A Formal Model for Natural-Language Timed Requirements of Reactive Systems

Gustavo Carvalho[1,3], Ana Carvalho[2], Eduardo Rocha[1], Ana Cavalcanti[3], and Augusto Sampaio[1]

[1]Universidade Federal de Pernambuco - Centro de Informática, 50740-560, Brazil
[2]Universidade Federal de Pernambuco - NTI, 50670-901, Brazil
[3]University of York - Department of Computer Science, YO10 5GH, UK
{ghpc,ebr,acas}@cin.ufpe.br,ana.alves@ufpe.br,ana.cavalcanti@york.ac.uk

Abstract. To analyse the behaviour of reactive systems formally, it is necessary to build a model. At the very beginning of the development, typically only natural language requirements are documented. We present a formal model, named Data-Flow Reactive Systems (DFRS), which can be automatically obtained from natural language requirements that may also describe temporal properties. We prove that a DFRS can be mapped to a timed input-output transition system, which is widely used to characterise conformance relations for timed reactive systems. To validate the proposed model as well as the mechanisation developed to support its analysis, 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.

Keywords: model mapping, TIOTS, test-case generation

1 Introduction

The need to model the behaviour of a system may become an obstacle to the use of formal methods as the requirements are commonly written in Natural Language (NL). In 2009, the Federal Aviation Administration (FAA) published a report [12] that discusses current practices concerning requirements engineering management. The report states that "*... the overwhelming majority of the survey respondents indicated that requirements are being captured as English text...*".

With this in mind, we have investigated how we can obtain formal models from NL requirements of reactive systems automatically, particularly to generate test cases. Automation is essential, since requiring knowledge of formal modelling by practitioners is often not feasible. Automation also allows an early application of formal methods within the development of reactive systems. To accomplish this goal, we have previously developed a strategy (NAT2TEST) that generates test cases from NL requirements based on different internal and hidden formalisms: SCR [14] (NAT2TEST$_{SCR}$ [8]), and IMR [18] (NAT2TEST$_{IMR}$ [6]).

Both in [8] and in [6], the input is NL requirements. The first phase of the test-generation strategy is a Syntactic Analysis to generate a syntax tree. The second

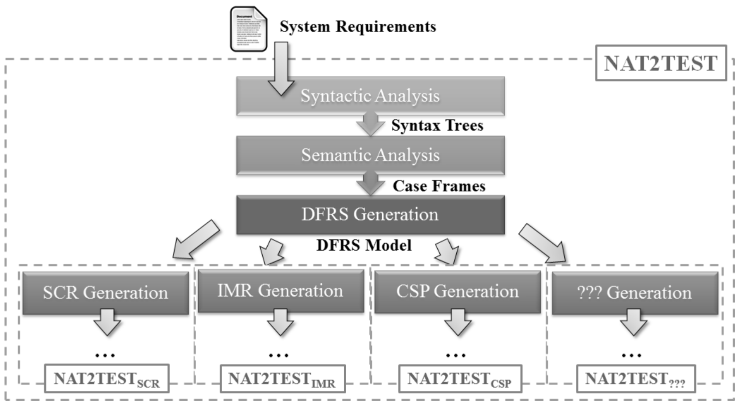

Fig. 1. The NAT2TEST Strategy.

phase is a Semantic Analysis, which maps the syntax trees into an informal semantic representation based on the Case Grammar theory [13].

Based on the experience of generating test cases using two different formal representations, and with the perspective of instantiating our approach to several other target notations, translating the NL requirements to an intermediate formal notation is a more promising alternative, since the translation from a NL is a more elaborate task. Then, from an intermediate, and formal, representation, one might explore different target notations and analyse the system from several perspectives, using different languages and tools. For example, one might want to generate SCR code and then use T-VEC [2] to generate test cases, as already mentioned, but also to analyse the completeness and disjointness of system requirements [3]. As another example, it is possible to generate CSP models and use tools like FDR[1] to prove both classical and domain specific properties of the system requirements.

Therefore, a new architecture for our strategy, which is based on the generation of an intermediate notation from NL requirements, is presented in Figure 1. Our focus here is the third step of this strategy (*DFRS Generation*) and the DFRS (Data-Flow Reactive System) model that it generates.

Our claim that a DFRS is a good candidate for such an intermediate notation comes from a theoretical and an empirical perspective. First, as we detail in this paper, 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. Second, we have so far derived two different formal models from NL requirements (namely, SCR [8] and IMR [6]), besides other notations that are currently being considered, and the DFRS model encompass the information required to derive models in these notations.

[1] https://www.cs.ox.ac.uk/projects/fdr/

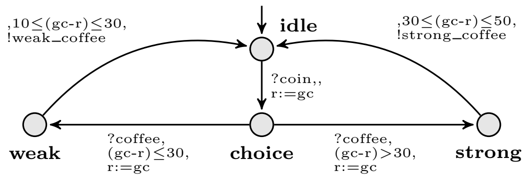

Fig. 2. The Vending Machine Specification.

In [10] we briefly present our first ideas of a DFRS as it is used as a source model to derive a CSP specification, which is later used within the context of a timed conformance relation. Here, we formalise the definition and properties of a DFRS, using Z [15] with the support of Z/EVES [19]. We also prove that a DFRS can be characterised as a Timed Input-Output Transition System

To evaluate the expressiveness of DFRSs, we consider examples from four domains: a Vending Machine (VM — toy example); a control system for safety injection in a Nuclear Power Plant (NPP — toy example), a Priority Command (PC) control provided by Embraer[2]; 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 those generated using a DFRS.

The main contributions of this paper are a formalisation of DFRSs, a theoretical and a practical analysis of these models, and a strategy to generate DFRSs from NL requirements automatically.

Next section gives the formal definition of a DFRS. Section 3 defines a TIOTS and how any DFRS can be mapped to a TIOTS. Section 4 describes how a DFRS can be automatically obtained from NL requirements. Section 5 considers the test cases of our examples for an empirical analysis of DFRSs. Finally, Section 6 presents our conclusions, and addresses related and future work.

2 Definition and Properties of a DFRS

To illustrate our work, we consider a toy example — the Vending Machine (VM) presented in Figure 2 as a timed statechart — it is an adaptation of the Coffee Machine in [16]. We present this statechart just for a concise illustration of the structure of DFRSs. The input of our strategy is NL requirements.

Initially, the VM is in an *idle* state. When it receives a coin, it goes to the *choice* state and resets the *reqTimer* (r in Figure 2) clock. This assigns the current global time (gc) to this variable. 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.

[2] www.embraer.com.br

Formally, a DFRS is a 7-tuple: $(I,\ O,\ T,\ \mathit{gcvar},\ S,\ s_0,\ \mathit{TR})$. 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 time. S denotes a (possibly infinite) set of states, s_0 is the initial state, and TR is a (possibly infinite) transition relation between states.

Below, we describe a formal definition of a DFRS available in full in [7].

2.1 Inputs, Outpus and Timers

We use a given set *NAME* containing the set of all valid variable names, and define *gc* to be the name of the system global clock ($\mathit{gc} : \mathit{NAME}$). Also *VNAME* is the set of all system variables except for the global clock ($\mathit{NAME} \setminus \{\mathit{gc}\}$).

Based on these definitions, we define *SVARS* and *STIMERS* to represent inputs and outputs (defined later as different mappings of the same type *SVARS*), and timers, respectively, as partial functions from *VNAME* to *TYPE*. In this work, we consider as valid types boolean and numerical types (*bool, int, nat, float, p_float* – where *p_float* represents non-negative floating-point numbers). We restrict our model to these types as they are sufficient to describe the considered domain of requirements – embedded reactive systems whose inputs and outputs can be seen as signals. Despite that, one can expand the model to incorporate new types.

$$\begin{aligned}
&\mathit{SVARS} == \{f : \mathit{VNAME} \pfun \mathit{TYPE} \mid f \neq \emptyset \land \operatorname{ran} f \subseteq \{\mathit{bool}, \mathit{int}, \mathit{float}\}\}\\
&\mathit{STIMERS} == \{f : \mathit{VNAME} \pfun \mathit{TYPE} \mid \operatorname{ran} f = \{\mathit{nat}\} \lor \operatorname{ran} f = \{\mathit{p\_float}\}\}
\end{aligned}$$

The functions f in *SVARS* are not empty: the system needs to have at least one input and one output variable. Differently, one can have a system without timers, that is, a DFRS whose behaviour is not dependent on time elapsing.

The possible types of an element of *SVARS* are *bool*, *int* and *float*. The types *nat* and *p_float* are used to restrict the possible values of timers since time is a non-negative number. Besides that, the type of all timers must be the same: you can analyse the behaviour of the system discretely or continuously, but not in both ways simultaneously.

Example Besides the system global clock, five variables are identified in the context of the VM example: two system inputs (*coin sensor*, *coffee request button*), two outputs (*system mode*, *coffee machine output*), and one timer (*request timer*) whose types are *bool*, *nat*, and *p_float*, respectively. □

2.2 States

A state is a relation between names and values ($\mathit{STATE} == \mathit{NAME} \pfun \mathit{VALUE}$). *VALUE* is a free type that includes booleans and numerical values. As float numbers are not part of Standard Z, we declare them as given sets. Despite being out of the scope of this work, it is possible to represent float numbers in Z. For more details, refer, for instance, to ProofPower-Z[3].

The valuation of a variable n defined to have a type t is well typed in a state s if, and only if, n belongs to the domain of s, and the value associated with n in s belongs to the set of possible values of t. The function *values* yields all possible values of a specific type t. This property of well typedness for variables in the context of a state is captured by the following predicate.

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

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.

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

The set of states is defined as a (possibly infinite) non-empty set of states ($\mathit{STATE\_SET} == \mathbb{P}_1\, \mathit{STATE}$), since it must contain at least an initial state.

Example Considering the VM example, a possible initial state of the corresponding DFRS is the following.

$$\begin{gathered}
\{(\text{coin sensor} \mapsto b(\mathit{false})), (\text{coffee request button} \mapsto b(\mathit{false}), \\
(\text{system mode} \mapsto n(1)), (\text{coffee machine output} \mapsto n(1)), \\
(\text{request timer} \mapsto p\_fl(0.0), (\text{system global clock} \mapsto p\_fl(0.0))\}
\end{gathered}$$

where b, n, and p_fl are free type constructors associated with boolean values, natural numbers, and non-negative floating-point values, respectively. We consider that *false* is used to represent that a coin was not inserted, as well as that the coffee request button was not pressed. Regarding the variables *system mode* and *coffee machine output*, 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*$\}$, and $\{0 \mapsto$*strong*, $1 \mapsto$*undefined*, $2 \mapsto$*weak*$\}$. □

2.3 Transitions

A transition relates two states by means of a label. A label represents the occurrence of a functional behaviour (*fun*) or time elapsing (*del*).

$$\begin{aligned}
&\mathit{TRANS} == (\mathit{STATE} \times \mathit{TRANS\_LABEL} \times \mathit{STATE}) \\
&\mathit{TRANS\_LABEL} ::= \mathit{fun}\langle\langle \mathit{FUNCTION\_ENTRY} \rangle\rangle \mid \\
&\qquad\qquad\qquad\qquad \mathit{del}\langle\langle \mathit{DELAY} \times \mathit{STMT\_SET} \rangle\rangle
\end{aligned}$$

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

Function Transition With the system behaviour defined as a function that describes how the system reacts in a given scenario, the occurrence of a function transition leads to the application of an entry of this function. A function entry models a scenario as a pair of static and timed guards, related to a set of statements. When both guards evaluate to true, the system reacts instantly performing the set of statements. One of the guards can be empty, but not both.

$$\begin{array}{l} \mathit{FUNCTION\_ENTRY} == \\ \quad \{\mathit{sGuard}, \mathit{tGuard} : \mathit{EXP};\ \mathit{stmts} : \mathit{STMT\_SET} \mid \mathit{sGuard} \cup \mathit{tGuard} \neq \emptyset\} \end{array}$$

The guards are expressions whose structure adheres to a Conjunctive Normal Form: a finite set of conjunctions of disjunctions, where each disjunction is a non-empty binary expression. Above, EXP refers to the set of such expressions. Each binary expression is a static or a timed expression. A binary expression is said to be static if, and only if, the name it mentions is the name of a system input or output. Otherwise, it is a timed expression. Similarly, a guard is static (or timed) if all its conjunctions and disjunctions are static (or timed). A statement ($\mathit{STMT} == \mathit{VNAME} \times \mathit{VALUE}$) is an assignment of a value to a name, and $\mathit{STMT\_SET}$ a non-empty set of statements ($\mathit{STMT\_SET} == \mathbb{F}_1\, \mathit{STMT}$).

Delay Transition Time elapsing is characterised by a delay and a set of statements, which model stimuli from the environment that happens immediately after the delay. A delay can represent a discrete or dense time elapsing. The former delay is characterised by a positive natural number ($\mathbb{N}_1$), whereas the latter by a positive float number (P_FLOAT$_1$), which is a subset of $\mathit{P\_FLOAT}$.

$$\mathit{DELAY} ::= \mathit{discrete}\langle\langle \mathbb{N}_1 \rangle\rangle \mid \mathit{dense}\langle\langle \mathit{P\_FLOAT}_1 \rangle\rangle$$

Based on these definitions, we define the DFRS transition relation as a set of transitions ($\mathit{TRANSREL} == \mathbb{P}\ \mathit{TRANS}$). Each transition must be well typed. A function transition is well typed if, and only if, the statements of its label modify only values of outputs and timers. In other words, the system does not interfere with the environment stimuli, which is modelled by the input variables.

$$\begin{array}{|l} \mathit{well\_typed\_function\_transition} : \mathbb{P}(\mathit{TRANS\_LABEL} \times \\ \quad (\mathit{VNAME} \nrightarrow \mathit{TYPE}) \times (\mathit{VNAME} \nrightarrow \mathit{TYPE}) \times \\ \quad (\mathit{VNAME} \nrightarrow \mathit{TYPE})) \\ \hline \forall\, \mathit{trans} : \mathit{TRANS\_LABEL};\ I, O : \mathit{VNAME} \nrightarrow \mathit{TYPE}; \\ T : \mathit{VNAME} \nrightarrow \mathit{TYPE} \mid \mathit{trans} \in \operatorname{ran} \mathit{fun} \bullet \\ \quad (\mathit{trans}, I, O, T) \in \mathit{well\_typed\_function\_transition} \Leftrightarrow \\ \quad (\forall\, \mathit{stmt} : (\mathit{functionTransition}(\mathit{trans})).3 \bullet \\ \qquad \mathit{stmt}.1 \in \operatorname{dom} O \cup \operatorname{dom} T) \wedge \\ \quad ((\mathit{functionTransition}(\mathit{trans})).1, I, O) \in \mathit{static\_exp} \wedge \\ \quad ((\mathit{functionTransition}(\mathit{trans})).2, T) \in \mathit{timed\_exp} \end{array}$$

Furthermore, the first guard of a function entry must be static, whereas the second must be timed. To formalise these requirements, we rely on an auxiliary function ($functionTransition : TRANS\_LABEL \pfun FUNCTION\_ENTRY$) that extracts the corresponding function entry given a transition label. We note that the notation $.i$ is used to refer to the projection of the i-th element of a tuple.

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, that is, on the occurrence of each delay transition, the system receives the current value of all its inputs.

Moreover, the delay transitions need to be compatible with the global clock, that is, if the delay is discrete, the type of the system global time must be *nat*, whereas if the delay is dense, the type of the clock must be *p_float*. As a consequence, all delay transitions share the same type of delay, that is, they are all discrete or dense. This is captured by the *clock_compatible_transition* property.

$$
\begin{array}{|l}
clock\_compatible\_transition : \mathbb{P}(TRANS\_LABEL \times (\{gc\} \to TYPE)) \\
\hline
\forall trans : TRANS\_LABEL;\ gcvar : \{gc\} \pfun TYPE \bullet \\
\quad (trans, gcvar) \in clock\_compatible\_transition \Leftrightarrow trans \in \operatorname{ran} del \land \\
\quad (((delayTransition(trans)).1 \in \operatorname{ran} discrete \land \operatorname{ran} gcvar = \{nat\}) \lor \\
\quad ((delayTransition(trans)).1 \in \operatorname{ran} dense \land \operatorname{ran} gcvar = \{p\_float\}))
\end{array}
$$

From these definitions, a transition is said to be well typed (*well_typed_transition*) if it satisfies the restrictions for function and delay transitions defined above.

Example If s_0 is the initial state presented in Section 2.2, inserting a coin after 3.14 time units is represented by the following entry in the transition relation. $(s_0, del(dense(3.1), \{(\text{coin sensor}, b(true)), (\text{coffee request button}, b(false))\}), s_1)$. It leads to a new state, named s_1 □

2.4 Complete Definition

The variables of a DFRS (I, O, T, and $gcvar$) are defined by the *DFRS_Variables* schema. It defines that the set of inputs, outputs and timers are disjoint, and the type of the timers is equal to that of the system global clock.

$$
\begin{array}{|l}
\multicolumn{1}{l}{DFRS\_Variables} \\
\hline
I, O : SVARS;\ T : STIMERS;\ gcvar : \{gc\} \to \{nat, p\_float\} \\
\hline
\mathsf{disjoint}\ \langle \operatorname{dom} I, \operatorname{dom} O, \operatorname{dom} T \rangle \land \operatorname{ran} T = \operatorname{ran} gcvar \\
\hline
\end{array}
$$

The initial state and the set of states of a DFRS (s_0, S) are defined by the following schema. The initial state of the DFRS is an element of its set of states.

$$DFRS\_States == [\, S : STATE\_SET;\ s_0 : STATE \mid s_0 \in S \,]$$

The transition relation (TR) is defined in *DFRS_TransitionRelation*, which establishes that for each state, it is not possible to have both function and delay transitions; that is, or the system receives stimuli from the environment or reacts to it, but not both at the same state.

$$
\begin{array}{|l}
\hline
\textit{DFRS\_TransitionRelation} \\
TR : \textit{TRANSREL} \\
\hline
\forall\, entry1, entry2 : TR \mid entry1.1 = entry2.1 \bullet \\
\quad \{entry1.2, entry2.2\} \subseteq \operatorname{ran} fun \lor \{entry1.2, entry2.2\} \subseteq \operatorname{ran} del \\
\hline
\end{array}
$$

Finally, a DFRS is defined formally by the following schema that includes the three previous schemas. It establishes that each state in S is well typed with respect to the system variables. As a consequence we impose that the same name cannot be associated with values of different types in different states. We also enforce that TR relates states of S, and each transition is well typed.

$$
\begin{array}{|l}
\hline
\textit{DFRS} \\
\textit{DFRS\_Variables} \\
\textit{DFRS\_States} \\
\textit{DFRS\_TransitionRelation} \\
\hline
\forall\, s : S \bullet (s, I \cup O \cup T) \in \textit{well\_typed\_state} \\
\forall\, entry : TR \bullet \{entry.1, entry.3\} \subseteq S \land \\
\quad (entry.2, I, O, T, gcvar) \in \textit{well\_typed\_transition} \\
\hline
\end{array}
$$

This structure is rich enough to represent requirements written using several different sentence formations in the context of a variety of application domains.

3 Theoretical Validation: Mapping DFRSs to TIOTSs

An important validation is the definition of the semantics of a DFRS. We show here that any DFRS can be mapped to a corresponding TIOTS.

3.1 Definition and Properties of a 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 actions and O output actions, D is a set of delays, and T is a (possibly infinite) transition relation relating states.

Inputs and Outputs *TIOTS_ACTION* is a given set of all valid actions, that is, inputs and outputs, and *TIOTS_ACTIONS* the set of sets of actions.

Delays A TIOTS delay represents a discrete or a dense time elapsing, but differently from a DFRS delay, a delay in a TIOTS can also be 0.

$$\textit{TIOTS\_DELAY} ::= \textit{tiots\_discrete}\langle\!\langle \mathbb{N} \rangle\!\rangle \mid \textit{tiots\_dense}\langle\!\langle \textit{P\_FLOAT} \rangle\!\rangle$$

TIOTS_DELAYS is defined as a set of *TIOTS_DELAY*.

States A state of a TIOTS is an element of the given set $\mathit{TIOTS\_STATE}$, and $\mathit{TIOTS\_STATE\_SET}$ is a non-empty set of states ($\mathbb{P}_1\ \mathit{TIOTS\_STATE}$).

Transition Relation The transition relation ($\mathit{TIOTS\_TRANSREL}$) relates two states by means of a label ($\mathit{TIOTS\_TRANS\_LABEL}$). A label may concern an input or output action, a delay, or an internal invisible action (τ – *tau*).

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

Complete Definition $\mathit{TIOTS\_Variables}$ defines input and output actions as disjoint sets, besides defining a set of delays, which needs to be time compatible ($\mathit{tiots\_time\_compatible}$): all delays must be of the same type (discrete or dense).

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{TIOTS\_Variables}} \\
\hline
I, O : \mathit{TIOTS\_ACTIONS};\ D : \mathit{TIOTS\_DELAYS} \\
\hline
\mathsf{disjoint}\ \langle I, O \rangle \land D \in \mathit{tiots\_time\_compatible} \\
\hline
\end{array}$$

A TIOTS comprises a set of states and the initial state is in this set.

$$\mathit{TIOTS\_States} == [\, Q : \mathit{TIOTS\_STATE\_SET};\ q_0 : \mathit{TIOTS\_STATE} \mid q_0 \in Q \,]$$

Finally, a TIOTS is defined by the schema below, which requires that each transition relates states of Q and is well-typed ($\mathit{well\_typed\_tiots\_transition}$), that is, comprises elements of I, O, or D. In other words, an input transition must be labelled by an input action, and an output transition by an output action. Similarly, a delay transition must be labelled by an element of D.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{TIOTS}} \\
\hline
\mathit{TIOTS\_Variables} \\
\mathit{TIOTS\_States} \\
T : \mathit{TIOTS\_TRANSREL} \\
\hline
\forall\, \mathit{entry} : T \bullet \{\mathit{entry}.1, \mathit{entry}.3\} \subseteq Q \land \\
\quad (\mathit{entry}.2, I, O, D) \in \mathit{well\_typed\_tiots\_transition} \\
\hline
\end{array}$$

It also makes sense to constraint a TIOTS by other properties [21]: time additivity ($\mathit{time\_additivity\_TIOTS}$), null delay ($\mathit{null\_delay\_TIOTS}$), and time determinism ($\mathit{time\_determinism\_TIOTS}$). Informally, time additivity states that if a state can be reached by two consecutive delay transitions, then it can also be reached by just one delay transition whose delay is equal to the sum of the original delays. The second property enforces that two states related by a zero delay transition are the same. Time determinism ensures that if two states can be reached by the same amount of delay, then they are the same too.

3.2 From DFRSs to TIOTSs

The function *fromDFRStoTIOTS* maps a DFRS to a TIOTS. Figure 3 presents an informal overview of this mapping process. The states of a DFRS are mapped to TIOTS states (for instance, $1 \mapsto A$; $2 \mapsto B$; $3 \mapsto C$), but some new states are also introduced in the TIOTS (these are unnamed in Figure 3). The delay of a delay transition (DFRS) is straightforwardly mapped to a delay in the TIOTS (for example, $tiots\_del(tiots\_dense(3.14))$), whereas the statements of a function or a delay transition (DFRS) are mapped to a chain of transitions such that each one corresponds to an output or an input action (TIOTS), respectively (for instance, $(coin, b(true)) \mapsto in(coin\_true)$; $(mode, n(0)) \mapsto out(mode\_0)$).

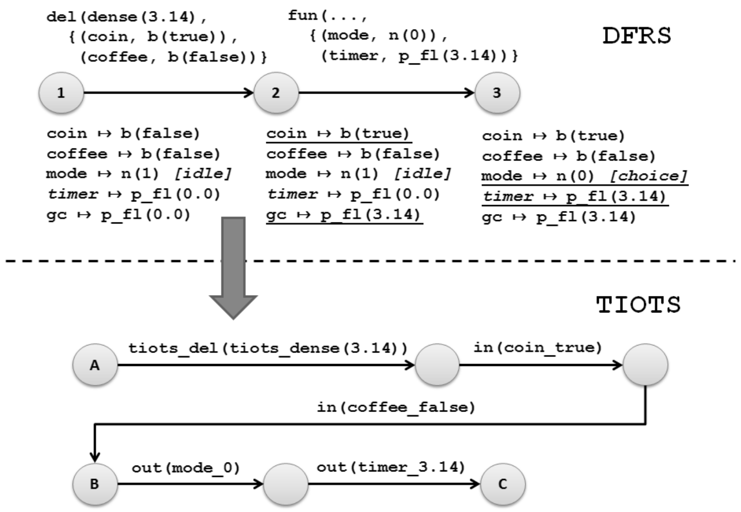

Fig. 3. From DFRS to TIOTS – Mapping Transitions.

The delay transition of the DFRS is mapped to a chain of three transitions (from states A to B) with two new intermediate states. The first transition represents the time elapsing, whereas the next two represents the stimuli from the environment. Similarly, the function transition of the DFRS is mapped to a chain of two transitions (from B to C). Next, we formalise the mapping process.

TIOTS – Inputs, Outputs and Delays The set of input actions is derived from the statements of a delay transition as they represent stimuli provided by the environment. This is formalised by the function *mapInputActions*. Similarly, the output actions are derived from the statements of a function transition as it represents a functional response of the system for a given context. The function *mapStatement* maps a DFRS statement into a TIOTS action. The mapping of delays is straightforward and its formalisation is omitted here.

$$
\begin{array}{|l}
\mathit{mapInputActions} : \mathit{TRANSREL} \rightarrow \mathit{TIOTS\_ACTIONS} \\
\hline
\forall \mathit{transRel} : \mathit{TRANSREL} \bullet \mathit{mapInputActions}(\mathit{transRel}) = \\
\quad \bigcup\{\mathit{entry} : \mathit{transRel} \mid \mathit{entry}.2 \in \operatorname{ran} \mathit{del} \bullet \\
\qquad \{\mathit{stmt} : (\mathit{delayTransition}(\mathit{entry}.2)).2 \bullet (\mathit{mapStatement}(\mathit{stmt}))\}\}
\end{array}
$$

TIOTS* – *States The function $\mathit{mapState} : \mathit{STATE} \rightarrowtail \mathit{TIOTS\_STATE}$ maps a DFRS state to a TIOTS state, represented abstractly by a name; $\mathit{mapStatements} : \mathit{STMT\_SET} \rightarrow \mathit{TIOTS\_ACTIONS}$ maps DFRS statements to a set of TIOTS actions. Each action is a name to represent the corresponding statement.

TIOTS* – *Transitions From the transition relation of a DFRS we derive that of the TIOTS using the function *mapTransitionRelation*, formalised below. Figure 3 illustrates this mapping with an example.

$$
\begin{array}{|l}
\mathit{mapTransitionRelation} : \mathit{TRANSREL} \rightarrow \mathit{TIOTS\_TRANSREL} \\
\hline
\forall \mathit{transRel} : \mathit{TRANSREL} \bullet \\
\quad \exists \mathit{tr}1, \mathit{tr}2 : \mathit{TIOTS\_TRANSREL} \mid \\
\quad \mathit{tr}1 = \mathit{mapFunctionTransitions}(\mathit{getTransitions}(\mathit{transRel}, \operatorname{ran} \mathit{fun}), \\
\qquad \operatorname{ran} \mathit{mapState}) \land \\
\quad \mathit{tr}2 = \mathit{mapDelayTransitions}(\mathit{getTransitions}(\mathit{transRel}, \operatorname{ran} \mathit{del}), \\
\qquad \operatorname{ran} \mathit{mapState} \cup \mathit{getStates}(\mathit{tr}1)) \bullet \\
\quad \mathit{mapTransitionRelation}(\mathit{transRel}) = \mathit{tr}1 \cup \mathit{tr}2
\end{array}
$$

The function *getTransitions* is used to extract the transitions of the transition relation *transRel* of a particular type characterised by the range of the *fun* or *del* constructors. The translation functions *mapFunctionTransitions* and *mapDelayTransitions* for the different kinds of transitions take the sets of states already in use as an extra parameter,since, as illustrated, they create new states.
For illustration, we show the mapping of delay transitions.

$$
\begin{array}{|l}
\mathit{mapDelayTransitions} : (\mathit{TRANSREL} \times \mathit{TIOTS\_STATE\_SET}) \rightarrow\!\!\!\!\!+ \\
\quad \mathit{TIOTS\_TRANSREL} \\
\hline
\operatorname{dom} \mathit{mapDelayTransitions} = \\
\quad \{\mathit{tr} : \mathit{TRANSREL} \mid \forall \mathit{en} : \mathit{tr} \bullet \mathit{en}.2 \in \operatorname{ran} \mathit{del}\} \times \mathit{TIOTS\_STATE\_SET} \\
\forall \mathit{transRel} : \mathit{TRANSREL};\ \mathit{used} : \mathit{TIOTS\_STATE\_SET} \bullet \\
\ \exists \mathit{tr}1, \mathit{tr}2 : \mathit{TIOTS\_TRANSREL} \bullet \\
\ \ \mathit{tr}1 = \mathit{mapTDDelayTransitions}(\\
\qquad \bigcup\{\mathit{set} : \mathit{groupNTDDelayTrans}(\mathit{transRel}) \mid \#\mathit{set} = 1\}, \mathit{used}) \land \\
\ \ \mathit{tr}2 = \mathit{mapSetOfNTDDelayTransitions}(\\
\quad \{\mathit{set} : \mathit{groupNTDDelayTrans}(\mathit{transRel}) \mid \#\mathit{set} > 1\}, \mathit{used} \cup \mathit{getStates}(\mathit{tr}1)) \land \\
\ \ \mathit{mapDelayTransitions}(\mathit{transRel}, \mathit{used}) = \mathit{tr}1 \cup \mathit{tr}2
\end{array}
$$

The function mapDelayTransitions applies to sets of transitions whose entries *en* are of type *del* ($\mathit{en}.2 \in \operatorname{ran} \mathit{del}$). For those, the functions *mapTDDelayTransitions* and *mapSetOfNTDDelayTransitions* are used to map the deterministic and non-deterministic transitions. The function *groupNTDDelayTrans* defines a set of sets of transitions with the same delay. The sets *set* of size 1 contain the time deterministic transitions. The sets of size greater than one group the nondeterministic transitions. For each of these sets, a chain of transitions is defined. For the nondeterministic transitions, we ensure that the TIOTS target state obtained from the mapping of the transitions is the same. This becomes the initial state of the chain of input actions obtained from the statements of each delay transition.

Mapping a DFRS to a TIOTS We use the functions named above to define how a DFRS is mapped to a TIOTS. The set of states of a TIOTS is defined as the union of the states obtained from its transition relation with its initial state. Our mapping function is total, that is, every data-flow reactive system can be mapped to a corresponding timed input-output system.

Theorem 1 *Totality of fromDFRStoTIOTS*

$$\forall\, d : \mathit{DFRS} \bullet (\exists\, t : \mathit{TIOTS} \bullet t = \mathit{fromDFRStoTIOTS}\,(d))$$

Furthermore, the obtained TIOTS preserves the time additivity and null delay properties, and it is time deterministic. The proofs of these results are in [7].

4 Formalising Natural Language Requirements

A DFRS model can automatically generated from NL requirements described by actions guarded by conditions. Here, we provide an overview of how it is done. Pseudo-code of the related algorithms can be seen in [7]. For more details about the format of the requirements we refer to [8, 6, 9].

First, the requirements are parsed to assess whether they are correct with respect to a Controlled Natural Language (CNL) defined in [9]. For instance, the following is an example of a valid requirement of the VM: "*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*" [REQ001].

Afterwards, the syntax trees obtained from the requirements are automatically mapped into an informal semantic representation based on the Case Grammar theory [13]. In this theory, a sentence is analysed in terms of the semantic roles played by each word or group of words in the sentence (e.g., Agent — who performs the action; Patient — who is affected by the action, To Value — value associated with action, and so on). Table 1 shows a concrete example obtained from REQ001. More details of this step are reported in [9].

Finally, we employ an algorithm defined to generate a DFRS from a list of case frames (Algorithm 1). First, the algorithm calls *identifyVariables* to identify the system variables (line 1). A variable is classified as an input if, and only if, it appears only in *patient* roles of conditions; otherwise it is an output. To distinguish timers, we require their names to have "timer" as a suffix.

The type of the variables is inferred from the values mentioned in the *to value* role. Then, we create an initial state for these variables (line 2) considering initial default values (like 0 for *int* and *nat*, and *false* for *bool*, for instance).

Table 1. Example of Case Frames (Vending Machine).

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

Algorithm 1: Derive DFRS

```
input  : reqCFList
output : dfrs
1  $inputList, outputList, timerList = identifyVariables(reqCFList)$;
2  $initialState = buildInitialState(inputList, outputList, timerList)$;
3  $functionEntries = identifyFunctions(reqCFList, inputList, outputList, timerList)$;
4  $dfrs = new\ DFRS()$;
5  $dfrs.I = inputList$;
6  $dfrs.O = outputList$;
7  $dfrs.T = timerList$;
8  $dfrs.gcvar = setSystemGC(timerList)$;
9  $dfrs.s_0 = initialState$;
10 $dfrs.TR = generateTransition(functionEntries)$;
```

Afterwards, the algorithm calls *identifyFunctions* to identify the function transitions that describe the system behaviour (line 3). We identify one function for each different agent. Therefore, *identifyFunctions* yields a list of functions indexed by the corresponding agents. Each function is a list of action statements mapped to the respective discrete and timed guards. In the end (lines 4–9), the algorithm creates a DFRS. The complete definition can be seen in [7]. Here, we now present the algorithm for statement generation.

Algorithm 2 generates an action statement from a case frame that depicts an action. First (lines 1–3), we retrieve the verb from the Action and the name of the variable involved from the Patient. If the variable is a timer and the verb is not reset, an exception is raised since timers can only be reset (line 4–5).

The next step is the identification of the value being assigned to the variable (lines 6–10). If the verb is *"reset"* the value is the system global time (line 7). Otherwise, it is the content of the To Value (line 8). If the content of the To Value is not an integer, a float or a boolean, it is a string and the value is the index of this string within the list of possible values of the variable (lines 9–10).

If the verb being used describes a simple mathematical operation, the algorithm creates the corresponding expression considering the variable and values identified (lines 11-16). Then, a new statement is created considering the variable and values identified (lines 17–18).

5 Practical Validation: Test Cases from NL Requirements

To provide an empirical argument as to whether the DFRS model is expressive enough to represent the behaviour of a timed reactive system as defined using

Algorithm 2: Generate Statement

```
input  : $action, varList$
output : $actionStatement$
$verb = action.ACT$;
$varName = toString(action.PAT)$;
$var = varList.find(varName)$;
if $var.kind = timer \wedge \neg\ verb.equals(\text{“reset”})$ then
    $throw\ Exception$(“timers can only be reset”);
$value = null$;
if $verb.equals(\text{“reset”})$ then $value =$“gc”;
else $value = toString(action.TOV)$;
if $\neg\ isInteger(value) \wedge \neg\ isFloat(value) \wedge \neg\ isBoolean(value)$ then
    $value = var.possibleValuesList.getIndex(value)$;
$rhsExp = newExp()$;
if $verb.equals(\text{“add”})$ then $rhsExp = varName$ + “+” + $value$;
else if $verb.equals(\text{“subtract”})$ then $rhsExp = varName$ + “-” + $value$;
else if $verb.equals(\text{“multiply”})$ then $rhsExp = varName$ + “*” + $value$;
else if $verb.equals(\text{“divide”})$ then $rhsExp = varName$ + “/” + $value$;
else $rhsExp = value$;
$actionStatement = new\ Statement()$;
$actionStatement = varName$ + “:=” + $rhsExp$;
```

natural language, we consider the four examples listed in Section 1. Supported by a mechanisation of the strategy presented in Section 4, we assess whether test cases, either independently written by specialists of our industrial partner or generated by a commercial tool (RT-Tester[4]) from the same set of requirements, are compatible with the corresponding DFRS models.

To analyse the compatibility with the corresponding DFRS model, we implemented a depth-first search algorithm that explores the DFRS state space guided by a test case. We provide to the DFRS the inputs described by each test vector, and check whether the outputs provided by the system are equal to those in the vector. This comparison is straightforward since we are dealing with primitive types.

The selected tests are relevant as they are able to detect a high amount of errors introduced by mutation testing as reported in [6]. The verdict of our testing experiments have been successful as all considered test vectors are compatible with the corresponding DFRS models, which gives evidence that the generated DFRSs indeed capture the NL requirements as suggested in this paper.

6 Conclusions

We have presented DFRSs, a concise formal model to represent timed reactive systems. It is part of an automatic strategy to generate test cases from natural language requirements that may also describe temporal properties. We have given a semantics for DFRSs based on TIOTSs. This mapping preserves desired properties of a TIOTS, namely, time additivity, null delay, and time determinism. We have also considered examples from four different domains, and showed

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

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[5] that automatically generates DFRS models from NL requirements, besides other features such as animation of DFRS models.

Previous studies have already addressed the topic of formal modelling natural languages. These works differ in two main aspects: (1) structure of NL requirements, or (2) support for timed requirements.

Some studies opt for a more free structure, whereas other impose more restrictions when writing the requirements. In general, this choice is related to the trade-off of a greater or lesser level of automation. Works such as those reported in [4, 20] generate a formal model from unrestricted NL requirements. This makes the strategy more flexible than ours, but requires user interaction for the generation process, whereas our strategy is fully automated. Other studies [17] achieve a high level of automation by imposing restrictions that make the NL requirements resemble an algorithm. In our work, we reach a compromise.

Our NL imposes some restrictions, but the requirements still resemble a textual specification. Our restrictions make our approach suitable for describing actions guarded by conditions, and thus we cannot express properties like invariants; this can be accomplished by works such as [1].

A compromise similar to ours is reached, for instance, in [5], but timed requirements are not covered. In [17] timed requirements are considered, but as previously said from a not so natural textual representation. In [11] timed requirements are handled, but the strategy requires human intervention.

We intend to: (1) integrate this study with our previous works to take advantage of the generality of DFRS as indicated in Figure 1; (2) analyse the soundness of our DFRS encoding in CSP; (3) propose a conformance relation to DFRS models, and (4) compare it with typical conformance relations defined to TIOTSs, as well as with the conformance relation we define in [10].

Acknowledgments. This work was carried out with the support of the CNPq (Brazil), INES[6], and the grants: FACEPE 573964/2008-4, APQ-1037-1.03/08, CNPq 573964/2008-4 and 476821/2011-8.

References

1. Bajwa, I., Bordbar, B., Anastasakis, K., Lee, M.: On a chain of transformations for generating alloy from NL constraints. In: International Conference on Digital Information Management. pp. 93–98 (2012)
2. Blackburn, M., Busser, R., Fontaine, J.: Automatic Generation of Test Vectors for SCR-style Specifications. In: Annual Conference on Computer Assurance (1997)
3. Blackburn, M.R., Busser, R., Nauman, A.: Removing Requirement Defects and Automating Test. In: International Conference on Software Testing Analysis & Review (2001)

[5] The NAT2TEST tool can be obtained by contacting the authors.
[6] www.ines.org.br

4. Boddu, R., Guo, L., Mukhopadhyay, S., Cukic, B.: RETNA: from Requirements to Testing in a Natural Way. In: IEEE International Requirements Engineering Conference. pp. 262–271 (2004)
5. Brottier, E., Baudry, B., Le Traon, Y., Touzet, D., Nicolas, B.: Producing a Global Requirement Model from Multiple Requirement Specifications. In: Entreprise Distributed Object Computing Conference. pp. 390–401 (2007)
6. Carvalho, G., Barros, F., Lapschies, F., Schulze, U., Peleska, J.: Model-Based Testing from Controlled Natural Language Requirements. In: Artho, C., lveczky, P.C. (eds.) Formal Techniques for Safety-Critical Systems, Communications in Computer and Information Science, vol. 419, pp. 19–35. Springer International Publishing (2014)
7. Carvalho, G., Carvalho, A., Rocha, E., Cavalcanti, A., Sampaio, A.: Z Definition, Algorithms and Proofs for Data-Flow Reactive Systems. Tech. rep., UFPE (2014), `http://www.cin.ufpe.br/~ghpc/`
8. Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., Blackburn, M.: Test Case Generation from Natural Language Requirements based on SCR Specifications. In: Symposium on Applied Computing. vol. 2, pp. 1217–1222 (2013)
9. Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., Blackburn, M.: NAT2TEST$_{SCR}$: Test case generation from natural language requirements based on SCR specifications. Science of Computer Programming (2014)
10. Carvalho, G., Sampaio, A., Mota, A.: A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. In: Formal Methods and Software Engineering, LNCS, vol. 8144, pp. 148–164. Springer Berlin Heidelberg (2013)
11. Cavada, R., Cimatti, A., Mariotti, A., Mattarei, C., Micheli, A., Mover, S., Pensallorto, M., Roveri, M., Susi, A., Tonetta, S.: Supporting Requirements Validation: The EuRailCheck Tool. In: International Conference on Automated Software Engineering. pp. 665–667 (2009)
12. FAA: Requirements Engineering Management Findings Report. Tech. rep., U.S. Department of Transportation - Federal Aviation Administration (2009)
13. Fillmore, C.J.: The Case for Case. In: Bach, Harms (eds.) Universals in Linguistic Theory, pp. 1–88. New York: Holt, Rinehart, and Winston (1968)
14. Heninger, K., Parnas, D., Shore, J., Kallander, J.: Software Requirements for the A-7E Aircraft - TR 3876. Tech. rep., U.S. Naval Research Laboratory (1978)
15. ISO: Z formal specification notation (ISO/IEC 13568). Tech. rep., International Organization for Standardization (2002)
16. Larsen, K., Mikucionis, M., Nielsen, B.: Online Testing of Real-time Systems using Uppaal: Status and Future Work. In: Perspectives of Model-Based Testing - Dagstuhl Seminar. vol. 04371 (2004)
17. Li, J., Pu, G., Wang, Z., Chen, Y., Zhang, L., Qi, Y., Gu, B.: An Approach to Requirement Analysis for Periodic Control Systems. In: Annual IEEE Software Engineering Workshop. pp. 130–139 (2012)
18. Peleska, J., Vorobev, E., Lapschies, F., Zahlten, C.: Automated Model-Based Testing with RT-Tester. Tech. rep., Universität Bremen (2011)
19. Saaltink, M.: The Z/EVES System. In: International Conference of Z Users. pp. 72–85. ZUM '97, Springer-Verlag (1997)
20. Santiago Junior, V., Vijaykumar, N.L.: Generating Model-based Test Cases from Natural Language Requirements for Space Application Software. Software Quality Journal 20, 77–143 (2012)
21. Schmaltz, J., Tretmans, J.: On Conformance Testing for Timed Systems. In: International Conference on Formal Modelling and Analysis of Timed Systems, vol. 5215, pp. 250–264. Springer Berlin Heidelberg (2008)