of all valid s-DFRSs. Two invariants hold for all valid elements of this type. An invariant is a constraint that must always be satisfied. First, the initial state is well typed with respect to all system variables. Second, for all entries of all functions defined, the static guard is defined only in terms of input and output variables, the timed guard only considers timers, and the assignments can only modify the value of outputs and timers.

3. Formalising natural-language requirements

Before explaining how an s-DFRS can be generated from natural-language requirements, we present information about the first two phases of the NAT2TEST strategy (see Fig. 1) that is essential for understanding the generation of s-DFRSs.

3.1. Syntactic and semantic analyses

The syntactic analysis checks whether the system requirements are correct with respect to the SysReq-CNL grammar, yielding the corresponding syntax trees if they are. Briefly, this Controlled Natural Language (CNL) allows writing requirements that have the form of action statements guarded by conditions. For example, consider the REQ001 requirement for the VM, which is correct according 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.*

The semantic analysis phase receives as input the generated syntax trees, and delivers a requirement semantic representation. In this work, we adopt the Case Grammar theory [Fil68] to represent meaning. In this theory, a sentence is analysed in terms of the semantic (Thematic) Roles (TR) played by each word, or group of words in the sentence. The verb is the main element of the sentence, and it determines the possible semantic relations with other words in the sentences, that is, the role that each word plays with respect to the action or state described by the verb.

The verb's associated TRs are aggregated into a structure named as Case Frame (CF). Each verb in a requirement specification gives rise to a different CF. All derived CFs are joined afterwards to compose what we call a *Requirement Frame* (RF). In other words, a requirement frame is a structure to encode data such as the one presented in Table 1: a collection of case frames for conditions and action statements.

In this work, we consider four TRs (the adopted nomenclature is inspired by [All95]) for the verbs used in action statements: action (ACT) – the action performed if the requirement conditions are satisfied; agent (AGT) – entity who performs the action; patient (PAT) – entity who is affected by the action; TOV – the patient value after action completion. Similarly, other five roles are defined for the verbs used in conditions: condition action (CAC), condition patient (CPT), condition from value (CFV), condition to value (CTV), and condition modifier (CMD).

Table 1. Example of requirement frame for REQ001

Condition #1 - Main Verb (CAC): is			
CPT:	the system mode	CFV:	-
CMD:	-	CTV:	idle
Condition #2 - Main Verb (CAC): changes			
CPT:	the coin sensor	CFV:	-
CMD:	-	CTV:	true
Action #1 - Main Verb (ACT): reset			
AGT:	the coffee machine system	TOV:	-
PAT:	the request timer		
Action #2 - Main Verb (ACT): assign			
AGT:	the coffee machine system	TOV:	choice
PAT:	the system mode		

Based on the inference rules defined in [CFB+14], each word, or group of words, identified in the syntax tree is associated to the corresponding TR. For instance, Table 1 shows the requirement frame corresponding to REQ001. The requirement frames obtained from the system requirements are the input for the generation of s-DFRSs.

An s-DFRS is derived from requirement frames according to three consecutive steps. First, the system variables are identified. Then, the functions that describe the system behaviour are defined. Finally, an s-DFRS is created from these two pieces of information. The following sections detail each step.

3.2. Identifying variables

We consider inputs as variables provided to the system by the environment; their values cannot be modified by the system. Thus, a variable is classified as an input if, and only if, it appears only in conditions. Otherwise, if it also appears in action statements, it is classified as an output. To distinguish between timers and other variables, we require the former to have the word "timer" as a suffix. Timers can appear both in conditions and in statements.

Our algorithm for identifying variables (Algorithm 1) receives as input a list of requirement frames and yields a list of variables.

After initializing the output (Line 1), the algorithm iterates over the list of requirement frames (Line 2) analysing each condition (Lines 3–4), which comprises a conjunction of disjunctions, and each action (Line 15). When analysing conditions, we extract variables from the *Condition Patient* (CPT) role.

For example, Table 1 shows that *"the system mode"* is the CPT of the first condition. Thus, if the corresponding variable has not yet been identified (Lines 6–7), we create a new variable considering the CPT content, replacing white spaces by an underscore (Lines 8–9), which is done by the *"toString"* function (Line 5). So, in this case, we create the variable *the_system_mode*. Then, we verify whether the variable has the word *"timer"* as a suffix; if so, it is classified as a timer, otherwise it is an input (Lines 10–11). Then we add the created variable to the list of identified variables (Line 12).

To infer the type of the variable we analyse the value associated with it in the case frame, which is the content of the CTV role. For instance, the variable *the_system_mode* is associated with the value *"idle"* in the first condition of Table 1. Thus, the algorithm extracts the CTV content (Line 13), and uses it to infer the variable type, which is done by the *inferType* function (Line 14) that is later explained (see Algorithm 2).

Lines 15–27 are analogous to those previously explained. The differences are as follows:

- The variables are identified from the Patient (PAT) role;
- If a variable that is initially identified as an input appears in action statements, it is reclassified as an output (Lines 24–25);

Algorithm 1: Identify Variables

```
   input  : reqCFList
   output : varList
 1 varList = new List();
 2 for reqCF ∈ reqCFList do
 3     for andCond ∈ reqCF do
 4         for orCond ∈ andCond do
 5             varName = toString(orCond.CPT);
 6             var = varList.find(varName);
 7             if var == null then
 8                 var = new Var(varName);
 9                 var.type = undefined;
10                 if varName.endsWith("timer") then var.kind = timer;
11                 else var.kind = input;
12                 varList.add(var);
13             value = toString(orCond.CTV);
14             inferType(var, value, varList);
15     for action ∈ reqCF do
16         varName = toString(action.PAT);
17         var = varList.find(varName);
18         if var == null then
19             var = newVar(varName);
20             var.type = undefined;
21             if varName.endsWith("timer") then var.kind = timer;
22             else var.kind = output;
23             varList.add(var);
24         else if var.kind = input then
25             var.kind = output;
26         value = toString(action.TOV);
27         if value ≠ null then inferType(var, value, varList);
28 for var ∈ varList do
29     if var.type = enum then
30         if var.possibleValuesList.size() = 1 then var.type = boolean;
31         else var.type = integer;
32     else if var.kind = timer ∧ var.type = undefined then
33         var.type = float
34 gcVar = newVar(gc);
35 allDiscrete = true;
36 allContinuous = true;
37 for var ∈ varList do
38     if var.kind = timer ∧ var.type = integer then allContinuous = false;
39     if var.kind = timer ∧ var.type = float then allDiscrete = false;
40 if allDiscrete then gcVar.type = integer;
41 else if allContinuous then gcVar.type = float;
42 else throw Exception("timers: incompatible types");
43 varList.add(gcVar);
```

- The variable value is the content of the TOV role, excluding the case (Line 27) when the *"reset"* verb is used (see the first action of Table 1). In this case, the TOV is empty and what is assigned to the timer is the system global clock, which is an integer or a float. In such a situation, we do not try to infer the type of the timer. If this timer is also mentioned in a condition, its type is determined by the value associated with it in this condition. If this timer is never mentioned within a condition, its type is left undefined (Lines 32–33), and then we assume that its type will be float, representing continuous time.

Lines 34–43 create the system global clock (gc), besides inferring its type. If all timers are discrete (integer) or continuous (float), the type of gc is integer or float, respectively. If there are mixed types, an exception is thrown (Line 42). Finally, Lines 29–31 are related to the type inference outcome, which is explained in what follows.

Algorithm 2 infers the variable type. First, this function verifies whether the value received as argument is already listed as a possible value of the corresponding variable (Line 1). If not, this value is added to the list of possible values of the respective variable (Line 2), and this value is used to infer the variable type.

Algorithm 2: Infer Type

```
   input  : var, value, varList
   output : –
 1 if value ∉ var.possibleValuesList then
 2     outVar.possibleValuesList.add(value);
 3     newType = undefined;
 4     var = varList.find(varName);
 5     if var.kind = timer then
 6         if isFloat(value) then newType = float;
 7         else if isInteger(value) then newType = integer;
 8         else throw Exception("incompatible type for a timer");
 9     else
10         if isBoolean(value) then newType = boolean;
11         else if isFloat(value) then newType = float;
12         else if isInteger(value) then newType = integer;
13         else newType = enum;
14     if var.type ≠ undefined ∧ var.type ≠ newType then throw Exception("type change is not allowed");
15     else var.type = newType;
```

If the variable is a timer, the associated values need to be numbers (float or integer), otherwise an exception is raised (Lines 5–8). If the variable is an input or an output, its type might be boolean (if the value is the boolean constants "true" or "false" – Line 10), a float or an integer (Lines 11-12), or an *enum* (e.g., if the value is a string such as "idle" – see Table 1). It is worth mentioning that the *enum* type is not expected within DFRS models. Therefore, later it is mapped to an integer.

If the type of the variable is undefined, the function assigns the inferred type to the corresponding variable (Line 15). However, if the variable already has a type, it is verified whether the inferred type from the current value is the same. If not, an exception is raised, since we expect type coherence between the values used with respect to the same variable (Line 14). In other words, for instance, a variable cannot be treated as a boolean and as an integer simultaneously.

Finally, Lines 29–31 of Algorithm 1 map an *enum* type to a boolean or an integer. It is mapped to a boolean when the enumeration has only one possible value (Line 30). For instance, for the variable *the_coffee_request_button*, whose possible value is *"pressed"*, we assume that *"pressed"* denotes *true*, whereas *"not pressed"* means *false*. However, if the number of possible values is greater than 1, the variable is classified as an integer (Line 31). This is the case of the variable *the_system_mode*, whose possible values are *"choice", "idle", "preparing strong coffee", "preparing weak coffee", "reset"*. The type of this variable is integer considering the following mapping: $\{0 \mapsto$*choice*, $1 \mapsto$*idle*, $2 \mapsto$*preparing strong coffee*, $3 \mapsto$*preparring weak coffee*, $4 \mapsto$*reset*$\}$.

3.3. Identifying functions

Algorithm 3 identifies functions that describe the system behaviour. We identify one function for each different Agent (AGT). We consider an agent as a system component, since this thematic role denotes the entity that performs an action. This algorithm yields a list of functions indexed by the corresponding agents. As previously formalised, each function is a list of action statements mapped to the respective static and timed guards.

The algorithm iterates over the list of requirement frames (Line 2) to identify the guards (Lines 3–24) and the corresponding actions (Lines 25–28). The variables *staticGuard* and *timedGuard* are declared to store the static and timed guards that are extracted from the conditions (conjunctions of disjunctions) of each requirement (Lines 3–7). Then, for each disjunction, we obtain the corresponding boolean expression by means of the function *generateConditionExpression* (Line 8). Then, Lines 9–16 find out the type (static or timed) of the expression. If the expression concerns a timer variable, it represents a timed guard (Line 12), otherwise it is a static one (Line 15).

Algorithm 3: Identify Functions

```
   input  : reqCFList, varList
   output : functionMap
 1 functionMap = new Map();
 2 for reqCF ∈ reqCFList do
 3     staticGuard, timedGuard = null;
 4     for andCond ∈ reqCF do
 5         guardType = undefined;
 6         newTerm = null;
 7         for orCond ∈ andCond do
 8             exp = generateConditionExpression(orCond, varList);
 9             varName = toString(orCond.PAT);
10             var = varList.find(varName);
11             if var.kind = timer then
12                 if guardType = undefined then guardType = timed;
13                 else if guardType = static then throw Exception("format error");
14             else
15                 if guardType = undefined then guardType = static;
16                 else if guardType = timed then throw Exception("format error");
17             if newTerm = null then newTerm = exp;
18             else newTerm = newTerm + "∨" + exp;
19         if guardType = static then
20             if staticGuard = null then staticGuard = (newTerm);
21             else staticGuard = staticGuard + "∧" + (newTerm);
22         else
23             if timedGuard = null then timedGuard = (newTerm);
24             else timedGuard = timedGuard + "∧" + (newTerm);
25     actionList = new List();
26     for action ∈ reqCF do
27         actionStatement = generateStatement(action, varList);
28         actionList.add(actionStatement);
29     componentName = toString(reqCF.actions.get(0).AGT);
30     function = functionMap.find(componentName);
31     if foundFunction = null then
32         function = new Function();
33         functionMap.add(componentName, function);
34     previousActionList = function.find(staticGuard, timedGuard);
35     if previousActionList ≠ null then
36         previousActionList.add(actionList)
37     else
38         function.add(staticGuard, timedGuard, actionList)
```

With this information, we check whether each conjunction concerns the same type of guards (static or timed). If it is not the case, an exception is raised (Lines 13, 16). This is necessary, since we want to divide the conditions into two disjoint categories (static and timed) without performing boolean algebra manipulation. As examples, we consider the following abstract cases: $c_1 : T \wedge (c_2 : T \vee c_3 : S) \wedge c_4 : S$ and $c_1 : T \wedge (c_2 : S \vee c_3 : S) \wedge c_4 : S$, where c_i denotes the i-th condition, and "$:S$" and "$:T$" indicates whether the condition concerns a static or a timed guard, respectively. The first expression does not comprise two disjoint sets of static and timed guards, whereas the second one does (timed: c_1; static: $(c_2 \vee c_3) \wedge c_4$). Lines 17–18 group each disjunction in *newTerm*, and Lines 19–24 group the disjunctions in *staticGuard* or *timedGuard* depending on the type of the disjunctions.

After identifying the static and timed guards, the algorithm iterates over the list of actions of the requirement frame and creates a list of action statements using the function *generateStatement* (Lines 25–28). Then, the algorithm checks whether a function is already created for the current agent. If not, it creates a new function and maps it to the current agent (Lines 29–33). Finally, the element *staticGuard* × *timedGuard* × *actionList* is added to the corresponding function (Lines 37–38). If an entry for the pair

staticGuard × *timedGuard* already exists, the list of actions is added to this entry (Lines 35–36). In what follows, we explain the auxiliary functions: *generateConditionExpression* and *generateStatement*.

3.3.1. Generating condition expressions

Algorithm 4 yields a boolean expression from a single case frame, which comprises the condition thematic roles. The variable name is obtained from the CPT role (Line 1). Initially (Lines 2–7), the algorithm verifies whether the verb being used, which is obtained from the CAC role, denotes the previous value of a variable. This is the case when the verbs *"was"* and *"were"* are used. In this situation, the boolean expression concerns not the current value of a variable, but its previous one. As explained in Section 2.2.3, we use the predicate $previous(v)$ to denote the previous value of v, and $current(v)$, to denote its current value.

For instance, the fragment *"v was 2"* means the condition where the previous value of v is x, $previous(v) = 2$. As we do not allow the use of the predicate $previous(v)$ when v is a timer, the algorithm raises an exception if it happens (Line 5).

The next step is to obtain the value, which is compared to the variable. First, the value is obtained from the CTV role (Line 8). If the value is a string, we consider as value the index of this string within the list of possible values of the corresponding variable (Line 9). For a concrete example, see the one shown in the end of Section 2.2.2.

Afterwards, the algorithm inspects the content of the CMD role to find out which operator is used in the expression (Line 10). Lines 11–16 check the content of the CMD role, and set boolean flags accordingly. If *"lesser than"* or *"greater than"* is used with a non-boolean variable, an exception is raised (Line 16). Based on the boolean flags, Lines 17–25 assign to *operator* the operator symbol used in the expression.

After that, it creates the expression assembling these three elements: variable, operator, and value (Line 26). Line 27 negates the expression if the *negation* flag is true: when *"not"* is used as a modifier.

Finally, Lines 28–45 deal with a special case that occurs when the verbs *"change"* or *"become"* are used. When *"change"* is used, as explained in depth in [CFB+14], we expect one of the two following structures: *"v changes from x to y"* or *"v changes to y"*, whose meaning is $previous(v) = x \wedge current(v) = y$ and $previous(v) \neq y \wedge current(v) = y$, respectively. In the first case, the CFV is not *null*, whereas in the second case it is *null*. It is important to note that the expression $current(v) = y$ is already built by the algorithm (denoted as *exp*). Therefore, we just need to create a second condition expression related to the previous value of v. Lines 30–44 create a temporary and auxiliary case frame with the verb *"was"*, which enforces the use of $previous(v)$, and then we recursively call the function *generateConditionExpression*. If CFV is not null (Lines 30–35), e.g., *"changes from x to y"*, the CTV in the auxiliary case frame comprises the current CFV (x), otherwise (e.g., *"changes to y"*) it is the negation of the current CTV (y). After that, we compose the yielded expression (*previousExp*) with the expression previously identified by the algorithm (*exp*) (Line 45). When the verb *"become"* is used (e.g., *"becomes y"*), the algorithm behaves similarly to the case *"changes to y"*.

This algorithm is tightly dependent on the verbs used. However, the verbs currently supported by our approach are sufficient to express requirements from different examples and domains. If more verbs are used, one just needs to extend this function, informing how to form an expression from its thematic roles. No extra change is needed.

3.3.2. Generating action statements

Algorithm 5 generates an action statement from a case frame that depicts an action. First, Lines 1–3 retrieve the verb from the ACT role, as well as the name of the variable involved in the action from the PAT role. If the variable is a timer and the verb is not reset, an exception is raised, since timers can only be reset (Line 4).

The next step concerns the identification of the value being assigned to the involved variable (Lines 5–9). If the verb is *"reset"*, the value that is assigned to the timer is 0 or 0.0, depending on its type (integer or float). As already mentioned, and detailed when describing how an s-DFRS is used to produce an expanded one in Section 4.2, this assignment actually means assigning to the timer the system global clock. If the variable is not a timer, the value is the content of the TOV role (Line 8). If the content of TOV is not an integer, a float or a boolean, it is a string. Therefore, we consider as value the index of this string within the list of possible values of the corresponding variable (Line 9). Finally, the action statement is created assembling the variable and the assigned value (Lines 10–11).

Algorithm 4: Generate Condition Expression

```
   input  : cond, varList
   output : exp
 1 varName = toString(cond.CPT);
 2 var = varList.find(varName);
 3 verb = cond.CAC;
 4 if verb.equals("was") ∨ verb.equals("were") then
 5 |   if var.kind = timer then throw Exception("previous cannot be used with timers");
 6 |   else varName ="previous(" + varName + ")";
 7 else varName ="current(" + varName + ")" ;
 8 value = toString(cond.CTV);
 9 if ¬ isInteger(value) ∧ ¬ isFloat(value) ∧ ¬ isBoolean(value) then value = var.possibleValuesList.getIndex(value) ;
10 modifier = cond.CMD;
11 negation, lesserThan, greaterThan, equalTo = false;
12 if modifier.contains("not") then negation = true;
13 if modifier.contains("lesser than") then lesserThan = true;
14 if modifier.contains("greater than") then greaterThan = true;
15 if modifier.contains("equal to") then equalTo = true;
16 if (lowerThan ∨ greaterThan) ∧ var.type = boolean then throw Exception("lt/le/gt/ge cannot be used with booleans") ;
17 operator = new String();
18 if lesserThan then
19 |   if equalTo then operator ="le";
20 |   else operator ="lt";
21 else if greaterThan then
22 |   if equalTo then operator ="ge";
23 |   else operator ="gt";
24 else
25 |   operator ="eq";
26 exp = varName + operator + value;
27 if negation then exp ="¬ (" + exp + ")" ;
28 if verb.contains("change") ∨ verb.contains("become") then
29 |   prevExp = null;
30 |   if cond.CFV ≠ null then
31 |   |   auxiliaryCond = new OrCond();
32 |   |   auxiliaryCond.CPT = cond.CPT;
33 |   |   auxiliaryCond.CAC ="was";
34 |   |   auxiliaryCond.CTV = cond.CFV;
35 |   |   previousExp = generateConditionExpression(auxiliaryCond);
36 |   else
37 |   |   auxiliaryCond = new OrCond();
38 |   |   auxiliaryCond.CPT = cond.CPT;
39 |   |   auxiliaryCond.CAC ="was";
40 |   |   auxiliaryCond.CTV = cond.CTV;
41 |   |   auxiliaryCond.CMD = cond.CMD;
42 |   |   previousExp = generateConditionExpression(auxiliaryCond);
43 |   |   previousExp = ¬ previousExp;
44 |   |   previousExp ="¬ (" + previousExp + ")";
45 |   exp = prevExp + "∧" + exp;
```

3.4. Creating an s-DFRS

Based on the algorithms previous described, we create an s-DFRS from a list of requirement frames. This is done by Algorithm 6. First, the algorithm calls *identifyVariables* to identify the system variables (Line 1). Then, it divides this list into inputs, outputs, timers, and the global clock (Lines 2–9).

This algorithm also creates an initial binding considering *0* as the initial default value for *integers*, *0.0* for *floats*, and *false* for *booleans* (Lines 10–12). Afterwards, the algorithm calls *identifyFunctions* to identify the functions that describe the system behaviour (Line 13). In the end (Lines 14–20), the algorithm creates an s-DFRS considering the list of inputs, outputs and timers, as well as the initial binding and the functions identified.

Algorithm 5: Generate Statement

```
input  : action, varList
output : actionStatement
1  verb = action.ACT;
2  varName = toString(action.PAT);
3  var = varList.find(varName);
4  if var.kind = timer ∧ ¬ verb.equals("reset") then throw Exception("timers can only be reset") ;
5  value = null;
6  if verb.equals("reset") ∧ var.type = integer then value ="0";
7  else if verb.equals("reset") ∧ var.type = float then value ="0.0";
8  else value = toString(action.TOV);
9  if ¬ isInteger(value) ∧ ¬ isFloat(value) ∧ ¬ isBoolean(value) then value = var.possibleValuesList.getIndex(value) ;
10 actionStatement = new Statement();
11 actionStatement = varName + ":=" + value;
```

Algorithm 6: Derive s-DFRS

```
input  : reqCFList
output : dfrs
1  varList = identifyVariables(reqCFList);
2  inputList, outputList, timerList = new List();
3  gc = null;
4  initialBinding = new Map();
5  for var ∈ varList do
6      if var.kind = input then inputList.add(var);
7      else if var.kind = output then outputList.add(var);
8      else if var.kind = timer then timerList.add(var);
9      else gc = var;
10     if var.type = integer then initialBinding.add(var.name, 0);
11     else if var.type = float then initialBinding.add(var.name, 0.0);
12     else initialBinding.add(var.name, false);
13 functionMap = identifyFunctions(reqCFList, varList);
14 dfrs = new s_DFRS();
15 dfrs.I = inputList;
16 dfrs.O = outputList;
17 dfrs.T = timerList;
18 dfrs.s0 = initialBinding;
19 dfrs.gcvar = gc;
20 dfrs.F = functionMap;
```

3.5. Tool support

The algorithms presented here are implemented in the NAT2TEST tool. It is written in Java (it is multi-platform), and its Graphical User Interface (GUI) is built using the Eclipse RCP[8] framework, which provides means to create client-side applications quickly using a collection of plug-ins.

Each phase of the NAT2TEST strategy (see Fig. 1), is realised by a different component. The DFRS-Generator component is the one that implements the above algorithms. Fig. 5 shows the inferred variables, along with their types for our vending machine. The tool also allows the user to edit the initial values and, thus, the initial state.

In Fig. 6 one can see part of the function obtained from the VM requirements. It is important to note that the tool keeps traceability information between the requirements and the function entries. We also note that there are some syntactic sugars to prevent a verbose representation. For instance, *previous* is reduced to *prev*, and *current* is simply hidden. Moreover, the format of the timed guard, as well as the assignment of timers, show explicitly how timers are dealt with by our strategy: the reset of a timer is encoded as assigning the value of *gc* to the timer, and comparisons concerning the timer mean comparing the difference between the current value of *gc* and the timer. More details are provided in Section 4.2, where we explain how to obtain an e-DFRS from a symbolic one.

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

Kind	Name	Type	Expected Values	Initial Value
INPUT	the_coffee_request_button	BOOLEAN	{false, true}	false
INPUT	the_coin_sensor	BOOLEAN	{false, true}	false
OUTPUT	the_system_mode	INTEGER	{choice, idle, preparing strong coffee, preparing weak coffee}	idle
OUTPUT	the_coffee_machine_output	INTEGER	{strong, weak}	strong
TIMER	the_request_timer	FLOAT	-	0.0
GLOBAL CLOCK	gc	FLOAT	-	0.0

Fig. 5. NAT2TEST tool – editing initial value of DFRS variables

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 <= 30.0	the_request_timer := gc, the_system_mode := 3	REQ002

Fig. 6. NAT2TEST tool – viewing DFRS' functions and traceability information

The tool also supports validation of the requirements by animating the s-DFRS; in other words, by manually exploring the state space of the corresponding e-DFRS. In Section 4.5 we detail this feature. In [CBC+15] we provide a comprehensive explanation of other aspects of the NAT2TEST tool. Particularly, we emphasise that, for all examples considered, the s-DFRS models are generated from the corresponding natural-language requirements within 1 second. The NAT2TEST tool, as well as the examples that are public, can be downloaded in `http://www.cin.ufpe.br/~ghpc/`.

4. Definition and properties of an e-DFRS

Here, we formalise e-DFRSs (Section 4.1), and show how they can be obtained from their symbolic counterpart (Section 4.2) via a sound process (Section 4.3). Moreover, we describe how an e-DFRS can be used to verify properties of the system requirements, such as consistency, completeness and reachability (Section 4.4). We also describe here the support provided by the NAT2TEST tool with respect to e-DFRS models (Section 4.5).

4.1. Formal model of an e-DFRS

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. As we detail later, states are obtained from an s-DFRS by applying its functions to states where the corresponding guards evaluate to true, but also letting the time evolve.

4.1.1. Transition relation

An e-DFRS has a set of states, which is named S by the schema $DFRS\_STATES$ below. Besides that, it also has an initial state (s_0), which is an element of S. We note that, by definition, S has at least one state (the initial state), since it is an element of $STATES$, which represents the non-empty power set of $STATE$.

$$STATES == \mathbb{P}_1\, STATE$$
$$DFRS\_STATES == [\, S : STATES;\ s_0 : STATE \mid s_0 \in S \,]$$

A transition relation (an element of $TRANSREL$ defined below) comprises a set of transitions ($TRANS$). A transition relates two states by a label ($TRANS\_LABEL$). As shown in Fig. 4, this label can be of a *delay*

(*del*) or a *function* (*fun*) transition.

$$TRANS\_LABEL ::= fun\langle\langle ASGMTS\rangle\rangle \mid del\langle\langle DELAY \times ASGMTS\rangle\rangle$$
$$TRANS == (STATE \times TRANS\_LABEL \times STATE)$$
$$TRANSREL == \mathbb{P}\, TRANS$$

A function transition represents the system instantaneous reaction as a set of assignments (*ASGMTS*), which are performed atomically. It is worth noting that, although the function transition describes an instantaneous reaction, it is possible to model system reactions that occur after some time elapsing. We just need to consider a timer, which is reset when the event of interest happens, and then use it later to check the elapsed time and to decide on what event to engage next. For instance, this approach is used in the VM example (see Fig. 4). When the coffee request button is pressed, the request timer is reset (see the first state on the second row). Afterwards, when a specific time has elapsed, the system reacts producing coffee (see the last state on the third row).

A delay transition represents model stimuli from the environment (input signals values) that happen immediately after a delay (*DELAY*). We note that environment stimuli are modelled as a set of assignments (*ASGMTS*). A delay can represent a discrete or dense (continuous) time elapsing. The former delay is characterised by a positive natural number ($\mathbb{N}_1$), whereas the latter by a positive float number (R_1^+).

$$DELAY ::= discrete\langle\langle \mathbb{N}_1\rangle\rangle \mid dense\langle\langle R_1^+\rangle\rangle$$

The reason for not allowing delays equal to 0 is that the delay transition represents interaction with the environment and, thus, it is not reasonable to assume that the environment can interact with the system, providing it with new stimuli, without time elapsing.

Aiming at legibility, we define two auxiliary functions (*functionTransition* and *delayTransition*), which project the elements of a transition. The definition of *delayTransition* is shown below. It is a partial function, since it can only be applied to delay transitions; its domain is equal to the set of valid delay transitions ($\mathrm{dom}\, delayTransitions = \mathrm{ran}\, del$). To obtain the delay and assignments embedded in a delay transition (denoted by *label* below), we use the inverse definition of the constructor *del* ($del^\sim$), which yields a pair of delay and assignments ($DELAY \times ASGMTS$) from a given delay transition (*label* below). The inverse of *del* is well defined, since, by definition in Z, all constructors are defined as injections. Therefore, $del^\sim$ exists, since the inverse of an injection is also a function. The function *functionTransition* is defined similarly.

$$delayTransition : TRANS\_LABEL \pfun DELAY \times ASGMTS$$
$$\mathrm{dom}\, delayTransition = \mathrm{ran}\, del$$
$$\forall\, label : TRANS\_LABEL \mid label \in \mathrm{ran}\, del \bullet delayTransition(label) = (del^\sim)(label)$$

All transitions of an e-DFRS are required to be well typed: a function transition must belong to the set of well typed function transitions (*well_typed_function_transition*), while a delay transition must belong to the analogous set (*well_typed_delay_transition*), besides being compatible with the type of the system global clock (*clock_compatible_transition*).

To be well typed, a function transition must modify only values of outputs and timers. In other words, the system does not interfere with the environment stimuli, which are modelled by input variables. This property is formalised by *well_typed_function_transition* when stating that the domain of *functionTransition* is a subset of or equal to the union of the domains of O and T. The outputs and timers that are not changed by the transition retain the same value.

$$well\_typed\_function\_transition : \mathbb{P}(TRANS\_LABEL \times (VNAME \pfun TYPE) \times (VNAME \pfun TYPE))$$
$$\forall\, label : TRANS\_LABEL;\ O, T : VNAME \pfun TYPE \mid label \in \mathrm{ran}\, fun \bullet (label, O, T) \in well\_typed\_function\_transition \Leftrightarrow (\mathrm{dom}(functionTransition(label)) \subseteq (\mathrm{dom}\, O \cup \mathrm{dom}\, T))$$

Similarly, a delay transition is well typed if, and only if, its statements modify only values of inputs. Furthermore, there must be one statement concerning each input; on the occurrence of each delay transition, the system receives the current value of all its inputs. The predicate *well_typed_delay_transition* formalises these two requirements when stating that the domain of *delayTransition* is equal to the domain of I.

$$well\_typed\_delay\_transition : \mathbb{P}(TRANS\_LABEL \times (VNAME \nrightarrow TYPE))$$

$$\begin{aligned}
&\forall\, label : TRANS_LABEL;\ I : VNAME \nrightarrow TYPE \mid label \in \operatorname{ran} del \bullet \\
&\quad (label, I) \in well_typed_delay_transition \Leftrightarrow \\
&\quad \operatorname{dom}(delayTransition(label)).2 = \operatorname{dom} I
\end{aligned}$$

One might find strange that here we expect the assignments to range over all inputs, whereas the function transition can cover only a subset of its outputs. We could have also assumed here that the inputs that are not mentioned by the assignments retain the same value. However, this modelling decision would make the translation from s-DFRSs to e-DFRSs more complicated. We return to this topic later, when explaining how e-DFRSs are obtained from symbolic ones.

The delay transitions also need to be compatible with the system global clock in the sense that if the delay is discrete (an element of ran *discrete*), the type of the system global time must be *nat*, whereas if the delay is dense (an element of ran *dense*), the type of the clock must be *ufloat*. As a consequence, all delay transitions share the same type of delay, meaning that they are all discrete or dense. This is captured by the *clock_compatible_transition* property.

$$clock\_compatible\_transition : \mathbb{P}(TRANS\_LABEL \times (NAME \times TYPE))$$

$$\begin{aligned}
&\forall\, label : TRANS_LABEL;\ gcvar : NAME \times TYPE \bullet \\
&\quad (label, gcvar) \in clock_compatible_transition \Leftrightarrow \\
&\quad label \in \operatorname{ran} del \land \\
&\quad ((delayTransition(label)).1 \in \operatorname{ran} discrete \Rightarrow gcvar.2 = nat) \land \\
&\quad ((delayTransition(label)).1 \in \operatorname{ran} dense \Rightarrow gcvar.2 = ufloat)
\end{aligned}$$

Now, we define in the schema *DFRS_TRANSITION_RELATION* the transition relation (*TR*) of an e-DFRS as an element of *TRANSREL*. As previously said, we assume that when the system is ready to react it does so instantaneously. Therefore, it would not make sense to have both delay and function transitions from the same state, since the system always reacts (performing the function transition), instead of letting the time evolve (performing the delay transition). This invariant is formalised in what follows by the first predicate: for every two transitions (*trans*1 and *trans*2) emanating from the same state (*trans*1.1 = *trans*2.1), they are either function (they belong to ran *fun*) or delay transitions (they belong to ran *del*).

DFRS_TRANSITION_RELATION

$$TR : TRANSREL$$

$$\begin{aligned}
&\forall\, trans1, trans2 : TR \mid trans1.1 = trans2.1 \bullet \\
&\quad \{trans1.2, trans2.2\} \subseteq \operatorname{ran} fun \lor \{trans1.2, trans2.2\} \subseteq \operatorname{ran} del \\
&\forall\, trans : TR \bullet \neg\, (trans.1 = trans.3)
\end{aligned}$$

Another invariant associated with *TR* is the absence of self-transitions: for all transitions, $\neg\ (trans.1 = trans.3)$ holds. In the case of delay transitions, self-transitions do not make sense as every delay transition advances the time by some amount greater than 0 and, thus, the global clock of the next state is different from the previous one. Concerning function transitions, self-transitions are superfluous, since the absence of function transitions already indicates that the system state has not changed.

For a concrete example, we refer to the second delay transition presented in Fig. 4: after the delay of 3s, there is no reaction by the system (there is no function transition), and the system state remains the same until the following delay transition. Moreover, we note that the possibility of adding a function transition to this state (the last state on the second row) would violate the invariant that requires that only function or delay transitions fire from the same state.

4.1.2. Complete definition of an e-DFRS

An e-DFRS is an element of the type defined by the following schema: *e_DFRS*. Hereafter, for simplicity, we only consider discrete delays, since dense delays are analogously defined. For all valid e-DFRSs, three invariants hold. First, all states are well typed (they range over the same set of variables defined as the system inputs, outputs, timers and global clock, besides mapping values consistent with the corresponding variable types). Second, all transitions are well typed. Third, the state reached by any transition is defined by the previous state updated by the assignments performed by the transition.

To formalise this last property, we rely on the auxiliary function *nextState*. Given a source state and a set of assignments, the function *nextState* yields a new state updating all system variables, but the global clock, according to these assignments. When dealing with delay transitions, besides considering the output of the function *nextState*, we also update the global clock adding to its value in the source state the delay performed. This last case is formalised by the last invariant ($trans.2 \in \operatorname{ran} del \Rightarrow trans.3 = ...$). After extracting the value embedded in the delay transition via the inverse definition of *discrete* ($discrete^\sim$), and similarly the current value of the system global clock (value mapped to gc) in the source state ($(n^\sim)\,((trans.1(gc)).2)$), we add these two values and the result is defined as the current value of the system global clock in the target state. The constructor n indicates that this result is a natural number. We note that the current value of the system global clock in the source state ($(trans.1(gc)).2$) becomes the previous value of gc in the target state.

e_DFRS

DFRS_VARIABLES
DFRS_STATES
DFRS_TRANSITION_RELATION

$$
\begin{array}{l}
\forall\, s : S \bullet (s, I \cup O \cup T \cup \{gcvar\}) \in well\_typed\_state \\
\forall\, trans : TR \bullet \{trans.1, trans.3\} \subseteq S \land \\
\quad (trans.2, I, O, T, gcvar) \in well\_typed\_transition \land \\
\quad (trans.2 \in \operatorname{ran} fun \Rightarrow trans.3 = nextState(trans.1, T, functionTransition(trans.2))) \land \\
\quad (trans.2 \in \operatorname{ran} del \Rightarrow trans.3 = \\
\qquad nextState(trans.1, T, (delayTransition(trans.2)).2) \oplus \{(gc, ((trans.1(gc)).2, \\
\qquad\quad n((n^\sim)\,((trans.1(gc)).2) + (discrete^\sim)\,((delayTransition(trans.2)).1)))))\})
\end{array}
$$

The state yielded by the function *nextState* is obtained by overriding the values of the previous state by the assignments of a given transition ($s \oplus ...$). Moreover, it updates accordingly the previous and current values of the variables: when a variable has its value updated, the current value of the previous state ($(n, (v1, v2))$) becomes the previous value of the next state, $(n, (v2, asgmts(n)))$.

$$
\begin{array}{l}
nextState : (STATE \times (NAME \nrightarrow TYPE) \times ASGMTS) \to STATE \\
\hline
\forall\, s : STATE;\ T : (NAME \nrightarrow TYPE);\ asgmts : ASGMTS \bullet nextState(s, T, asgmts) = s \oplus \\
\quad (\{n : NAME;\ v1, v2 : VALUE \mid (n, (v1, v2)) \in s \land n \in \operatorname{dom} asgmts \land \\
\qquad n \notin \operatorname{dom} T \bullet (n, (v2, asgmts(n)))\} \cup \\
\quad \{n : NAME;\ v1, v2 : VALUE \mid (n, (v1, v2)) \in s \land n \in \operatorname{dom} asgmts \land \\
\qquad n \in \operatorname{dom} T \bullet (n, (v1, (s(gc)).2))\})
\end{array}
$$

We note that there is a different definition when dealing with timers ($n \in \operatorname{dom} T$). In this case, the reset of a timer, which is represented by assigning 0, is encoded as an assignment of the current value of the global clock, $(s(gc)).2$. As the system has a single clock, it is easier to encode time reset by assigning the current value of the global clock, instead of assigning 0 and updating its value every time a delay transition is performed. Therefore, when evaluating timed guards such as $t < v$, where t is a timer and v a value, we actually evaluate the result of $(gc - t) < v$, where gc is the current value of the system global clock. Despite this representation, the previous value of the timer remains unchanged.

4.2. From s-DFRSs to e-DFRSs

The function *expandedDFRS* defines how an e-DFRS can be obtained from a symbolic one. The inputs (I), outputs (O), timers (T), the global clock ($gcvar$), and the initial state (s_0) are the same within both representations ($dfrs.I = symDFRS.I \land dfrs.O = symDFRS.O \land dfrs.T = symDFRS.T \land dfrs.gcvar = symDFRS.gcvar \land dfrs.s_0 = symDFRS.s_0$). The transition relation (TR) is obtained via the auxiliary function *buildTR* ($dfrs.TR = buildTR(\{dfrs.s_0\}, \emptyset, dfrs.I, dfrs.O, dfrs.T, symDFRS.F)$). The states of an e-DFRS (S) are defined as the states related by this transition relation ($trans.1$ and $trans.3$), besides the initial state.

$$
\begin{array}{|l}
expandedDFRS : s\_DFRS \rightarrow e\_DFRS \\
\hline
\forall\, symDFRS : s\_DFRS;\ dfrs : e\_DFRS \bullet expandedDFRS(symDFRS) = dfrs \Leftrightarrow \\
\quad dfrs.I = symDFRS.I \land dfrs.O = symDFRS.O \land dfrs.T = symDFRS.T \land \\
\quad dfrs.gcvar = symDFRS.gcvar \land dfrs.s_0 = symDFRS.s_0 \land \\
\quad dfrs.TR = buildTR(\{dfrs.s_0\}, \emptyset, dfrs.I, dfrs.O, dfrs.T, symDFRS.F) \land \\
\quad dfrs.S = \bigcup\{trans : dfrs.TR \bullet \{trans.1, trans.3\}\} \cup \{dfrs.s_0\}
\end{array}
$$

The function *buildTR* has six parameters: a set of states to visit (*toVisit*), a set of visited states (*visited*), the inputs (I), the outputs (O), the timers (T), and the functions of an s-DFRS (F). We note that in *expandedDFRS*, with respect to the function *buildTR*, *toVisit* has a single state to visit ($\{dfrs.s_0\}$), and *visited* is an empty set. Recursively, the function *buildTR* identifies new states to visit by the application of function and delay transitions that can emanate from the already visited states.

$$
\begin{array}{|l}
buildTR : ((\mathbb{P}\, STATE) \times (\mathbb{P}\, STATE) \times (NAME \nrightarrow TYPE) \times \\
\quad (NAME \nrightarrow TYPE) \times (NAME \nrightarrow TYPE) \times (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION)) \rightarrow TRANSREL \\
\hline
\forall\, toVisit, visited : \mathbb{P}\, STATE;\ I, O, T : NAME \nrightarrow TYPE;\ F : \mathbb{F}_1\, \mathbb{F}_1\, FUNCTION \bullet \\
\quad (toVisit = \emptyset \Rightarrow buildTR(toVisit, visited, I, O, T, F) = \emptyset) \land \\
\quad (toVisit \neq \emptyset \Rightarrow \exists\, s : toVisit;\ tr1 : TRANSREL \bullet \\
\qquad genTransitions(s, I, O, T, F) = tr1 \land \\
\qquad buildTR(toVisit, visited, I, O, T, F) = tr1 \cup \\
\qquad\quad buildTR((toVisit \cup \{trans : tr1 \bullet trans.3\}) \setminus (visited \cup \{s\}), visited \cup \{s\}, I, O, T, F))
\end{array}
$$

As an inductive function, the base case for *builtTR* happens when *toVisit* is empty. For this value of *toVisit*, we have that $buildTR(toVisit, visited, I, O, T, F)$ is an empty transition relation. In the inductive case, *toVisit* is not empty and, thus, there is at least one state s in the states to visit ($s : toVisit$). The result of *buildTR* is then defined as the union of the relation transition ($tr1$) obtained via *genTransitions*, which considers the emanating transitions from s, with the result of the recursive application of *buildTR*. This recursive application considers the not yet visited states, and also the new states reached by $tr1$ ($toVisit \cup \{trans : tr1 \bullet trans.3\}) \setminus (visited \cup \{s\})$. We note that we also need to add s to the set of visited states ($visisted \cup \{s\}$).

The function *genTransitions* identifies either function or delay transitions from a given state s. Delay transitions are performed from stable states, whereas function transitions occur in non-stable states. A state s is stable, $(s, ...) \in is\_stable$, when it does not represent a situation that triggers a system reaction: for all entries of the functions ($entry \in f$) of an s-DFRS ($f \in F$), their static ($entry.1$) and timed guards ($entry.2$) evaluate to false. The predicates *static_guards_true* and *timed_guards_true*, which are not presented here, are defined as the set of all static and timed guards that evaluate to true in a given state.

$$
\begin{array}{|l}
is\_stable : \mathbb{P}(STATE \times (NAME \nrightarrow TYPE) \times (NAME \nrightarrow TYPE) \times (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION)) \\
\hline
\forall\, s : STATE;\ IO : (NAME \nrightarrow TYPE);\ T : (NAME \nrightarrow TYPE);\ F : \mathbb{F}_1\, \mathbb{F}_1\, FUNCTION \bullet \\
\quad (s, IO, T, F) \in is\_stable \Leftrightarrow \\
\quad \forall\, f : F \bullet \forall\, entry : f \bullet (s, entry.1, IO, T) \notin static\_guards\_true \lor \\
\qquad (s, entry.2, T) \notin timed\_guards\_true \lor s = nextState(s, T, entry.3)
\end{array}
$$

A state is also considered to be stable if the reaction denoted by the assignments associated with these guards lead to a target state that is equal to the current one ($s = nextState(s, T, entry.3)$). In other words, the assignments do not have any effect. If there is no effect, this state is considered stable, since we do not have self transitions.

If a state s is stable, there are delay transitions emanating from s for all possible delays, $delay \in possibleDelays(...)$, which is formalised later, and all possible valid assignments, those whose values are consistent with the variable types ($asgmts.2 \in values(I(asgmts.1))$). These assignments also need to range over the complete set of inputs ($\text{dom}\ asgmts = \text{dom}\ I$). The reached state is defined by the function *nextState*, but also updating the system global clock based on the performed delay ($nextState(...) \oplus \{(gc, ... + ...)\}$). These three information (the given state – s; the delay transition considering a delay value and assignments – $del((delay, assigmts))$; and the reached state – $nextState(...) \oplus \{(gc, ...)\}$) are used to define the delay transition part of the result of *genTransitions*.

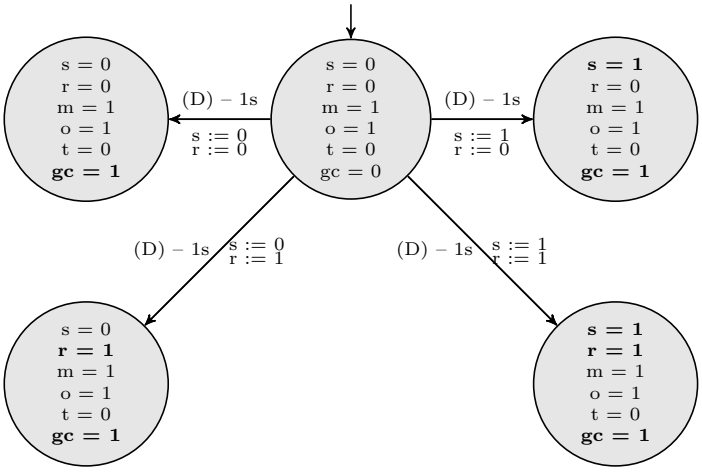

Fig. 7. The vending machine specification – example of delay transitions

$$
\begin{array}{|l}
genTransitions : (STATE \times (NAME \pfun TYPE) \times (NAME \pfun TYPE) \times \\
\quad (NAME \pfun TYPE) \times (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION)) \fun TRANSREL \\
\hline
\forall\, s : STATE;\ I, O, T : (NAME \pfun TYPE);\ F : \mathbb{F}_1\, \mathbb{F}_1\, FUNCTION \bullet \\
\quad ((s, I \cup O, T, F) \in is\_stable \Rightarrow genTransitions(s, I, O, T, F) = \\
\qquad \{delay : DELAY;\ asgmts : ASGMTS \mid delay \in genPossibleDelays(s, I \cup O, T, F) \land \\
\qquad \dom asgmts = \dom I \land (\forall\, asgmt : asgmts \bullet asgmt.2 \in values(I(asgmt.1))) \bullet \\
\qquad\quad (s, del((delay, asgmts)), nextState(s, T, asgmts) \oplus \\
\qquad\qquad \{(gc, ((s(gc)).2, n((n\,\tilde{}\,)\,((s(gc)).2) + (discrete\,\tilde{}\,)\,(delay))))\})\}) \land \\
\quad ((s, I \cup O, T, F) \notin is\_stable \Rightarrow genTransitions(s, I, O, T, F) = \\
\qquad \{entry : FUNCTION \mid (\exists f : F \bullet entry \in f) \land \\
\qquad (s, entry.1, I \cup O, T) \in static\_guards\_true \land (s, entry.2, T) \in timed\_guards\_true \bullet \\
\qquad\quad (s, fun(entry.3), nextState(s, T, entry.3))\})
\end{array}
$$

Fig. 7 shows a concrete example of delay transitions emanating from the initial state of the VM. We note that we have a transition for each valid combination of input values: the coin sensor and the request button remain false (first state on first row), only the coin sensor becomes true (third state on first row), only the request button becomes true (first state on second row), and both signals become true (second state on second row). Although only the transitions with delay equal to 1 second are shown, there are transitions with greater delays (2s, 3s, ...) emanating from the initial state. In this case, all delays are possible and, thus, there is no upper bound. This leads to an infinite number of delay transitions emanating from the initial state.

To understand how we define the maximum valid delay, we first need to explain the concept of enabling delays, which is captured by the following partial function *enablingDelays*. The domain of this function is the set of states that are stable, $(s, ...) \in is\_stable$. Given a stable state s and a single entry ($entry$) of a function of an s-DFRS, the function *enablingDelays* yields a set of delays such that, after advancing the time by this delay ($(gc, ... + ...)$), without changing any input value, the reached state ($next$) is not stable, $(next, ...) \notin is\_stable$. In other words, if we just let the time evolve by some amount, we are going to see some reaction of the system.

$$
\begin{array}{|l}
enablingDelays : (STATE \times (NAME \pfun TYPE) \times (NAME \pfun TYPE) \times FUNCTION) \pfun \mathbb{P}\, DELAY \\
\hline
\dom enablingDelays = \{s : STATE;\ IO : (NAME \pfun TYPE);\ T : (NAME \pfun TYPE); \\
\quad entry : FUNCTION \mid (s, IO, T, \{\{entry\}\}) \in is\_stable \bullet (s, IO, T, entry)\} \\
\forall\, s : STATE;\ IO : (NAME \pfun TYPE);\ T : (NAME \pfun TYPE);\ entry : FUNCTION \bullet \\
\quad enablingDelays(s, IO, T, entry) = \{delay : DELAY;\ next : STATE \mid next = s \oplus \\
\qquad \{(gc, ((s(gc)).2, n((n\,\tilde{}\,)\,((s(gc)).2) + (discrete\,\tilde{}\,)\,(delay))))\} \land \\
\qquad\quad (next, IO, T, \{\{entry\}\}) \notin is\_stable \bullet delay\}
\end{array}
$$

The situation described in the end of the last paragraph (reaching a non-stable state by just letting the time advance) happens in the VM when the system is producing coffee. After pressing the coffee request button, if a weak coffee is going to be produced, we observe this system reaction within 10 to 30 seconds. Therefore, if we are in the first state on the third row (Fig. 4), for delays greater than or equal to 10 and lower than or equal to 30, we observe a system reaction leading the system to the reset state, besides changing accordingly the system output. In such a state, for instance, it would not make sense to have a delay transition, whose delay is 31, since we would be modelling an input received after elapsing 31 seconds, but before this input being received we should have observed a system reaction. This captures the principle of *delayable* transitions: the time might advance an arbitrary amount as long as it does not disable an enabled transition.

Considering the situation explained in the last paragraph for the VM example, the function *enablingDelays* yields the set 10..30. It is worth noting that the result of *enablingDelays* can be an infinite set, for example, if a weak coffee should be produced at least 10 seconds after its request. In such a case, as we do not have an upper bound, the result of *enablingDelays* is $10..\infty$.

Now, given a stable state s, and considering all functions of an s-DFRS (F), the function *maxDelays* yields the upper bound (*upperBound*) of the set enabling delays ($delays = enablingDelays(...)$) with respect to each entry ($entry \in f$) of the s-DFRS functions ($f : F$). If *delays* is not empty, $discrete(upperBound) \in delays$, and there is an upper bound, $\forall\, n : delays \bullet (discrete^{\sim})(n) \leq upperBound$, *delays* is not infinite; this upper bound is considered in the return of *maxDelays*. In other words, its result considers the maximum delay allowed, based on the *delayable* principle, for each entry of the functions of an s-DFRS.

$$
\begin{array}{|l}
maxDelays : (STATE \times (NAME \nrightarrow TYPE) \times (NAME \nrightarrow TYPE) \times (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION)) \nrightarrow \mathbb{F}\, \mathbb{N}_1 \\
\hline
\mathrm{dom}\, maxDelays = \{s : STATE;\ IO : (NAME \nrightarrow TYPE);\ T : (NAME \nrightarrow TYPE); \\
\quad F : (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION) \mid (s, IO, T, F) \in is\_stable \bullet (s, IO, T, F)\} \\
\forall\, s : STATE;\ IO : (NAME \nrightarrow TYPE);\ T : (NAME \nrightarrow TYPE);\ F : \mathbb{F}_1\, \mathbb{F}_1\, FUNCTION \bullet \\
\quad maxDelays(s, IO, T, F) = \{f : F;\ entry : FUNCTION;\ delays : \mathbb{P}\, DELAY;\ upperBound : \mathbb{N}_1 \mid \\
\qquad entry \in f \land delays = enablingDelays(s, IO, T, entry) \land discrete(upperBound) \in delays \land \\
\qquad (\forall\, n : delays \bullet (discrete^{\sim})(n) \leq upperBound) \bullet upperBound\}
\end{array}
$$

To define the set of possible delays that we need to consider when generating delay transitions, we rely on the auxiliary function *genPossibleDelays*. Basically, for a given state s, if the result of the application of *maxDelays* is empty ($maxDelays(...) = \emptyset$), it means that there is no upper bound we need to consider and, thus, all delays are possible, $genPossibleDelays(...) = \{delay : DELAY\}$. Otherwise, we can perform all delays that are lower than or equal to the lowest upper bound defined by *maxDelays*, $(discrete^{\sim})(delay) \leq miniumDelay(...)$. The function *mininumDelay* yields this lowest upper bound, whose definition is not shown here as it is straightforward.

$$
\begin{array}{|l}
genPossibleDelays : (STATE \times (NAME \nrightarrow TYPE) \times (NAME \nrightarrow TYPE) \times (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION)) \\
\quad \nrightarrow \mathbb{P}\, DELAY \\
\hline
\mathrm{dom}\, genPossibleDelays = \{s : STATE;\ IO : (NAME \nrightarrow TYPE);\ T : (NAME \nrightarrow TYPE); \\
\quad F : (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION) \mid (s, IO, T, F) \in is\_stable \bullet (s, IO, T, F)\} \\
\forall\, s : STATE;\ IO : (NAME \nrightarrow TYPE);\ T : (NAME \nrightarrow TYPE);\ F : \mathbb{F}_1\, \mathbb{F}_1\, FUNCTION \bullet \\
\quad (maxDelays(s, IO, T, F) = \emptyset \Rightarrow genPossibleDelays(s, IO, T, F) = \{delay : DELAY\}) \land \\
\quad (maxDelays(s, IO, T, F) \neq \emptyset \Rightarrow genPossibleDelays(s, IO, T, F) = \{delay : DELAY \mid \\
\qquad (discrete^{\sim})(delay) \leq minimumDelay(maxDelays(s, IO, T, F))\})
\end{array}
$$

To finish our explanation of how to obtain an e-DFRS from a symbolic one, we need to detail how function transitions are created. If we refer to the definition of *genTransitions*, presented at the beginning of this section and partially reproduced below, we can see that function transitions, $(s, fun(...), ...)$, emanate from states that are not stable. $(s, ...) \notin is\_stable$.

$$
\begin{array}{|l}
genTransitions : (STATE \times (NAME \pfun TYPE) \times (NAME \pfun TYPE) \times \\
\quad (NAME \pfun TYPE) \times (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION)) \fun TRANSREL \\
\hline
\forall\, s : STATE;\ I, O, T : (NAME \pfun TYPE);\ F : \mathbb{F}_1\, \mathbb{F}_1\, FUNCTION \bullet \\
\quad ... \land \\
\quad ((s, I \cup O, T, F) \notin is\_stable \Rightarrow genTransitions(s, I, O, T, F) = \\
\qquad \{entry : FUNCTION \mid (\exists f : F \bullet entry \in f) \land \\
\qquad (s, entry.1, I \cup O, T) \in static\_guards\_true \land (s, entry.2, T) \in timed\_guards\_true \bullet \\
\qquad\quad (s, fun(entry.3), nextState(s, T, entry.3))\})
\end{array}
$$

For every entry ($entry \in f$) of the functions of an s-DFRS ($f : F$), whose static ($entry.1$) and timed guards ($entry.2$) evaluate to true, we add a function transition with the corresponding assignments, $fun(entry.3)$, leading to a target state that is the previous one modified by these assignments ($nextState(...)$). In the VM example, we have a deterministic system. However, for non-deterministic systems, we would have more than one function transition emanating from the same source state.

In summary, from the initial state of an s-DFRS, we recursively identify which states are reached by function and delay transitions. The set of all reachable states, which is defined as the states of an e-DFRS, besides their transitions, is considered the transition relation of an e-DFRS. The other elements of an e-DFRS are directly obtained from the corresponding symbolic ones.

4.3. Soundness of generation of an e-DFRS

Besides presenting a function that yields an e-DFRS from a symbolic one, it is important to show that this function is sound: for all s-DFRSs, all invariants of *e_DFRS* hold in the obtained e-DFRS (Theorem 4.1).

Theorem 4.1. Soundness of *expandedDFRS*

$$\forall\, symDFRS : s\_DFRS \bullet expandedDFRS(symDFRS) \in e\_DFRS$$

The complete proof of Theorem 4.1 is available in [CCS15]. Here, we present a proof sketch. The three invariants of *DFRS_VARIABLES* (reproduced below) trivially hold as the elements I, O, T, and $gcvar$ are the same of the corresponding s-DFRS, and these properties are also invariants of valid s-DFRSs.

$$
\begin{array}{l}
gcvar = (gc, nat) \lor gcvar = (gc, ufloat) \\
\mathsf{disjoint}\ \langle \mathrm{dom}\, I, \mathrm{dom}\, O, \mathrm{dom}\, T \rangle \\
\mathrm{ran}\, T \subseteq \{gcvar.2\}
\end{array}
$$

The invariant of *DFRS_STATE* ($s_0 \in S$) also holds, since *expandedDFRS* defines S as the union of the states of TR with s_0. The invariants of *DFRS_TRANSITION_RELATION* (reproduced below) are also preserved by the function *expandedDFRS*.

$$
\begin{array}{l}
\forall\, trans1, trans2 : TR \mid trans1.1 = trans2.1 \bullet \\
\quad \{trans1.2, trans2.2\} \subseteq \mathrm{ran}\, fun \lor \{trans1.2, trans2.2\} \subseteq \mathrm{ran}\, del \\
\forall\, trans : TR \bullet \neg\, (trans.1 = trans.3)
\end{array}
$$

The first one holds because function transitions are only created from non-stable states, whereas delay transitions are created from stable states. As one state cannot be non-stable and stable simultaneously, all transitions emanating from a state are function or delay ones. We also do not have self transitions as the delay transitions advance the time by a value greater than 0 and, thus, it leads to a different state (a different value for global clock). The function transition is only performed if it has a collateral effect (changes the value of at least one variable) and, thus, it also leads to a different state. Therefore, the second invariant of *DFRS_TRANSITION_RELATION* also holds.

The function *expandedDFRS* also preserves the invariants of *e_DFRS*. Concerning the first one (reproduced below), a state is said to be well typed if, and only if, it ranges over the complete set of system variables, and the values assigned to them are consistent with the variable types.

$$\forall\, s : S \bullet (s, I \cup O \cup T \cup \{gcvar\}) \in well\_typed\_state$$

Considering the definition of *buildTR*, the states of an e-DFRS are reachable from its initial state, which is well typed based on the definition of s-DFRSs, performing delay and function transitions. Each transition

only changes the values mapped to the variables, but not the set of variables. Therefore, all states consider the same set of variables, which are all system variables. Concerning the consistency of values, the assignments performed by the transitions also need to be consistent with the variable types and, thus, this consistency is respected in all states.

Now, we explain why the invariants related to the transition relation (reproduced below) also hold.

$$\begin{aligned}
&\forall\, trans : TR \bullet \{trans.1, trans.3\} \subseteq S \land \\
&\quad (trans.2, I, O, T, gcvar) \in well_typed_transition \land \\
&\quad (trans.2 \in \operatorname{ran} fun \Rightarrow trans.3 = nextState(trans.1, T, functionTransition(trans.2))) \land \\
&\quad (trans.2 \in \operatorname{ran} del \Rightarrow trans.3 = \\
&\qquad nextState(trans.1, T, (delayTransition(trans.2)).2) \oplus \{(gc, ((trans.1(gc)).2, \\
&\qquad\quad n((n\,^{\sim})\,((trans.1(gc)).2) + (discrete\,^{\sim})\,((delayTransition(trans.2)).1))))\})
\end{aligned}$$

The first invariant is clearly preserved, since *expandedDFRS* defines S as all states related by TR, besides its initial state. Regarding the second invariant of *e_DFRS*, which states that all transitions are well typed, it is a consequence of how delay and function transitions are defined by the function *genTransitions*. As one can notice from the definition of this function, the delay transition considers all input variables, and the delay value is consistent with the system global clock. Therefore, the delay transitions are well typed and clock compatible. The function transitions are defined considering the assignments mapped to static and timed guards of the functions of an s-DFRS. Considering the definition of s-DFRSs, these assignments are well typed and, thus, consider a subset of the system outputs and timers. Therefore, the function transitions of an e-DFRS are also well typed.

Finally, the last invariants of *e_DFRS* say that the target state of a delay and a function transition is defined by the source state updated with the corresponding assignments, besides advancing the system global clock by the delay value in delay transitions. This is exactly how the next (target) states are defined by the auxiliary function *genTransitions* and, thus, this last invariant holds too.

4.4. Verifying properties of requirements via e-DFRSs

By exploring the state space of an e-DFRS, we can verify interesting properties of the system requirements. Besides checking whether the requirements are ambiguous (hereafter, called inconsistent) or incomplete, we can also verify the presence of unreachable requirements and time lock.

4.4.1. Consistent requirements

The system requirements are said to be consistent if, and only if, they do not describe different system reactions for the same context (state). Definition 4.1 formalises this concept.

Definition 4.1. Consistent requirements: let *reqs* be an arbitrary set of requirements, and *symDFRS* the corresponding s-DFRS obtained via Algorithm 6; the following predicate defines when these requirements are said to be consistent:

$$\begin{aligned}
&consistent(reqs) \Leftrightarrow \exists\, dfrs : e_DFRS \mid dfrs = expandedDFRS(symDFRS) \bullet \\
&\quad \forall\, s : dfrs.S \bullet (s, dfrs.I \cup dfrs.O, dfrs.T, symDFRS.functions) \notin is_stable \Rightarrow \\
&\qquad \forall\, f1, f2 : symDFRS.F \bullet \forall\, e1 : f1;\ e2 : f2 \bullet \\
&\qquad\quad \{(s, e1.1, dfrs.I \cup dfrs.O, dfrs.T), (s, e2.1, dfrs.I \cup dfrs.O, dfrs.T)\} \subseteq static_guards_true \\
&\qquad\quad \land \{(s, e1.2, dfrs.T), (s, e2.2, dfrs.T)\} \subseteq timed_guards_true \Rightarrow e1 = e2
\end{aligned}$$

According to the algorithms presented in Section 3, each requirement is mapped to an entry of a function. Therefore, if the requirements are consistent, for all states (s) of the e-DFRS (*dfrs*), if the guards ($\_.1, \_.2$) of two entries ($e1, e2$) of two arbitrary functions ($f1, f2$) evaluate to true (... $\subseteq$ *static_guards_true* and ... $\subseteq$ *timed_guards_true*) in the same non-stable state, $(s, ...) \notin$ *is_stable*, these entries are the same ($e1 = e2$). In such a case, we say that the requirements are consistent. Otherwise, we would have two different system reactions for the same state. □

To give a concrete example of inconsistent requirements, we consider the following ones:

- *When input1 is true, the system shall assign 1 to output1.*

- *When input1 is true, the system shall assign 2 to output1.*

These two requirements are not consistent, since they describe different system reactions (assigning 1 or assigning 2, respectively) for the same context (when input1 is true).

4.4.2. Complete requirements

The requirements are said to be complete if for every possible system input (after each delay transition), there is some system reaction (a function transition). This notion of completeness is formalised by Definition 4.2

Definition 4.2. Complete requirements: let *reqs* be an arbitrary set of requirements, and *symDFRS* the corresponding s-DFRS obtained via Algorithm 6; the following predicate defines when these requirements are said to be complete:

$$
\begin{array}{l}
complete(reqs) \Leftrightarrow \exists\, dfrs : e\_DFRS \mid dfrs = expandedDFRS(symDFRS) \bullet \\
\quad \forall\, s1, s2 : dfrs.S \mid (\exists\, trans : dfrs.TR \mid trans.1 = s1 \land trans.3 = s2 \land trans.2 \in \operatorname{ran} del) \bullet \\
\quad\quad \exists\, s3 : dfrs.S;\ trans2 : dfrs.TR \bullet trans2.1 = s2 \land trans2.3 = s3 \land trans.2 \in \operatorname{ran} fun
\end{array}
$$

□

To exemplify complete requirements, we consider a simple system that has a single input (*input*1) and a single output (*output*1). The following requirements are said to be complete:

- *When input1 is true, the system shall assign 1 to output1.*
- *When input1 is false, the system shall assign 2 to output1.*

We note that for every possible value of *input*1 (*true* or *false*), the requirements define the expected system reaction (assigning 1 or assigning 2, respectively). Therefore, after every delay transition, there is a function transition.

4.4.3. Reachable requirements

A requirement is reachable if there is a state of the e-DFRS where the guards of the function entry obtained from this requirement evaluate to true. If there is no such a state, we say that this requirement is not reachable, since it describes a system reaction that will never occur. Definition 4.3 defines formally the notion of reachable requirements.

Definition 4.3. Reachable requirements: let *reqs* be an arbitrary set of requirements, and *symDFRS* the corresponding s-DFRS obtained via Algorithm 6; the following predicate defines when these requirements are said to be reachable:

$$
\begin{array}{l}
reachable(reqs) \Leftrightarrow \exists\, dfrs : e\_DFRS \mid dfrs = expandedDFRS(symDFRS) \bullet \\
\quad \forall\, f : symDFRS.F \bullet \forall\, entry : f \bullet \exists\, s : dfrs \bullet \\
\quad\quad (s, dfrs.I \cup dfrs.O, dfrs.T, symDFRS.functions) \notin is\_stable \land \\
\quad\quad (s, entry.1, dfrs.I \cup dfrs.O, dfrs.T) \in static\_guards\_true \land \\
\quad\quad (s, entry.2, dfrs.T) \in timed\_guards\_true
\end{array}
$$

□

To give a concrete example of an unreachable requirement, we consider the following one:

- *When input1 is true, and input1 is false, the system shall assign 1 to output1.*

This requirement is not reachable (its reaction is never observed), since its condition does not evaluate to true in any possible state due to the fact that the same boolean variable (*input*1) cannot be *true* and *false* simultaneously.

4.4.4. Absence of time lock

Finally, the last property concerns the absence of time lock (see Definition 4.4). A time lock is characterised by a state from which it is not possible to perform delay transitions, immediately and not even from states reachable by this state. If such a state exists, we have a time lock, since delay transitions cannot occur and,

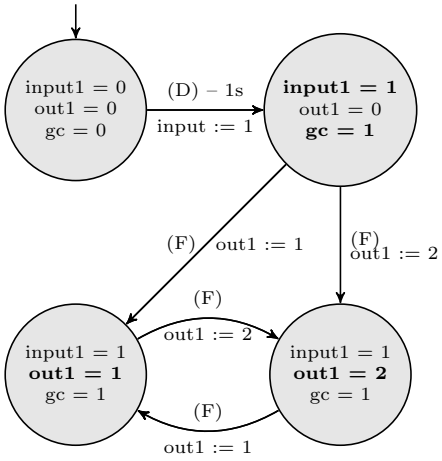

Fig. 8. Example of time lock

thus, time cannot elapse. Another way of expressing this property is to say that time lock happens if there is a state from which it is not possible to reach stable states (states that have delay transitions).

Definition 4.4. Absence of time lock: let *reqs* be an arbitrary set of requirements, and *symDFRS* the corresponding s-DFRS obtained via Algorithm 6; the following predicate defines when these requirements describe a system without time lock:

$$noTimeLock(reqs) \Leftrightarrow \exists\, dfrs : e\_DFRS \mid dfrs = expandedDFRS(symDFRS) \bullet \\ \forall\, trans : dfrs.TR \mid trans.2 \in \operatorname{ran} fun \bullet \exists\, trans2 : dfrs.TR \bullet \mid trans2.2 \in \operatorname{ran} del \bullet \\ (trans.3, trans.1, dfrs.TR) \in is\_reachable$$

□

To give a concrete example of time lock, we consider the sample example that was given to illustrate Definition 4.1. Fig. 8 shows part of the e-DFRS obtained from these two requirements. We note that when *input*1 becomes true (equal to 0), the system reaches a state from which is not possible to reach a stable state, since it can perform indefinitely function transitions. In such a situation, we say that there is a time lock. In this example, the requirements are also inconsistent, since we can perform more than one function transition from the second state on the first row.

The properties defined in this section can be verified by exploring the e-DFRS state space. However, as an e-DFRS possibly comprises an infinite set of states, it is necessary to specify a bound for this check. Then, one can dynamically create an e-DFRS until this bound is reached, and, while it is created, check whether the desired properties are met (bounded model checking). Eligible criteria for this bound are the number of delay (function) transitions performed, and an upper bound for the system global clock, among others. If the specification is inconsistent or there is an unreachable requirement, we can easily identify the requirements involved as we keep traceability between the s-DFRS functions and the requirements (see Section 3.5).

4.5. Tool support

The NAT2TEST tool implements the function *genTransitions*, allowing us to create and explore the states of an e-DFRS dynamically (see Fig. 9). When the *animator* screen is opened, it automatically creates the e-DFRS initial state (state 0): the initial state of the corresponding s-DFRS, which is obtained from the system requirements.

On the top right, the tool shows the possible delay or function transitions that can be performed from the selected state. A double-click on a transition creates an edge to the target state to represent it. If the transition is a delay one, a pop-up opens, and the user can inform the amount of (discrete or continuous) time that advances with the delay transition, and new values for the input signals. On the right, the tool

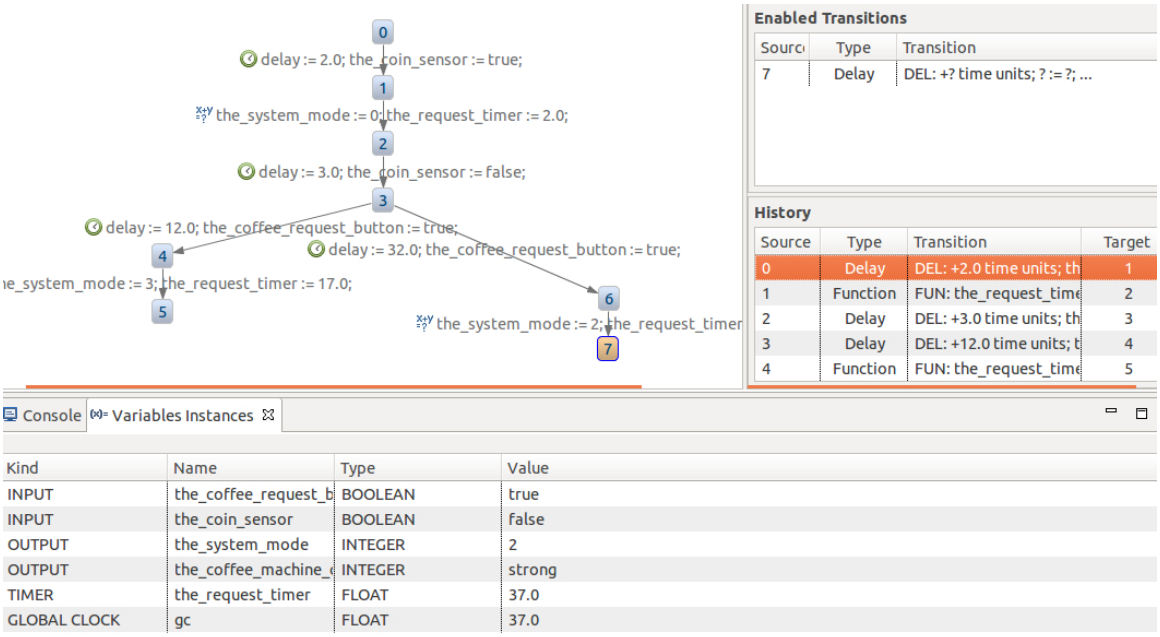

Fig. 9. The NAT2TEST tool – dynamic creation of e-DFRSs

shows the history of performed transitions. On the bottom, the tool shows the value of the system variables considering the selected state.

Fig. 9 illustrates part of the e-DFRS for the example shown in Fig. 4, but it also describes the transition that leads to the production of strong coffee. We note that from the state 3, if the coffee request button is pressed 12 seconds after inserting the coin, the system goes to the weak mode ($the\_system\_mode := 3$), which is represented by the state 5. Differently, if the request is made 32 seconds after inserting the coin, the system goes to the strong mode ($the\_system\_mode := 2$), which is represented by the state 7.

With the aid of this tool support, as explained in Section 5.2, we assess whether test cases, either independently written by domain specialists from industry or generated by a commercial tool from the same set of requirements, are compatible with the corresponding DFRS models. We detail this analysis in Section 5.2.

5. Theoretical and practical validations

After presenting the DFRS models (s-DFRS and e-DFRS), we now discuss a theoretical (Section 5.1) and a practical validation (Section 5.2).

5.1. Theoretical validation: mapping e-DFRSs to TIOTSs

While an e-DFRS can be viewed as a semantics for the s-DFRS, from which it is obtained, in order to connect such a semantic representation to established ones in the literature, we show that an e-DFRS can be encoded as a Timed Input-Output Transition System (TIOTS). This is an alternative timed model based on the widely used IOLTS and ioco [Tre99]. First, we define TIOTSs in Z (Section 5.1.1), and then we show how it can be obtained from e-DFRSs (Section 5.1.2) via a sound process (Section 5.1.3).

5.1.1. Formal model of TIOTS

A TIOTS is a 6-tuple $(Q,\ q_0,\ I,\ O,\ D,\ T)$, where Q is a (possibly infinite) set of states, q_0 is the initial state, I represents input and O output actions, D is a set of delays, and T is a (possibly infinite) transition relation on states.

In a TIOTS, the states are related by labelled transitions. A label can be an input or an output action, a delay, or an internal action. The given set *TIOTS_ACTION* represents all valid actions, and *TIOTS_ACTIONS* a set of actions. A TIOTS delay (an element of *TIOTS_DELAY*) represents a discrete or a dense time elapsing, but differently from an e-DFRS delay, a delay in a TIOTS can also be 0. *TIOTS_DELAYS* is a set of delays.

$$[TIOTS\_ACTION]$$
$$TIOTS\_ACTIONS == \mathbb{P}\ TIOTS\_ACTION$$
$$TIOTS\_DELAY ::= tiots\_discrete\langle\langle \mathbb{N} \rangle\rangle \mid tiots\_dense\langle\langle R^+ \rangle\rangle$$
$$TIOTS\_DELAYS == \mathbb{P}\ TIOTS\_DELAY$$

The schema *TIOTS_LABELS* formalises the concept of TIOTS labels.

TIOTS_LABELS
$$I, O : TIOTS\_ACTIONS$$
$$D : TIOTS\_DELAYS$$

$$\mathsf{disjoint}\ \langle I, O \rangle$$
$$D \in tiots\_time\_compatible$$

The sets of input and output actions are disjoint, and the delays need to be time compatible, which means that all delays are of the same type (discrete or dense). The time compatible delays are characterised by the elements of a set *tiots_time_compatible*, whose simple definition is omitted here.

A state of a TIOTS is an element of the given set *TIOTS_STATE*, and *TIOTS_STATES_SET* is a non-empty set of states. The initial state of a TIOTS (q_0) is necessarily an element of the set of states of a TIOTS (Q). The schema *TIOTS_STATES* formalises this invariant.

$$[TIOTS\_STATE]$$
$$TIOTS\_STATES\_SET == \mathbb{P}_1\ TIOTS\_STATE$$
$$TIOTS\_STATES == [\ Q : TIOTS\_STATES\_SET;\ q_0 : TIOTS\_STATE \mid q_0 \in Q\ ]$$

The transition relation (T), which is an element of the set of all possible TIOTS transition relations (*TIOTS_TRANSREL*), relates two states by means of a label, an element of *TIOTS_TRANS_LABEL*. A TIOTS has four types of transitions: input, output, delay and internal transitions, which are labelled with input actions, output actions, delay events, and internal actions, represented by the invisible event τ, respectively.

$$TIOTS\_TRANS\_LABEL ::= in\langle\langle TIOTS\_ACTION \rangle\rangle \mid out\langle\langle TIOTS\_ACTION \rangle\rangle \mid$$
$$\quad tiots\_del\langle\langle TIOTS\_DELAY \rangle\rangle \mid tau$$
$$TIOTS\_TRANS == (TIOTS\_STATE \times TIOTS\_TRANS\_LABEL \times TIOTS\_STATE)$$
$$TIOTS\_TRANSREL == \mathbb{P}\ TIOTS\_TRANS$$

The schema *TIOTS_TRANSITION_RELATION* defines the component T.

TIOTS_TRANSITION_RELATION
$$T : TIOTS\_TRANSREL$$

Finally, a TIOTS is defined by the schema *TIOTS*, which requires that each transition relates states of Q and is well-typed.

TIOTS
$$TIOTS\_LABELS$$
$$TIOTS\_STATES$$
$$TIOTS\_TRANSITION\_RELATION$$

$$\forall\ entry : T \bullet \{entry.1, entry.3\} \subseteq Q \land (entry.2, I, O, D) \in well\_typed\_tiots\_transition$$

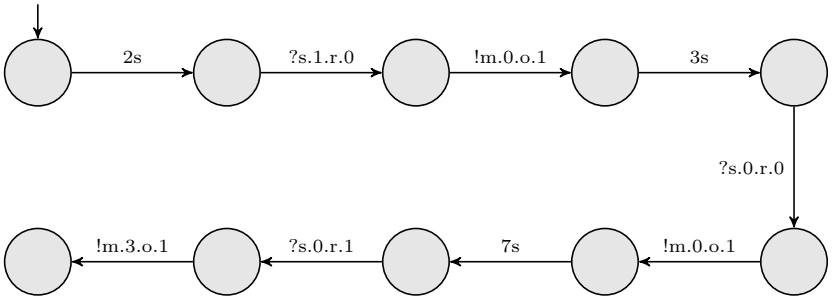

Fig. 10. The vending machine specification – TIOTS representation

A transition is said to be well typed (as characterised by the elements of *well_typed_tiots_transition*) if, and only if, its label is equal to τ, *in*, *out*, or *del* ($label = tau$, $label \in \operatorname{ran} in$, $label \in \operatorname{ran} out$, $label \in \operatorname{ran} tiots\_del$, respectively).

$$
\begin{array}{|l}
well\_typed\_tiots\_transition : \mathbb{P}(TIOTS\_TRANS\_LABEL \times \\
\quad TIOTS\_ACTIONS \times TIOTS\_ACTIONS \times TIOTS\_DELAYS) \\
\hline
\forall\, label : TIOTS\_TRANS\_LABEL;\ I, O : TIOTS\_ACTIONS;\ D : TIOTS\_DELAYS \bullet \\
\quad (label, I, O, D) \in well\_typed\_tiots\_transition \Leftrightarrow \\
\quad (label = tau) \lor (label \in \operatorname{ran} in \land (in^{\sim})\, label \in I) \lor \\
\quad (label \in \operatorname{ran} out \land (out^{\sim})\, label \in O) \lor (label \in \operatorname{ran} tiots\_del \land (tiots\_del^{\sim})\, label \in D)
\end{array}
$$

If the label represents an input action ($label \in \operatorname{ran} in$), it comprises elements of I, $(in^{\sim})\, label \in I$. Similarly, the same idea applies to output actions and delays, where O and D are considered, respectively.

5.1.2. From e-DFRSs to TIOTSs

Before formalising the generation of a TIOTS from an e-DFRS, we explain the intuition behind the generation process. While a function transition is mapped to an output action, a delay transition is mapped to a delay followed by an input action. When a function transition leads to a non-stable event, this transition is mapped to an internal hidden event, since only stable communication of outputs can be observed. If a delay transition leads to a state from which there are other delay transitions (after the first delay no system reaction is observed), we also consider an output action between these two transitions to show explicitly that the system outputs have not changed.

Fig. 10 shows the TIOTS obtained from the first five transitions presented in Fig. 4. To differentiate input from output actions, we add "?" as a prefix to the former, and "!" to the latter. We note that the actions performed are strings that represent the value received for all system inputs or generated for all system outputs, even if the function transition does not range necessarily over the complete set of system outputs. We note that an output action is performed between the delay transitions, whose delays are 3s and 7s, to show that the system outputs remain unchanged. In this short example, we do not have τ events, as all states reached by function transitions are stable. However, considering the example shown in Fig. 8, the corresponding TIOTS does not have any output action after the delay transition, but a loop of τ events due to the time lock.

The function *fromDFRStoTIOTS* defines how a TIOTS is obtained from an e-DFRS. The main step is how to obtain the TIOTS transition relation ($tiots.T$), which is defined by *mapTransitionRelation*. The TIOTS inputs ($tiots.I$), outputs ($tiots.O$) and delays ($tiots.D$) are defined as the result of auxiliary projection functions (*getInputActions*, *getOutputActions*, and *getDelays*, respectively). Basically, these functions yield the labels of TIOTS transitions.

$$
\begin{array}{|l}
fromDFRStoTIOTS : e\_DFRS \rightarrow TIOTS \\
\hline
\forall\, dfrs : e\_DFRS;\ tiots : TIOTS \bullet \\
\quad fromDFRStoTIOTS(dfrs) = tiots \Leftrightarrow \\
\quad tiots.Q = getStates(tiots.T) \cup \{tiots.q_0\} \land tiots.q_0 = mapState(dfrs.s_0) \land \\
\quad tiots.I = getInputActions(tiots.T) \land tiots.O = getOutputActions(tiots.T) \land \\
\quad tiots.D = getDelays(tiots.T) \land tiots.T = mapTransitionRelation(dfrs.TR, dfrs.I, dfrs.O)
\end{array}
$$

The function *mapState* is a total injection from DFRS states to TIOTS ones. It is used to define the initial state ($tiots.q_0$) of the TIOTS; it is the result of this function when applied to the initial state of the e-DFRS. The states ($tiots.Q$) of a TIOTS are the ones related by its transition relation ($tiots.T$), which are characterised by *getStates*, besides its initial state.

To obtain the TIOTS transition relation, the function *mapTransitionRelation* considers two partitions of e-DFRS transitions: the first one comprises only function transitions, $getTransitions(tr, \operatorname{ran} fun)$, whereas the second one comprises delay transitions, $getTransitions(tr, \operatorname{ran} del)$. The function *getTransitions* filters function or delay transitions from a given transition relation (tr). The functions *mapFunTransitions* and *mapDelTransitions* consider these partitions and yield transition relations ($tr1$, and $tr2$), whose union is defined as the result of *mapTransitionRelations* ($mapTransitionRelation = tr1 \cup tr2$).

$$
\begin{array}{|l}
mapTransitionRelation : TRANSREL \times (NAME \nrightarrow TYPE) \times (NAME \nrightarrow TYPE) \rightarrow \\
\quad TIOTS\_TRANSREL \\
\hline
\forall\, tr : TRANSREL;\ I, O : (NAME \nrightarrow TYPE) \bullet \exists\, tr1, tr2 : TIOTS\_TRANSREL \bullet \\
\quad tr1 = mapFunTransitions(getTransitions(tr, \operatorname{ran} fun), tr, O) \land \\
\quad tr2 = mapDelTransitions(getTransitions(tr, \operatorname{ran} del), tr, \operatorname{ran} mapState, I, O) \land \\
\quad mapTransitionRelation(tr, I, O) = tr1 \cup tr2
\end{array}
$$

One important concern is related to the fresh states that are needed during this process. Therefore, we note that the third argument of *mapDelTransitions* is ran *mapState* (all TIOTS states that can be obtained from DFRS states), which is later used to identify fresh ones. For instance, one can see in Fig. 10 that the first and third states are obtained from the first and second states in Fig. 4, whereas the second state does not have any correspondence with a DFRS state.

The recursive function *mapFunTransitions* applies *mapFunTransition* for each function transition that leads to a stable state. The latter function yields an output action, *out*(...), relating the TIOTS states obtained from the source, $mapState(s1)$ and target, $mapState(s2)$, states of the e-DFRS function transition.

$$
\begin{array}{|l}
mapFunTransition : (TRANS \times (NAME \nrightarrow TYPE)) \nrightarrow TIOTS\_TRANSREL \\
\hline
dom(mapFunTransition) = (STATE \times \operatorname{ran} fun \times STATE) \times (NAME \nrightarrow TYPE) \\
\forall\, s1, s2 : STATE;\ label : TRANS\_LABEL;\ O : (NAME \nrightarrow TYPE) \mid label \in \operatorname{ran} fun \bullet \\
\quad mapFunTransition((s1, label, s2), O) = \{(mapState(s1), \\
\qquad out(genAction(currentValues(\operatorname{dom} O \lhd s2))), mapState(s2))\}
\end{array}
$$

The output action, *out*(...), is defined in terms of the current values of the output variables in the target state ($currentValues(\operatorname{dom} O \lhd s2)$). An example of output action is $!m.0.o.1$ (see Fig. 10). When the function transition leads to a non-stable state, the application *mapFunTransitions* yields a transition relation, whose single element relates the source, $mapState(trans.1)$, and target, $mapState(trans.3)$, states with a τ event, $\{(mapState(trans.1), tau, mapState(trans.3))\}$.

The process of mapping delay transitions is more complicated due to three main reasons. First, as previously explained, we need to identify fresh states (that do not have correspondence to DFRS states); second, as also commented before, we need to define an output action between consecutive delay transitions; finally, the delay transitions that have the same amount of time elapsing are grouped into *non-time deterministic* partitions, since they have a particular treatment. To exemplify this last situation, we consider the states presented in Fig. 7. The TIOTS transition relation obtained from this example is shown in Fig. 11. One can see that first we have a delay of 1s leading to a state from which there are multiple possible input actions.

5.1.3. Soundness of mapping to TIOTS

Similarly to Theorem 4.1, the function *fromDFRStoTIOTS* is also proven to be sound: for all e-DFRSs, all invariants of TIOTS hold in the obtained TIOTS (Theorem 5.1).

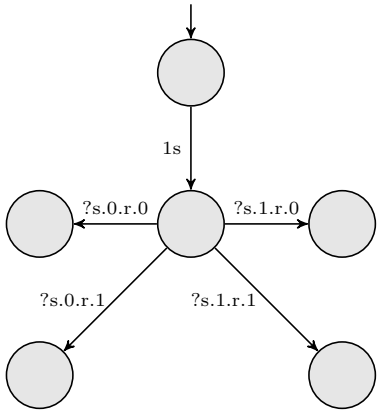

Fig. 11. The vending machine specification – TIOTS representation of delay transitions

Theorem 5.1. Soundness of *fromDFRStoTIOTS*

$$\forall\, expDFRS : e\_DFRS \bullet fromDFRStoTIOTS(expDFRS) \in TIOTS$$

The detailed proof is available in [CCS15]; here we present a proof sketch. Concerning the invariants of *TIOTS_LABELS* (reproduced below), the sets I and O are disjoint because they are defined by the auxiliary function *genAction*, which is an injection, applied to different elements: the value of DFRS input and output variables, respectively.

$$\mathsf{disjoint}\ \langle I, O\rangle$$
$$D \in tiots\_time\_compatible$$

The TIOTS delays are compatible (all of them are discrete or dense) because the e-DFRS delays are time compatible, and the TIOTS delays preserve the delay type (discrete and dense delays in an e-DFRS are translated to discrete and dense delays in a TIOTS, respectively).

The invariant of *TIOTS_STATES* ($q_0 \in Q$) also holds, since the states of a TIOTS (Q) are defined by *fromDFRStoTIOTS* as the union of the application of *getStates* with its initial state (q_0). Therefore, it is valid that q_0 is an element of Q.

Concerning the invariants of *TIOTS* (reproduced below), as Q is obtained from all states mentioned by T, it is trivial that all states related by T belong to Q.

$$\forall\, entry : T \bullet \{entry.1, entry.3\} \subseteq Q \land (entry.2, I, O, D) \in well\_typed\_tiots\_transition$$

To be well typed, an input transition must be labelled with an element of I, and an output transition with an element of O. Similarly, a delay transition must be labelled with an element of D. As the sets I, O, and D are defined in terms of the labels used on the TIOTS transitions, this invariant also holds. Therefore, we conclude that the function *fromDFRStoTIOTS* is also sound.

5.2. Practical validation: compatibility between test cases and DFRSs

Here, we provide an empirical argument as to whether the DFRS models are expressive enough to represent the behaviour of a timed reactive system as defined using natural language. We assess whether test cases, either independently written by domain specialists from industry or generated by a commercial tool (RT-Tester[9]) from the same set of requirements, are compatible with the corresponding DFRS models. Compatibility requires that there is a sequence of delay and function transitions of the e-DFRS, which is obtained from the s-DFRS generated from the natural-language requirements, that illustrates the delays, the system inputs and the expected outputs described in the test case. In other words, in our evaluation, we generate test cases from a set of requirements, without the aid of DFRS models. Later, we use the same

[9] www.verified.de/products/rt-tester/

Table 2. Performance metrics

	VM	**NPP**	**PC**	**TIS**
Time to process the requirements:	0.11s	0.07s	0.14s	0.35s
Time to identify the requirement frames:	0.10s	0.14s	0.10s	0.20s
Time to generate the s-DFRSs:	0.02s	0.01s	0.01s	0.01s
Total time:	0.23s	0.22s	0.25s	0.56s

requirements to derive s-DFRS models. Finally, we check whether the tests can be obtained from simulation of the corresponding e-DFRS models. This analysis considers examples from four different domains.

- Vending machine (toy example): this vending machine (VM) is an adaptation of the coffee machine presented in [LMN04]. As explained in Section 2, this machine dispenses weak and strong coffee depending on the amount of time elapsed between inserting a coin and requesting the coffee. The system has two input signals (the coin sensor, and the coffee request button) and two output signals (the system mode, and the coffee machine output).
- Nuclear power plant (toy example): we consider a simplified version of a control system for safety injection in a nuclear power plant (NPP) as described in [LH03]. This is a system that controls the injection of coolant in the reactor. If the water pressure is too low (less than 900 units), the system injects coolant into the reactor, otherwise there is no need to inject coolant. This system has three input signals: the actual water pressure, a switch to block the injection of coolant, and a switch that reset the system after blockage. There are three output signals: the safety injection mode, the current blockage mode, and the pressure mode.
- Priority command (provided by Embraer): the priority command function (PC) decides whether the pilot or copilot will have priority in controlling the airplane side sticks. The system monitors whether the pilot and copilot side sticks are in the neutral position, and whether the side stick priority button has been pressed. Taking into account this information, a control logic is applied to decide who has priority. This system has four input signals (the stick position and the status of the priority button for both the pilot and the co-pilot) and one output signal (a priority command).
- Turn indicator system (from Mercedes): we have also considered a simplification of the turn indicator system (TIS) specification that is currently used by Daimler for automatically deriving test cases, concrete test data and test procedures. In 2011 Daimler allowed the publication of this specification to serve as a "real-world" benchmark supporting research on MBT techniques. Our simplification results in a size reduction of the original model presented in [Pel11], but serves well as a proof of concept, because it still is a safety-critical system component with real-time and concurrent aspects. The system has three inputs: (1) the turn indicator lever, which may be in the idle, left or right position; (2) the emergency flashing button; and (3) the battery voltage. The system outputs are the car flashing lights. The simplified TIS comprises two parallel components: the *flashing mode* component, which is responsible for controlling the system flashing state (only left or right lights flashing, left and right lights flashing, left or right tip flashing, and no lights flashing), and the *lights controller* component, which is responsible for turning on and off the flashing lights respecting the flashing periods: 320 milliseconds on and 240 milliseconds off.

Using the mechanisation of our strategy presented in Sections 3.5 and 4.5, we have derived DFRS models from the natural-language requirements of the aforementioned four examples. Table 2 presents the metrics related to our performance analysis. All time measurements are for experiments using an i3 CPU with 2.27GHz equipped with 4 GB of RAM memory running the Ubuntu 14.04 LTS operating system. As can be seen in Table 2, the time required to process the system requirements and to deliver the corresponding s-DFRSs models is low (less than 1s). Furthermore, it is worth noting that the total time required increases linearly according to the size of the specification. Therefore, we can expect that the proposed approach might scale for larger examples.

Afterwards, we have assessed whether test cases, either independently written or generated from the same set of requirements, are compatible with the corresponding DFRS models. To analyse whether the test cases are compatible with the corresponding DFRS models, we have developed a depth-first search algorithm that explores the state space of an e-DFRS guided by a test case. We provide to the model the inputs described

Table 3. Example of test case

TIME (ms)	Volt.	Emerg. Button	Turn Indic.	L. Lights	R. Lights
0	80	off	right	off	off
7918	81	off	left	on	off
8258	81	off	left	off	off
8478	81	off	left	on	off

Table 4. Metrics concerning compatibility analysis

	VM	NPP	PC	TIS
# of requirements:	6	11	8	21
# of words:	228	268	294	942
# of test vectors:	5	16	401	83
# of compatible test vectors:	5 (100%)	16 (100%)	401 (100%)	83 (100%)
Time for analysis:	9ms	52ms	184ms	192ms

by each test vector, and check whether the outputs provided by the system are equal to those in the vector. This comparison is straightforward (that is, the test oracle is trivial) since we are dealing with primitive types.

Table 3 shows a test case comprising four test vectors (one in each line), for the turn indicator system. The first line tests that no lights are turned on, even if, for instance, the turn indicator is on the right position, when the car voltage is too low (below 81 volts). However, when the voltage is greater than 80 (Line 2), the lights are turned on based on the turn indicator position (in this case, left), and the light remains on for 340ms, and off for 220ms, periodically. In Table 3, *Volt.*, *Emerg. Button*, *Turn Indic.*, *L. Lights*, and *R. Lights* refer to *Voltage*, *Emergency Button*, *Turn Indicator*, *Left Lights*, and *Right Lights*, respectively. The first line of the test is derived from the initial state of the obtained s-DFRS. The next three lines are derived from six alternating delay and function transitions.

1. Delay transition: after 7918ms, the voltage becomes 81, and the turn indicator changes to the left position;
2. Function transition: the system reacts turning on the left lights;
3. Delay transition: the input signals remain the same for 340ms;
4. Function transition: the system reacts turning off the left lights;
5. Delay transition: the input signals remain the same for 220ms;
6. Function transition: the system reacts turning on the left lights.

The selected set of test cases is relevant as discussed in details in [CBL$^+$14]. They are able to detect a significant number of errors introduced by mutation testing (for instance, roughly 98% of the mutants generated for the TIS example). Table 4 presents some metrics concerning our compatibility analysis. The verdict of our testing experiments have been successful, since all selected test cases are compatible with the corresponding DFRS models, which gives evidence that these models for the four examples indeed capture the underlying semantics of the natural-language requirements as suggested in this paper.

6. Related work

A classical notation for modelling timed reactive systems is timed automata. DFRS, however, is specifically designed to facilitate automatic generation of formal models from natural-language requirements. In particular, DFRS is tailored for embedded systems whose inputs and outputs are always available, as signals. The 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 of the models that are required of reactive embedded systems. For example, DFRS models enforce the principle of

Table 5. Analysis of related work

	Domain	**Input**	**Model**	**Data**	**Time**	**Requirement analyses**
[LCK98]	General	Use cases	CMPN	No	No	Consistency Completeness
[LSL+14]	General	Use cases	Activity diagram	No	No	Consistency Integrity
[Sch02]	General	NL requirements	FOL	No	No	Not reported
[ADS14]	General	NL requirements	CCM	Yes	No	Off-nominal
[BCMW15]	Embedded systems	NL requirements	AADL	Yes	No	Realisability
[BGMC04]	General	NL requirements	FMONA	Yes	No	Consistency
[NSM14]	Mobile app.	Use cases	CSP	Yes	No	Not reported
[ES07]	General	NL requirements	FRL	No	Yes	Not reported
[SHG10]	General	NL requirements	TUM	No	Yes	Consistency Completeness
[AG06]	General	NL requirements	CNM	Yes	Yes	Consistency Completeness Ambiguity
[Ili07]	General	NL requirements	B method	Yes	Yes	Consistency
[LHHR94]	Embedded systems	NL requirements	RSML	Yes	Yes	Consistency Completeness
[MTWH06]	Embedded systems	NL requirements	RSML^{-e}	Yes	Yes	Consistency Completeness Reachability
[Sch09]	Automotive systems	NL requirements	TQE	Yes	Yes	Reachability
[SJV12]	General	NL requirements	Statecharts	Yes	Yes	Not reported
NAT2TEST	Embedded systems	NL requirements	DFRS	Yes	Yes	Consistency Completeness Reachability Time lock

delayable transitions; use delay transitions to represent environment sitimuli and, thus, they cannot range over the output signals and timers of the system; include no self-transitions; ensure that there are no delay and function transitions emanating from any state. If we were to use general purpose notations such as timed automata to capture the natural-language requirements, the translation would be more complicated and costly.

Previous works have already investigated and proposed formal models for describing natural-language requirements. Here, we analyse such works from six distinct perspectives: (1) domain: whether the modelling approach is tailored for a specific domain; (2) input: how the system requirements are documented; (3) model: the underlying formal notation used to represent the system behaviour; (4) data: whether this notation can explicitly deal with variables; (5) time: whether this notation can explicitly deal with temporal behaviour; and (6) requirement analyses: which properties of the requirements can be analysed via this notation. Table 5 summarises our analyses of related work considering these six perspectives.

Some notations only consider the occurrence of events (e.g., the button has been pressed, the voltage is higher than 10), as opposed to others that have an explicit model of variables and values. The fundamental difference between these two approaches is that the second one is easier to connect with generation of automated test cases, since data (variables and values) are embedded in the model. However, as a drawback, if one considers a large amount of variables and possible values, the number of possibilities can be a problem to deal with, when symbolic techniques are not used. As the ultimate goal of our work is the generation of

test cases, we consider as more appropriate for our purposes the second approach, when variables and values are part of the model.

Similarly, as we want to model and test temporal aspects of systems, which can be discrete or continuous, we also want to incorporate time as an element of the model. Some approaches consider limited temporal analysis, for instance, when representing and verifying Linear Temporal Logic (LTL) properties. Here, we do not consider these works as allowing time modelling, since only the sequencing of events is considered.

Considering these remarks, we group similar works into three distinct categories (see Table 5). While the first group comprises techniques that do not support data and time information on requirements, the second one supports at least one of these two concepts. The last group, to which our approach belongs, supports both of them. In what follows, we summarise the previous studies, while comparing them with our own work.

While some approaches are tailored for use cases described in natural-language [LCK98, LSL$^+$14, NSM14], processing natural-language requirements is more common. In [LCK98], a variant of Petri Nets (Constraints-based Modular Petri Nets – CMPNs) is proposed for modelling use cases. To generate the corresponding CMPN model, one needs to fill an action-condition table manually, besides clarifying event names, which represent the actions described in the use cases. There is no support for data and time. On the other hand, using the CMPN model, it is possible to perform consistency and completeness analyses automatically.

In [LSL$^+$14] and [NSM14], use cases are used as source for the generation of formal models: the former uses a restricted and formal version of activity diagrams, and the latter CSP. The CNL considered by [NSM14] is tailored for mobile applications, whereas the strategy of [LSL$^+$14] is for general purpose. Data is considered in [NSM14] via annotations in the use cases. Only events are considered by [LSL$^+$14], where it is also shown how the derived activity diagrams can be used to verify the consistency and integrity of requirements.

[Sch02] proposes a computer-processable CNL for writing unambiguous and precise requirements: PENG. The specification written in PENG can be deterministically translated into first-order predicate logic (FOL). Data and time aspects are not considered, nor is the analysis of properties of the requirements.

In [ADS14], a Casual Component Model (CCM) is used to model the behaviour described by natural-language requirements. This formal model needs to be manually created from the specification. In CCM, the states can be used to model valuations of a variable (e.g., s1 – switch(off), s2 – switch(on)), from which a NuSMV specification [CCGR99] is automatically derived. Then, temporal logic can be used to seek off-nominal (undesired) behaviour.

The approach of [BCMW15] also considers variables, and it is tailored for embedded systems. It uses as internal notation AADL (Architecture Analysis and Design Language), where assume-guarantee contracts are manually created. In this work, it is possible to assess whether these contracts and the corresponding requirements are realisable. Differently from other approaches, the authors show how to perform this analysis in a compositional way.

The work reported in [BGMC04] presents a requirements analysis tool called RETNA. This tool accepts natural-language requirements and, with user interaction, it translates the requirements into a logical notation: FMONA, which is a high-level language for describing weak monadic second-order logic. This model can then be used to analyse whether the natural-language requirements are consistent.

In [ES07] requirements are written in a limited standardized format. The requirements need to be written according to a strict if-then sentence template, which, however, can be used to represent time properties. Despite describing how to translate these templates to the Formal Requirement Language (FRL), the work does not elaborate on how this model could be used to check properties of the requirements. In [SHG10] it is also possible to represent timed-behaviour using Timed Usage Models (TUM), which are Markov Chain Usage Models (MCUM) with time information. This model is manually created from the system requirements. Consistency and completeness properties can be verified automatically. Differently from other approaches, this model can also take into account probabilistic properties.

The works analysed so far do not take into account both data and time aspects and, thus, differ from our approach: the DFRS models consider both of them. The approaches proposed in [AG06, Ili07, LHHR94, MTWH06, Sch09, SJV12] are closer related to our work, since they share the fact that data and time are both supported.

The CIRCE environment, which enables analysis of natural-language requirements, is presented in [AG06]. In this environment, requirements are interpreted according to the CIRCE Native Meta-Model (CNM). Besides analysing properties of requirements, this environment allows the generation of UML models and code from the CNM notation. Differently from our strategy, it requires manual effort when modelling the system requirements. Here, one needs to create by hand designations and definitions. The former establishes equivalences between different terms that refer to the same entity, and the latter establishes notations for

expressing requirements in a succinct way. While these two elements have a well-defined and formal structure, the requirements description statements are expected to be free-form text.

In [Ili07], the B method [Abr96] is used to construct formal models for system requirements. In this work, the requirements need to be translated to predefined templates for describing events, data and time, from which B specifications are systematically translated. Each template defines the information that is mandatory. Comparing to our approach, the thematic roles are the counterpart of these templates. As just said, in [Ili07], one needs to classify manually the requirements according to these templates, besides filling them. Differently, thematic roles are automatically inferred from the requirements in the NAT2TEST strategy.

In [LHHR94, MTWH06], the Requirements State Machine Language (RSML) is used as formal model for natural-language requirements. A restricted version of this notation (RSML^{-e}) is adopted by [MTWH06], where events are not allowed. This notation, which is argued to be tailored for embedded systems, is used to document the system requirement. In [LHHR94], this internal model is analysed by proposed algorithms to check whether the requirements are consistent and complete. In [MTWH06], these analyses are automated with the aid of NuSMV and PVS [ORS92]. These two studies require user intervention to classify and edit requirements. This is not a necessity within the NAT2TEST strategy.

In [Sch09], assuming that the system specification is manually represented conforming to a set of templates, developed for automotive systems, a Temporal Qualified Expression (TQE) is derived. It is also necessary to identify manually signal names along with its possible values. Differently, in our approach, the signal names, their types, and possible values are automatically inferred.

In [SJV12], the SOLIMVA methodology is presented. The methodology has tool support to translate automatically natural-language requirements into statechart models. Another tool (GTSC) is used to generate test cases. In this work, besides writing the requirements, one needs to identify and partition inputs and outputs. This is not required in our approach. Moreover, this work does not explain how the statechart models can be used to analyse requirements, but considers this as a possible line for future work.

In summary, the NAT2TEST strategy formally describes system requirements using DFRS models, which are explained in details here. Similarly to other approaches previously identified, the formal model used to represent the system behaviour considers both data and time information. It can also be used to check system properties such as consistency, completeness, reachability, and absence of time lock. Our work stands out from the similar approaches reported here by the richness of the model generated solely from natural-language requirements without user intervention.

The absence of user intervention in our strategy is a consequence of the compromise reached by the defined controlled natural language. As we focus on a specific domain of embedded systems, whose behaviour can be described as actions guarded by conditions, we can impose some restrictions, while allowing the requirements to be expressed as a textual specification, and automatically obtain a formal model from these requirements. However, these restrictions make our approach not suitable for writing requirements that do not adhere to this format of actions and guards.

7. Conclusions

We have presented a symbolic formalism (s-DFRS) for modelling timed-systems, which is suitable to describe formal models that can be automatically obtained from natural-language requirements. We have also shown that an expanded version of a symbolic DFRS model can be dynamically generated and used for checking properties such as consistency, completeness and reachability of requirements, as well as the absence of time lock. To connect the proposed models to established ones in the literature, we have also presented how e-DFRSs, the notation used to describe the expanded models is called e-DFRS, can be encoded as Timed Input-Output Transition System (TIOTS): an alternative timed model based on the widely used IOLTS and ioco. All definitions have been formally described in Z, and checked with the aid of the CZT plug-in for Eclipse.

We have also considered examples from four different domains, and showed that the derived DFRS models are expressive enough to represent a set of independently written and generated test cases. To support this analysis, we have developed a tool NAT2TEST that automatically generates s-DFRS models from natural-language requirements, besides other features such as dynamic exploration of the e-DFRS state space.

DFRS models are suitable for modelling embedded systems whose inputs and outputs are always available,

as signals. Moreover, the system reactions are described as assignments guarded by static and timed guards. Further empirical analysis is a topic for future work.

We also envisage the following tasks as future work.

- Integrate this work with our previous work described in [CFaB+13, CBL+14] to take advantage of the generality of DFRSs as indicated in Figure 1;
- Enhance the DFRS models to deal with more complex structures. For instance, currently, only literals are allowed in assignments;
- Evolve the DFRS models to deal with new classes of timed-systems, namely, hybrid systems. To achieve such a goal, we plan to incorporate into our models differential equations to describe the continuous evolution of variables.

Despite the potential for improvement, we believe that the results obtained give some evidence that DFRSs are a promising modelling notation for describing the behaviour ot timed-systems, in particular, when this behaviour is defined using natural-language requirements of the form of actions guarded by conditions. A detailed account of our models, notations, tools, and examples are found in [Car16].

Acknowledgments

This work was carried out with the support of the CNPq (Brazil), INES[10], and the grants: FACEPE 573964/2008-4, APQ-1037-1.03/08, CNPq 573964/2008-4 and 476821/2011-8. The research reported in this paper was also partially funded by the UK EPSRC.

References

[ABJ+15] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification and Reliability*, 25(8):716–748, 2015.

[Abr96] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[ADS14] Daniel Aceituna, Hyunsook Do, and Sudarshan Srinivasan. A Systematic Approach to Transforming System Requirements into Model Checking Specifications. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 165–174, New York, NY, USA, 2014. ACM.

[AG06] Vincenzo Ambriola and Vincenzo Gervasi. On the Systematic Analysis of Natural Language Requirements with CIRCE. *Automated Software Engineering*, 13(1):107–167, 2006.

[All95] James Allen. *Natural Language Understanding*. Benjamin/Cummings, 1995.

[BBF97] M Blackburn, R Busser, and J Fontaine. Automatic Generation of Test Vectors for SCR-style Specifications. In *Annual Conference on Computer Assurance*, 1997.

[BCMW15] John Backes, Darren Cofer, Steven Miller, and MichaelW. Whalen. Requirements Analysis of a Quad-Redundant Flight Control System. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 9058 of *Lecture Notes in Computer Science*, pages 82–96. Springer International Publishing, 2015.

[BGFT10] Antonio Bucchiarone, Stefania Gnesi, Alessandro Fantechi, and Gianluca Trentanni. An experience in using a tool for evaluating a large set of natural language requirements. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 281–286, New York, NY, USA, 2010. ACM.

[BGMC04] R. Boddu, L. Guo, S. Mukhopadhyay, and Bojan Cukic. RETNA: from Requirements to Testing in a Natural Way. In *IEEE International Requirements Engineering Conference*, pages 262–271, 2004.

[Car16] Gustavo Carvalho. *NAT2TEST: Generating Test Cases from Natural Language Requirements based on CSP*. PhD thesis, Centro de Informática, Universidade Federal de Pernambuco (UFPE), Brazil, 2016.

[CBC+15] Gustavo Carvalho, Flávia Barros, Ana Carvalho, Ana Cavalcanti, Alexandre Mota, and Augusto Sampaio. NAT2TEST Tool: from Natural Language Requirements to Test Cases based on CSP. In *International Conference on Software Engineering and Formal Methods*. Springer International Publishing, 2015.

[CBL+14] Gustavo Carvalho, Flávia Barros, Florian Lapschies, Uwe Schulze, and Jan Peleska. Model-Based Testing from Controlled Natural Language Requirements. In Cyrille Artho and Peter Csaba lveczky, editors, *Formal Techniques for Safety-Critical Systems*, volume 419 of *Communications in Computer and Information Science*, pages 19–35. Springer International Publishing, 2014.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 495–499, London, UK, UK, 1999. Springer-Verlag.

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

[CCR+14] Gustavo Carvalho, Ana Carvalho, Eduardo Rocha, Ana Cavalcanti, and Augusto Sampaio. A Formal Model for Natural-Language Timed Requirements of Reactive Systems. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering, International Conference on Formal Engineering Methods ICFEM*, volume 8829 of *Lecture Notes in Computer Science*, pages 43–58. Springer International Publishing, 2014.

[CCS15] Gustavo Carvalho, Ana Cavalcanti, and Augusto Sampaio. DFRS: Definition and Proofs. Technical report, Universidade Federal de Pernambuco, 2015.

[CFaB+13] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn. Test Case Generation from Natural Language Requirements based on SCR Specifications. In *Symposium on Applied Computing*, volume 2, pages 1217–1222, 2013.

[CFB+14] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn. NAT2TEST$_{SCR}$: Test case generation from natural language requirements based on SCR specifications. *Science of Computer Programming*, 95, Part 3(0):275 – 297, 2014.

[CSM13] Gustavo Carvalho, Augusto Sampaio, and Alexandre Mota. A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. In *Formal Methods and Software Engineering*, volume 8144 of *LNCS*, pages 148–164. Springer Berlin Heidelberg, 2013.

[ES07] M. Esser and P. Struss. Obtaining Models for Test Generation from Natural-Language like Functional Specifications. In *International Workshop on Principles of Diagnosis*, pages 75–82, 2007.

[FAA09] FAA. Requirements Engineering Management Findings Report. Technical report, U.S. Department of Transportation - Federal Aviation Administration, 2009.

[Fil68] Charles J. Fillmore. The Case for Case. In Bach and Harms, editors, *Universals in Linguistic Theory*, pages 1–88. New York: Holt, Rinehart, and Winston, 1968.

[FLGS14] A. Ferrari, G. Lipari, S. Gnesi, and G. O. Spagnolo. Pragmatic ambiguity detection in natural language requirements. In *Artificial Intelligence for Requirements Engineering (AIRE), 2014 IEEE 1st International Workshop on*, pages 1–8, Aug 2014.

[Ili07] D. Ilic. Deriving Formal Specifications from Informal Requirements. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 145–152, July 2007.

[ISO02] ISO. Z formal specification notation (ISO/IEC 13568). Technical report, International Organization for Standardization, 2002.

[LCK98] Woo Jin Lee, Sung Deok Cha, and Yong Rae Kwon. Integration and analysis of use cases using modular Petri nets in requirements engineering. *Software Engineering, IEEE Transactions on*, 24(12):1115–1130, Dec 1998.

[LH03] Elizabeth Leonard and Constance Heitmeyer. Program Synthesis from Formal Requirements Specifications Using APTS. *Higher Order Symbol. Comput.*, 16:63–92, 2003.

[LHHR94] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements Specification for Process-Control Systems. *IEEE Trans. Softw. Eng.*, 20(9):684–707, Sep 1994.

[LMN04] Kim Larsen, Marius Mikucionis, and Brian Nielsen. Online Testing of Real-time Systems using Uppaal: Status and Future Work. In *Perspectives of Model-Based Testing - Dagstuhl Seminar*, volume 04371, 2004.

[LSL+14] Shuang Liu, Jun Sun, Yang Liu, Yue Zhang, Bimlesh Wadhwa, Jin Song Dong, and Xinyu Wang. Automatic Early Defects Detection in Use Case Documents. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 785–790, New York, NY, USA, 2014. ACM.

[MTWH06] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P.E. Heimdahl. Proving the shalls. *International Journal on Software Tools for Technology Transfer*, 8(4-5):303–319, 2006.

[NSM14] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Test generation from state based use case models. *Formal Aspects of Computing*, 26(3):441–490, 2014.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[Pel11] Peleska, J. et al. A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain. In *Proceedings of the ICTSS*, ICTSS'11, pages 146–161, Berlin, Heidelberg, 2011. Springer-Verlag.

[PVLZ11] Jan Peleska, Elena Vorobev, Florian Lapschies, and Cornelia Zahlten. Automated Model-Based Testing with RT-Tester. Technical report, Universität Bremen, 2011.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[Sch02] R. Schwitter. English as a Formal Specification Language. In *Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, 2002.

[Sch09] Matthias Schnelte. Generating Test Cases for Timed Systems from Controlled Natural Language Specifications. In *International Conference on System Integration and Reliability Improvements*, pages 348–353, 2009.

[SHG10] S. Siegl, K.-S. Hielscher, and R. German. Model Based Requirements Analysis and Testing of Automotive Systems with Timed Usage Models. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 345–350, Sept 2010.

[SJV12] Valdivino Santiago Junior and Nandamudi Lankalapalli Vijaykumar. Generating Model-based Test Cases from Natural Language Requirements for Space Application Software. *Software Quality Journal*, 20:77–143, 2012.

[SLDP09] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[Tre99] Jan Tretmans. Testing Concurrent Systems: A Formal Approach. In *Proceedings of CONCUR*, pages 46–65, London, UK, UK, 1999. Springer-Verlag.