Pós-Graduação em Ciência da Computação

Bruno Medeiros de Oliveira

SIMULATION OF HYBRID SYSTEMS FROM NATURAL LANGUAGE REQUIREMENTS

M.Sc. Dissertation

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE

2016

Bruno Medeiros de Oliveira

SIMULATION OF HYBRID SYSTEMS FROM NATURAL LANGUAGE REQUIREMENTS

M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: *Augusto Sampaio*
Co-Advisor: *Gustavo Carvalho*

RECIFE
2016

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

O48s Oliveira, Bruno Medeiros de
Simulation of hybrid systems from natural language requirements / Bruno Medeiros de Oliveira. – 2016.
93 f.: il., fig., tab.

Orientador: Augusto Sampaio.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.
Inclui referências e apêndices.

1. Engenharia de software. 2. Engenharia de requisitos. 3. Sistemas híbridos. I. Sampaio, Augusto (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2016-148

Bruno Medeiros de Oliveira

Simulation of Hybrid Systems from Natural Language Requirements

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

Aprovado em: 05/09/2016.

BANCA EXAMINADORA

Prof. Dr. Juliano Manabu Iyoda
Centro de Informática/UFPE

Prof. Dr. Eduardo Henrique da Silva Aranha
Departamento de Informática e Matemática Aplicada/UFRN

Prof. Dr.Augusto Sampaio
Centro de Informática/UFPE (**Orientador**)

I dedicate this thesis to my parents and professors who gave me the necessary support to get here.

Acknowledgements

I would like to thank Professor Augusto Sampaio, my advisor, for his guidance, and motivation and for providing the opportunity to work with him. He has been instrumental in bringing me to the field of Formal Methods and Testing and I am glad I got the chance to work with him.

I would like to thank Gustavo Carvalho for being on my thesis committee and providing invaluable advice and support in so many ways over past two years. Without his early coaching and enthusiasm none of this would have happened.

Finally, I would like to thank my parents who have been so close to me that I found them with me whenever I needed. It is their unconditional love that motivates me to set higher targets.

Far better is it to dare mighty things, to win glorious triumphs, even though checkered by failure... than to rank with those poor spirits who neither enjoy nor suffer much because they live in a gray twilight that knows not victory nor defeat.

—THEODORE ROOSEVELT

Abstract

Despite technological advances in the industry of systems development, testing is still the most commonly used verification method to ensure reliability. Model-based testing (MBT) techniques are principally employed for the purpose of generating test cases from specification models. Contributing to this branch of research an MBT strategy for creating test cases from controlled natural language (CNL) requirements was created, called NATural Language Requirements to TEST Cases (NAT2TEST). The NAT2TEST strategy deals with data-flow reactive systems, a class of embedded systems whose the main feature is to have the inputs and outputs always available as signals. However, there is a demand from the industry to to apply the strategy in the context of hybrid systems. These systems are a fusion of continuous dynamical and discrete dynamical systems, that is, they combine dynamical characteristics from both continuous and discrete worlds. Hybrid systems have received much attention in the last years. The main contribution of this work is to extend the NAT2TEST strategy to deal with hybrid systems. Using the new proposed approach, it is possible to write the requirements of a hybrid system, whose semantics is characterised based on the case grammar theory. Then, a formal representation of the system is built considering a model of hybrid data-flow reactive systems. Finally, to analyse the system behaviour via simulation, a modelling environment for simulation of hybrid systems was used, called Acumen. Thereby, a specification model in Acumen is generated and simulated in this environment. The characteristics of the new approach are exemplified using two examples, one belonging to the electronic field, the DC-DC Boost Converter (BC), and the other belonging to the automotive domain, the Adaptive Cruise Control (ACC).

Keywords: Hybrid systems. Model-based testing. Controlled natural language. Case grammar. Data-flow reactive system. Simulation

Resumo

Apesar dos avanços tecnológicos na indústria de desenvolvimento de sistemas, testes ainda é o método de verificação mais comumente usado para garantir confiabilidade. Técnicas de testes baseadas em modelo (MBT) são empregadas principalmente com a finalidade de geração de casos de testes a patir de modelos da especificação do sistema. Contribuindo para este ramo de pesquisa, foi criada uma estratégia MBT para a criação de casos de teste a partir de uma linguagem natural controlada (CNL), chamada de NAT2TEST. A estratégia NAT2TEST lida com sistemas reativos de fluxo de dados (DFRS), uma classe de sistemas embarcados cuja principal característica é a de terem as entradas e saídas sempre disponíveis como sinais. No entanto, há uma demanda oriunda da indústria para a utilização da estratégia no contexto de sistemas híbridos. Estes sistemas são uma fusão entre comportamentos dinâmicos e discretos, isto é, que combinam características dinâmicas de ambos os mundos, contínuo e discreto. Os sistemas híbridos têm recebido muita atenção nos últimos anos. A principal contribuição deste trabalho é estender a estratégia NAT2TEST para lidar com sistemas híbridos. Utilizando a abordagem proposta, é possível escrever os requisitos de um sistema híbrido, cuja semântica é caracterizada através da teoria de gramática de casos. Em seguida, uma representação formal do sistema é construída considerando um modelo DFRS para sistemas híbridos. Finalmente, para analisar o comportamento do sistema, por meio de simulação, um ambiente de modelagem e simulação de sistemas híbridos foi usado, chamado Acumen. Com isso, a estratégia proposta gera um modelo da especificação em Acumen e esse modelo é simulado no ambiente. As características da nova abordagem foram exemplificadas usando dois exemplos, um pertencente ao o campo eletrônico, o DC-DC Boost Converter (BC), e a outra pertencente ao domínio automobilístico, o Adaptive Cruise Control (ACC).

Palavras-chave: Sistemas híbridos. Testes baseados em modelos. Requisitos. Linguagem natural Controlada. Gramática de casos. Sistemas reativos de fluxo de dados. Simulação.

List of Figures

List of Tables

List of Acronyms

Contents

1

Introduction

The competitiveness of the market together with the need for software reliability led to the creation of automatic techniques as alternatives to the traditional approach of manual testing, especially, because the most common verification method in the industry is still testing. In this context, model-based testing was devised (MODEL-BASED TESTING IN PRACTICE, 1999).

This strategy involves developing and using a specification model to generate tests. This model encodes the intended behaviour of an implementation, known as, System Under Test (SUT). Test generation can be especially effective for systems that are vulnerable to changes because it suffices to modify the specification model and then rapidly regenerate an updated test suite (MODEL-BASED TESTING FOR THE SECOND GENERATION OF INTEGRATED MODULAR AVIONICS, 2011). Examples of systems testing using MBT include avionics (MODEL-BASED TESTING FOR THE SECOND GENERATION OF INTEGRATED MODULAR AVIONICS, 2011), automotive (ZANDER-NOWICKA, 2008), control, medical, military, and manufacturing systems (MODEL BASED TESTING USING SOFTWARE ARCHITECTURE, 2010).

It is important to consider that the model can be written in several modelling languages and, thus, different techniques can be adopted to generate test cases. For instance, one might derive execution traces from the specification model, and then use these traces to generate test cases for the SUT: a sequence of input and expected output actions (ZANDER; SCHIEFERDECKER; MOSTERMAN, 2011). The NATural Language Requirements to TEST Cases (NAT2TEST) strategy, for example, take as input textual requirements and translates them into an internal representation model named Data-Flow Reactive System (DFRS). This model is translated to a target formalism, like, for example, Software Cost Reduction (SCR) (AUTOMATIC GENERATION OF TEST VECTORS FOR SCR-STYLE SPECIFICATIONS, 1997), Intermediate Model Representation (IMR) (AUTOMATED TESTING WITH RT-TESTER - THEORETICAL ISSUES DRIVEN BY PRACTICAL NEEDS, 2000), Petri nets (MURATA, 1989) or the CSP process algebra (BROOKES; HOARE; ROSCOE, 1984), in order to generate test cases CARVALHO (2016).

According to UTTING; PRETSCHNER; LEGEARD (2012), Model-Based Testing (MBT) can be described by the process shown in Figure 1.1. Using the system requirements or even the system specification documents the specification model is constructed. During the model creation, its level of abstraction is related to the purpose of testing, because sometimes this model is called the test model. Having a more abstract model of the real system implies that the model is potentially easier to check, modify and maintain than the SUT. This abstraction is useful when validating the model, otherwise, verifying the model would be so costly as to validate the SUT. However, it is desirable that the model is accurate enough to generate concrete test cases, with actions, input parameters and expected results, as well as other information required to run the generated tests. Due to the commonly large number of test cases that can be generated from a specification model, selection criteria are necessary to guide the generation process. A test script is an executable code responsible for performing a test case, abstracts the output of the SUT, and then produces the test verdict. Usually, the environment is capable of adapting the abstract test data to the concrete SUT interface.

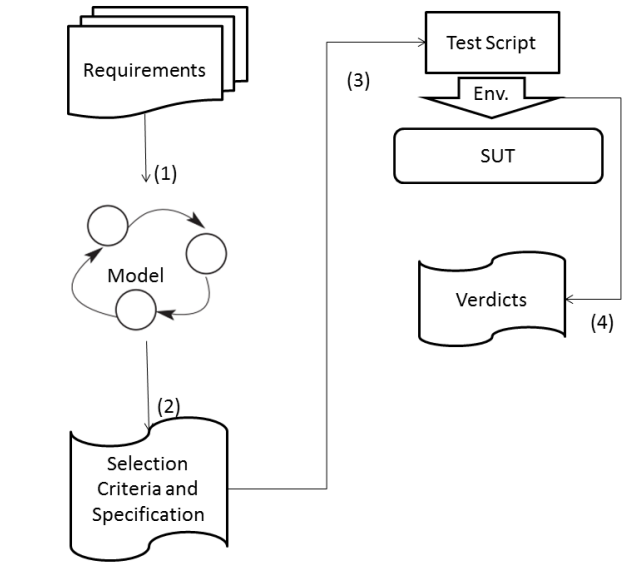

Figure 1.1: An overview of the model-based testing process

Despite all the benefits that MBT provides, there are also some negative points for their adoption. Among them, these models are not always available at the beginning of the project, and there may be some resistance to create them due to unfamiliarity with the associated syntax and semantics. As an alternative to solve this problem, CARVALHO (2016) proposes a strategy called NAT2TEST that can use Natural Language Processing (NLP) techniques to obtain the required models from natural-language specifications.

NAT2TEST is an entirely automatic strategy for test case generation from natural language requirements. The approach focuses on reactive systems (CARVALHO, 2016). The tests are generated from a Data-Flow Reactive System (DFRS): a class of embedded systems

whose inputs and outputs are always available as signals. Figure 1.2, presented originally by CARVALHO (2016), shows the phases of this approach. The three initial phases are fixed: (1) syntactic analysis, (2) semantic analysis, and (3) DFRS generation; the other phases are related to the chosen formalism to generate test cases, for instance, SCR (HEITMEYER; BHARADWAJ, 2000), IMR (PELESKA; VOROBEV; LAPSCHIES; ZAHLTEN, 2011), Communicating Sequential Processes (CSP) (CARVALHO, 2016), among others.

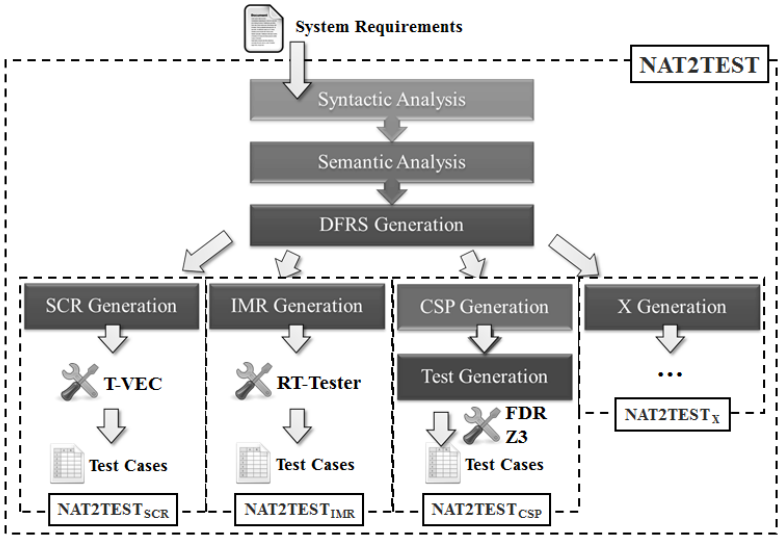

Figure 1.2: Phases of the NAT2TEST strategy

However, the strategy does not support hybrid systems, which is a class of systems that are widely used by the industry and is gaining popularity in the scientific community. In reality, for several years great effort has been devoted to the study of testing hybrid systems in particular, within the context of MBT MODEL-BASED TESTING AND MONITORING FOR HYBRID EMBEDDED SYSTEMS (2004).

Without even realising it, hybrid systems cross our way several times a day: an automatic teller machine, a car's anti-lock braking system, a video-recorder and a washing machine are examples thereof (HYBRID DYNAMICAL SYSTEMS, 1989). In general, hybrid systems are those that consist of "a logical discrete-event decision-making controller system interacting with a continuous-time process" (SAVKIN; EVANS, 1998). This kind of system has attracted considerable attention in recent years. Along with their importance the need for reliability also arises (LARSEN; STEFFEN; WEISE, 1997).

1.1 Research question and contributions

Taking into account all the discussion presented so far, the central research question of this work is: how to extend the NAT2TEST strategy to deal with hybrid systems?

The proposed extension NATural Language Requirements to TEST Cases for hybrid systems (h-NAT2TEST) must be conservative, so that the NAT2TEST strategy shall still be

employed for dealing with non-hybrid systems. As illustrated in Figure 1.3, the extension proposed here impacts all fixed stages of the original NAT2TEST strategy.

Our first concern is to extend the Controlled Natural Language to capture requirements of dynamic systems, particularly, differential equations. An extended version of DFRS is proposed as an internal formal model to represent such requirements.

Unlike the original NAT2TEST strategy, however, the main purpose here is to generate a target model with the purpose of simulation. To achieve this, a DFRS is translated to an Acumen TAHA; DURACZ; ZENG; ATKINSON et al. (2015) model and the environment presented in TAHA (2012) is used for performing simulation.

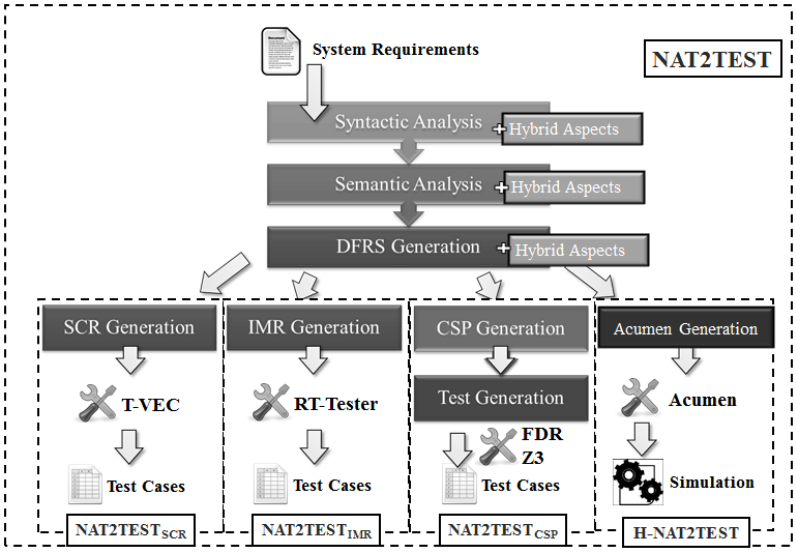

Figure 1.3: Phases of the hybrid NAT2TEST strategy

In summary, this work presents the following contributions:

- An extension of the Controlled Natural Language to allow expressing requirements of dynamic systems;
- An extension of the DFRS model to allow the internal and formal representation of such requirements;
- a systematic translation from requirements to the extended notion of DFRS;
- a translation from DFRS to Acumen;
- use of an Acumen environment to perform simulation; and
- two examples to illustrate the overall approach, from requirements definition to simulation.

Test case generation for hybrid systems is out of the scope of this work; it is one of the suggested topics for future work.

1.2 Thesis structure

This thesis is organised as follow:

Chapter 2 Discusses foundation concepts that are used in this research. First, the original NAT2TEST strategy is briefly presented. Afterwards, a definition of hybrid systems is presented, along with how they are modelled. Finally, is shown a brief discussion of simulation and how Acumen project fits in this context.

Chapter 3 Explains how the NAT2TEST strategy was extended, showing the parts that have been affected and the resulting impact. In addition, a running example is presented to illustrate the contributions of this work. Finally, it presents the formalism (Acumen) and the tool chosen to support the proposed extension and how models are simulated in this tool.

Chapter 4 Describes in details two case studies to illustrate the application of the h-NAT2TEST strategy. The first one is a DC-DC Boost Converter (BC), the running example, and the second is an Adaptive Cruise Control (ACC).

Chapter 5 Presents related work, our conclusions and discusses future work.

2

Problem background

This chapter introduces some of the main basic concepts that served as the foundation for this research. Particularly, the components of the original NAT2TEST strategy are presented, as well as how they are linked to generate test cases from natural language requirements. Soon after, it is introduced the concept of hybrid systems and how to represent them.

2.1 The NAT2TEST strategy

NAT2TEST is an entirely automatic strategy for test case generation from natural language requirements. This strategy aims at reactive systems, whose behaviour can be described through actions that should be taken when certain conditions are met. An important feature of the approach is the possibility of generating test cases for systems with discrete or continuous temporal properties (CARVALHO, 2016).

The rest of this section is devoted to a brief description of the three first phases of the NAT2TEST strategy. These are the phases that are adapted when dealing with hybrid systems and, thus, this explanation is required to understand the extension proposed in this work. For more comprehensive explanation of the original NAT2TEST strategy, we refer to CARVALHO (2016).

2.1.1 Syntactic and semantic analysis

For an automatic processing of requirements and generation of test cases, the requirements must be written according to a specific grammar, namely the SysReq-CNL. The System Requirements Controlled Natural Language (SysReq-CNL) is a Controlled Natural Language (CNL) grammar that consists of a subset of the English language. It has been created in order to turn the writing of system specifications more standardized and reliable, facilitating the conversion to a formal notation. In addition, this grammar was designed to handle requirements of data-flow reactive systems. These systems are part of a class of embedded systems where inputs and outputs are always available as signals.

The lexicon entries are classified into lexical categories to simplify the grammar, for example, it uses determiners, nouns, adjectives and so on. The complete grammar expressed in the Extended Backus-Naur Form (EBNF) notation, is shown in the Table 2.1. An important feature to highlight is that the lexicon is domain dependent which means that the instantiation of the categories must be manually created considering the current system domain. Nevertheless, a small set of a lexicon is initialised by default.

The grammar start symbol is *Requirement*, which consists of a *ConditionalClause* and an *ActionClause*. Therefore, the requirements have the form of action statements guarded by conditions. A *ConditionalClause* begins with a conjunction, and then its structure is similar to a Conjunctive Normal Form (CNF) – conjunction of disjunctions. The conjunctions are delimited by a *COMMA* and the *AND* keyword, whereas the disjunctions are delimited by the *OR* keyword. The elementary condition (*Condition*) comprises a *NounPhrase* (one or more nouns eventually preceded by a determiner and adjectives) and a *VerbPhraseCondition*, which begins with a *VerbCondition* (the verb "to be" or any other in the present or past tense). A *VerbCondition* is followed by an optional *NOT*, which negates the meaning of the next term, an optional ComparativeTerm and a *VerbComplement*.

An *ActionClause* begins with a *NounPhrase* followed by a *VerbPhraseAction*, which is rewritten as *SHALL* followed by at least one *VerbAction* and one *VerbComplement*. If more than one *VerbAction* and *VerbComplement* is used, then it is necessary to add a *COLON* after the *SHALL* keyword and use the *COMMA* to delimit the elements. A *VerbComplement* is an optional *VariableState* (a *NounPhrase*, an adjective, an adverb or a number) followed by zero or more *PrepositionalPhrase*, which consists of a preposition and a *VariableState*.

Although imposing writing structure, this grammar is general enough to allow the user to write sentences in several application domains. A simple example of how a requirement may be written, in SysReq-CNL, is shown below.

- *When the input1 becomes greater than or equal to 10, the System shall assign valid to the output1.*

The natural way of describing the system behaviour, imposed by the SysReq-CNL, facilitates the usability because the user does not need in-depth learning of a particular technology to describe a system entirely.

Table 2.1: SysReq-CNL – a grammar for system requirements

Requirement	→	ConditionalClause COMMA ActionClause PERIOD;
ConditionalClause	→	CONJ AndCondition;
AndCondition	→	AndCondition COMMA AND OrCondition
		| OrCondition;
OrCondition	→	OrCondition OR Condition | Condition;
Condition	→	NounPhrase VerbPhraseCondition;

ActionClause	→	NounPhrase VerbPhraseAction;
VerbPhraseAction	→	SHALL (VerbAction VerbComplement
		| COLON VerbAction VerbComplement
		(COMMA VerbAction VerbComplement)+);
VerbAction	→	VBASE;
VerbPhraseCondition	→	VerbCondition NOT? ComparativeTerm?
		VerbComplement;
VerbCondition	→	VTOBE_PRE3 | VPRE3RD
		| VTOBE_PRE | VTOBE_PAST3;
ComparativeTerm	→	(COMP (OR NOT? COMP)?);
VerbComplement	→	VariableState? PrepositionalPhrase*;
VariableState	→	(NounPhrase|ADV | ADJ | NUMBER);
PrepositionalPhrase	→	PREP VariableState;
NounPhrase	→	DETER? ADJ* Noun+;
Noun	→	NSING | NPLUR;

For the semantic analysis, a syntax tree is required, which is generated for each valid requirement during the syntactic analysis. The NAT2TEST strategy uses Thematic Role (TR) to give meaning for each sentence: each word or group of words has its role and functionality. In its notation, the TRs associated with a particular verb are grouped in a struct called Case Frame (CF).

Since the requirements are interpreted as actions that take place under certain conditions, the TRs and the CFs can also be classified into condition or action types. There are nine TRs grouped by type, which are:

TR's associated with conditions:

- Condition Patient (CPT): the entity related to each condition;
- Condition Action (CAC): the action that concerns each condition;
- Condition Modifier (CMD): a modifier related to the condition.
- Condition From Value (CFV): the CPT previous value;
- Condition To Value (CTV): the value satisfying the condition;

TRs associated with actions:

- Agent (AGT): entity who executes the action;
- Action (ACT): the action to be executed if the conditions are met;
- To Value (TOV): the Patient value after action completion.

- Patient (PAT): entity who is affected by the action

A requirement can generate several CFs: one for each verb. They are grouped in a structure called Requirement Frame (RF), in other words, a requirement gains full meaning through the interpretation of a RF.

For example, the thematic roles for the requirement described above are assigned as follows: TR's associated with conditions:

- Condition Patient (CPT): the input1;
- Condition Action (CAC): becomes;
- Condition Modifier (CMD): greater than or equal to;
- Condition From Value (CFV): -;
- Condition To Value (CTV): 10.

TRs associated with actions:

- Agent (AGT): the System;
- Action (ACT): assign;
- To Value (TOV): valid;
- Patient (PAT): the output1.

2.1.2 Generation of data-flow reactive systems

After creating all the case frames, the strategy has an informal, but structured, meaning of each system requirement, and the frames are used as input to the third phase. In this step, a formal representation of the system behavior is built: DFRS. This formal representation is a symbolic, timed and state-rich automata-based notation for representing natural-language requirements.

A DFRS has two different representations: a Symbolic Data-Flow Reactive System (s-DFRS) and an Expanded Data-Flow Reactive System (e-DFRS) one. The s-DFRS is a more abstract representation that avoids representing possible infinite sets, thus avoiding the state explosion problem. The e-DFRS representation is dynamically built from the symbolic model, and is used to check properties such as reachability, determinism, and completeness (CARVALHO; CAVALCANTI; SAMPAIO, 2016). In the present work we use s-DFRS to represent the requirements of hybrid systems.

The DFRS is intended to model an embedded system. To help to model temporal conditionals the DFRS can have timers in its definition. The Symbolic Data-Flow Reactive

System (s-DFRS) is formalised as a 6-tuple: $(I, O, T, gcvar, s0, F)$. Inputs (I) and outputs (O) are system variables, timers (T) are a special kind of variable whose values are non-negative numbers representing a discrete or a dense (continuous) time. The system global clock (gcvar) has the same type as the timers. The initial state is s0, and F is a set of functions.

The construction of the s-DFRS follows some steps, primarily, from the RFs it is inferred the system variables (inputs, outputs, and timers), as well as the functions F. Afterwards, these two pieces of information are compiled to instantiate the model. The definition and construction of the e-DFRS will not be discussed, given that it is not used in the NAT2TEST extension proposed in this work.

For example, the DFRS for the only requirement described above is formed by “the input1” in the set of input, “the output1” in the set of output, the set of Timers is empty, it has a global clock, also an initial state with initial values of the variables, and a function with the “input1 >= 10” representing a conditional, and “the output := valid” as statement of the conditional.

After deriving DFRS models from the RFs, in the original NAT2TEST strategy, an intermediate formal notation is considered to generate test cases. For instance, an s-DFRS model can be encoded as CSP processes, and then test cases are generated via refinement checking CARVALHO (2016). These phases are not further described since they are specific to the original approach, and, in the context of hybrid systems, our focus is on simulation, rather than on test case generation, as already emphasised.

2.2 Hybrid systems

Hybrid systems are an integral part of modern society. Numerous applications are all around us: rockets; autonomous auto-mobile systems; medical monitoring; process control systems; automatic pilot avionics, among others. Actually, hybrid systems is a generic term used to describe networks of interacting digital and analogue devices. Cyber-physical systems, control systems and embedded systems are, for example, relevant fields that share the concepts of hybrid systems (BRANICKY, 2005).

2.2.1 Definition of hybrid systems

It is a challenge to establish a unique definition for hybrid systems, since their investigation occurs on such a large variety of study areas, which have many concepts in common, even though the overall area of hybrid systems has not been fully consolidated (BRANICKY, 2005).

There are different perspectives of studying hybrid systems, e.g., the computing industry considers the context of a digital system interacting with an analogue environment (also known as *embedded systems*), where the key points are the analysis and verification of systems with discrete and dynamics events (SCHAFT, 2000).

Analysing physical systems, it was found that systems can usually operate in different modes, and changing from one mode to another sometimes can be described as an instantaneous discrete transition. From this context, the perspective of the modelling and simulation emerged. Another perspective involves *control systems*, where hierarchical systems have a discrete decision layer and a continuous implementation layer, e.g. supervisory control and multi-agent control (SCHAFT, 2000).

Therefore, in general, we can say that hybrid systems are a combination of discrete and continuous events. These events coexist, interact and change in response to dynamics as described by differential or difference equations in time (NICOLLIN; OLIVERO; SIFAKIS; YOVINE, 1993). These functions responsible for describing the behaviour of the variables are called *activities* (LARSEN; STEFFEN; WEISE, 1997).

One way to define the behaviour of hybrid systems is via the set of all possible trajectories of the continuous and discrete variables associated with the system. In this context, a hybrid systems is represented as a *hybrid automaton model* (FAHRENBERG; LARSEN; LEGAY, 2013).

To illustrate a hybrid system, we consider the thermostat described in (LUNZE; LAMNABHI-LAGARRIGUE, 2009). A thermostat is a device to regulate the temperature in a room. The heating system is supposed to work at its maximum power or completely turned off. This is a system that operates in two modes, "on" or "off". In each operation mode, the evolution of the temperature T can be expressed by a different differential equation. Figure 2.1, presented in (LUNZE; LAMNABHI-LAGARRIGUE, 2009), shows the modes and the differential equations of

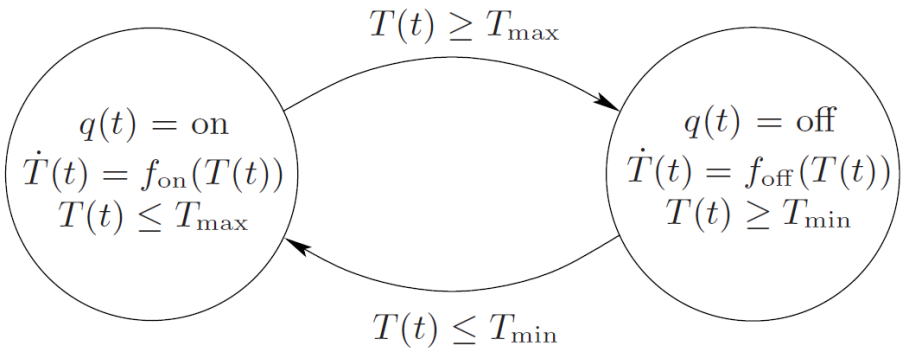

Figure 2.1: Thermostat model – an example of a hybrid system

the system.

Each mode corresponds to a node of a directed graph, while the edges indicate the possible discrete state transitions. The hybrid system in this example consists of two discrete states, $q \in \{on, off\}$, and a continuous state T, where T represents the temperature (real number). Regarding the behaviour of the system, we can say that the system has two distinct continuous behaviours that establish the evolution of the temperature T. One when in mode "on", ruled by the dynamics $\dot{T}(t) = f_{on}(T(t))$, which describe temperature lowering, and another when in the mode "off", governed by the dynamics $\dot{T}(t) = f_{off}(T(t))$, which governs the temperature increase.

The continuous state T and different conditions on T are responsible for changes of the discrete state *q*, since they may trigger discrete transitions. In Figure 2.1 it is possible to see that if the discrete state *q* is "on", and T is greater than or equal to T_{max}, the discrete transition from "on" to "off" becomes enabled. Differently, in state "off", the discrete transition is enabled if T is less than or equal to T_{min}. In addition, each discrete state also has an invariant, and the process may only stay within a state as long as it does not violate the *invariant* and when the transition is enabled it is executed instantaneously, without time-consuming (LARSEN; STEFFEN; WEISE, 1997).

The thermostat behaviour is shown graphically in Figure 2.2. It is possible to see that the temperature does not suffer from discontinuities while the state q changes discontinuously. Based on this example, we note that hybrid systems are two-phase systems as depicted in Figure 2.3, reproduced from LARSEN; STEFFEN; WEISE (1997). A continuous phase, where arbitrary continuous variables, including the clocks, evolve with time, and a discrete phase, in which one or more operations happen simultaneously with their corresponding state changes, but where no time passes. Thus, for each discrete state, it is necessary to define the behaviour of the continuous variables and, as discussed above, the most commonly used way is through differential equations as usual in physics. In what follows, we show how equations can be used as a basis for describing the system behaviour, in particular, of hybrid systems.

A system is a collection of parts that interact with each other and with its environment

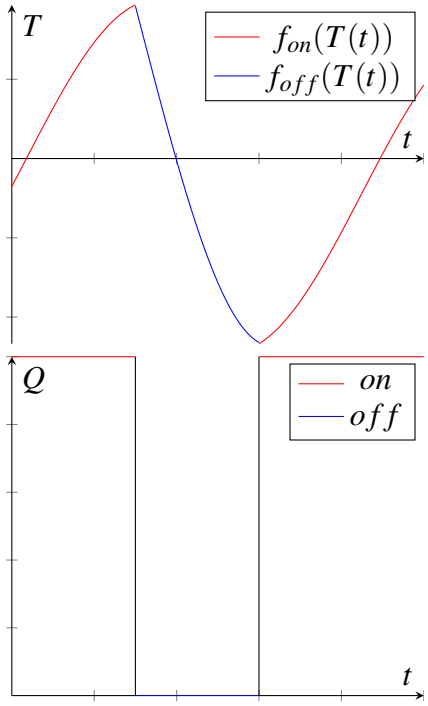

Figure 2.2: Example of the behaviour of a thermostat

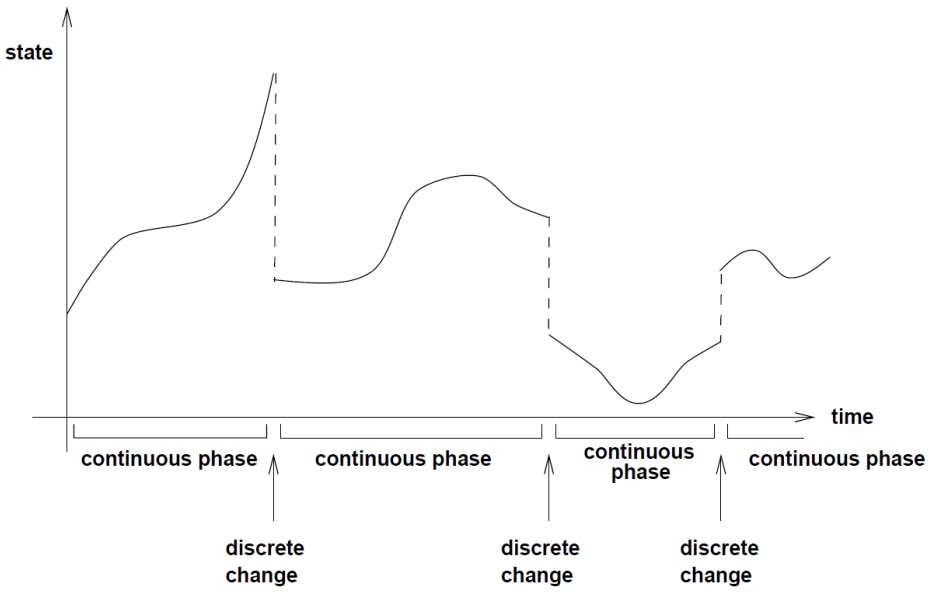

Figure 2.3: Two-phase system

through a set of input variables *u* and output variables *y*. In a continuous time system the time t is represented by $t \in \mathbb{R}$, whereas in a discrete time system the time t is represented by $t \in \mathbb{Z}$. Regularly, the symbol k is used instead of t to denote discrete time indices. A typical system that uses discrete-time is the computer, whereas physical systems use continuous time (HUBBARD; WEST, 1993).

A system can be classified as static or dynamic. The system is static if its output depends only on its present input. In others words, there is a function $f(u,t)$ to determine the output at any time t, $t \in T$ using only the input u like in equation 2.1.

$$y(t) = f(u(t),t) \tag{2.1}$$

Figure 2.4, originally presented in (CHEN, 2013), shows an example of static systems, which is a resistive circuit excited by an input voltage $u(t)$. Let the output be the voltage across the resistance R_3, according to the circuit theory, the output can be simply determined by the present input.

On the other hand, a dynamic system requires information on previously received input to determine the system output. i.e. to determine $y(t)$ one needs to know $u(\tau)$, $\tau \in (-\infty,t]$. Figure 2.5, originally presented (Y.LI, 2012), shows an example of a dynamic time invariant system: the flow control valve. The fluid pressure P is constant. A is orifice area and ρ is fluid density. However, the flow rate history is a function of the force $F(t)$ acting on the valve. It is necessary to know the time history of the forcing function $F(t)$ in order to determine the flow rate at any time. The position $x(t)$ of the valve is governed by the following differential equation (Y.LI, 2012):

$$\ddot{x} = F(t) - b\dot{x} - kx \tag{2.2}$$

Where k is the spring constant and b is the damping factor. For a circular pipe of radius R, the flow rate is then given by the following equation (Y.LI, 2012):

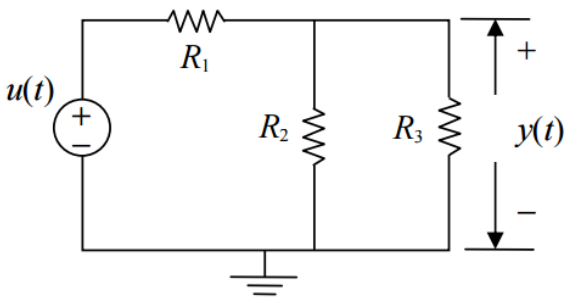

Figure 2.4: A resistive circuit

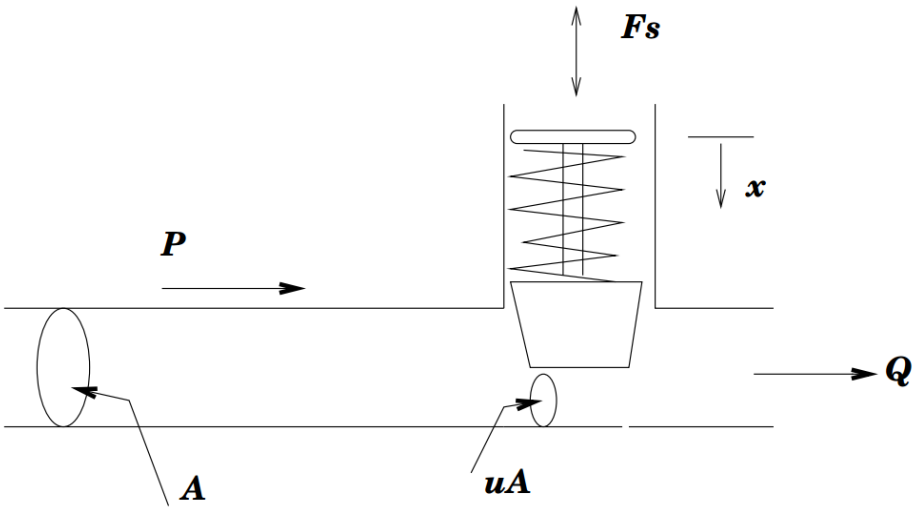

Figure 2.5: Flow control valve

$$Q = \frac{x^2}{R^2} A \sqrt{\frac{2P(t)}{\rho}} \tag{2.3}$$

This type of representation, shown in equations 2.2 and 2.3, is commonly used for system description in research areas such as electrical and mechanical engineering. This is called state-space representation (KHALI, 2002; SKOGESTAD; POSTLETHWAITE, 2005). In representation, $\dot{x}$ means the first derivative with respect to time, $\ddot{x}$ means the second derivative with respect to time, and so on.

Three essential elements are the basis of the state-space representation: a vector x of *state variables*, a vector u of *input variables*, and a vector y of *output variables*. All of them are explicit functions of time, and this means that their values depend on the time at which they are evaluated. In general, the values of x, u and y as a function of time are expressed as $x(t)$, $u(t)$, and $y(t)$, respectively. The temporal domain of the state-space representation is continuous or discrete, and the relationship among x and u is usually governed by differential or difference equations, respectively. For instance, in the following scheme:

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t), t) \\ x(k+1) &= f(x(k), u(k), k) \end{aligned} \tag{2.4}$$

where $t \in \mathbb{R}$ and $k \in \mathbb{Z}$. Using the equation 2.1 as a basis algebraic relation among y and the state and input variables, incrementally, it is possible to give an initial condition $x(0)$. Thus, to define a function of the inputs, $u(t)$, is required, then, all functions x that are solutions to Equation 2.4 denote the possible system behaviours.

2.2.2 Hybrid system representation

Once the key concepts to understand a representation of a hybrid system has been described, now we present how a hybrid system can be represented as a hybrid automaton model. ALUR; COURCOUBETIS; HALBWACHS; HENZINGER; HO; NICOLLIN; OLIVERO; SIFAKIS; YOVINE (1995) define a the hybrid automaton model as a finite automaton equipped with a set of continuous variables. Formally, it is a seven-tuple $HA = \langle X, Q, \psi, Inv, A, Ev, As \rangle$, where:

- X is a finite set of real-valued variables $\{x_i\}$. It is denoted by x, which is a vector of variables. The values of all variables at a given moment define a state of the variables. The set of all possible states is denoted by V.
- Q is a finite set of vertices called locations.
- ψ is a function that assigns to each location $l \in Q$ a function ψ_l describing the evolution of variables in time.

$$\psi_l(x, f) \rightarrow \dot{x}(f) = f(x, f). \tag{2.5}$$

- Inv is a function that assigns to each location $l \in Q$ a predicate Inv_l, called the invariant of l.
- A is a finite set of transitions. Each transition $a = (l, l')$ joins a source location $l \in Q$ to a target location $l' \in Q$.
- Ev is a function that assigns to each transition $a = (l, l')$, a predicate Ev_a called guard. The transition $a = (l, l')$ may be fired if the guard Ev_a is satisfied.
- As is a function that assigns to each transition $a = (l, l')$ a relation As_a called assignment. It is used to model the discrete changing of the values of variables.

$$As_a \rightarrow x = g(x). \tag{2.6}$$

A state of a hybrid automaton is represented by the pair (l, x), where $l \in Q$ and $x \in V$. In each position (vertice) of the automaton, the values of the variables change continuously with time according to the associated evolution function (an element of ψ). Each transition (edge) of the automaton is guarded by a condition, and its execution changes the values of the variables according to the associated assignment. Each location is also labelled with a condition, called invariant, which must hold while the system remains in the vertice.

The fully automatic analysis of hybrid systems described as hybrid automata is not feasible due to the complexity inherent to these systems. Therefore, subclasses of hybrid automata are studied. We can highlight the following subclasses described in (LARSEN; STEFFEN; WEISE, 1997):

Timed automata are the special case of hybrid systems where all *activities* have a growing and constant behaviour, invariants and pre-conditions are comparisons of clocks (and clock differences), and post-conditions are restricted to clocks reset.

Drifting-clock timed automata are a similar subclass of timed automata, where all *activities* may vary the behaviour within a given interval.

Linear hybrid systems is a subclass which can be analysed effectively and automatically by techniques shown in (ALUR; COURCOUBETIS; HENZINGER; HO, 1993), unlike the nonlinear. In these systems, invariants, guards and activities may only depend linearly on time. It is precisely this hybrid system subclass that this work considers.

2.2.3 DC-DC Boost Converter

To exemplify the use of the proposed strategy, we consider here a DC-DC Boost Converter (BC) as a running example. This example is the same one presented in (AERTS; MOUSAVI; RENIERS, 2015). This example is used in Chapter 3 to emphasize the changes with respect to the original NAT2TEST strategy, and is developed in full in Section 4.1.

The BC is a power converter that rises voltage while stepping down current, from its input to its output. This type of device is old in the electrical field but widely used in modern equipment. For example, the engines used in driving electric vehicles require much higher voltages, than could be provided via a single battery. Even if it were possible to use a single battery, its weight and size make its use impractical. The solution is to use fewer batteries and to boost, using a boost converter, the available DC voltage to the appropriate voltage.

The Figure 2.6, originally presented in (AERTS; MOUSAVI; RENIERS, 2015),illustrates a example physical of the BC and its basic circuit. A generic BC is composed of:

An inductor *(L)* is a passive electrical device that stores energy in the form of magnetic field, usually combining the effect of various loops of electric current;

A capacitor *(C)* is a passive two-terminal electrical component that stores electrical charges in an electric field, accumulating an internal imbalance of electric charge;

A switch *(S)* is a physical mechanism that rapidly switches a device on and off.

A diode *(D)* is a component that allows an electric current to pass in one direction (it has low resistance in one direction) while blocking current in the opposite direction (it has high resistance in the other direction).

A resistive load *(R)* is the output.

Closing the *(S)* generates a short circuit from the right side of L to the negative input. Consequently, a current flows between the positive and negative supply terminals through L,

which stores energy in its magnetic field. Initially, there is no current running in the right side of the circuit because of the combination of D, C and the resistive load represent a much higher impedance than the path straight through the *(S)*.

Opening the *(S)* causes a sudden drop in current and produce an electromotive force (emf) in the opposite polarity to the voltage across L (V_L) during the open period. The resulting current through D loads up C to $V_{IN} + V_L$ minus the forward voltage loss across D, and also supplies the load.

After the first start, the switch is closed, consequentely, the output of the circuit is isolated from the input, despite the load continues to be fulfilled with $V_{IN} + V_L$ from the charge on C. C is recharged each time the switch is open, so maintaining an almost steady output voltage across the load.

Summarising, the boost of the DC voltage is a consequence the combined physical properties of the inductor *L* and capacitor *C*, which are controlled by the switch *S* and diode *D*. This process transforms the input voltage *E* to an increased output voltage that is applied to the resistive load *R*. Note that the control elements of the boost converter transform the otherwise continuous system into a hybrid system. Finally, the system is made input dependent by tuning the resistive load *R* which results an internal stabilizing behaviour of the boost converter. In Figure 2.7, originally presented (AERTS; MOUSAVI; RENIERS, 2015), this system is modelled as a hybrid automaton.

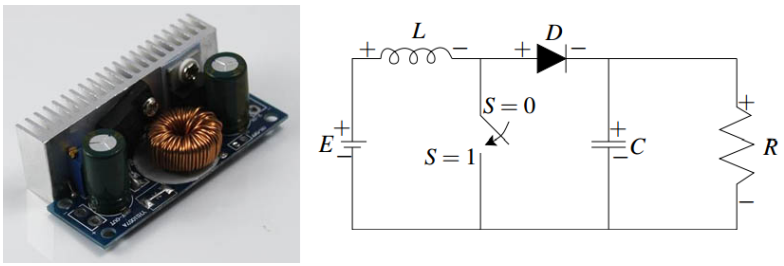

Figure 2.6: Boost converter

The four discrete states of the system are solely dependent on the position of the switch *S* and the mode of the diode *D* (conducting/blocking). In addition, the physical properties of the system are modelled by the electric charge *q* of the capacitor and the magnetic flux Φ of the inductor.

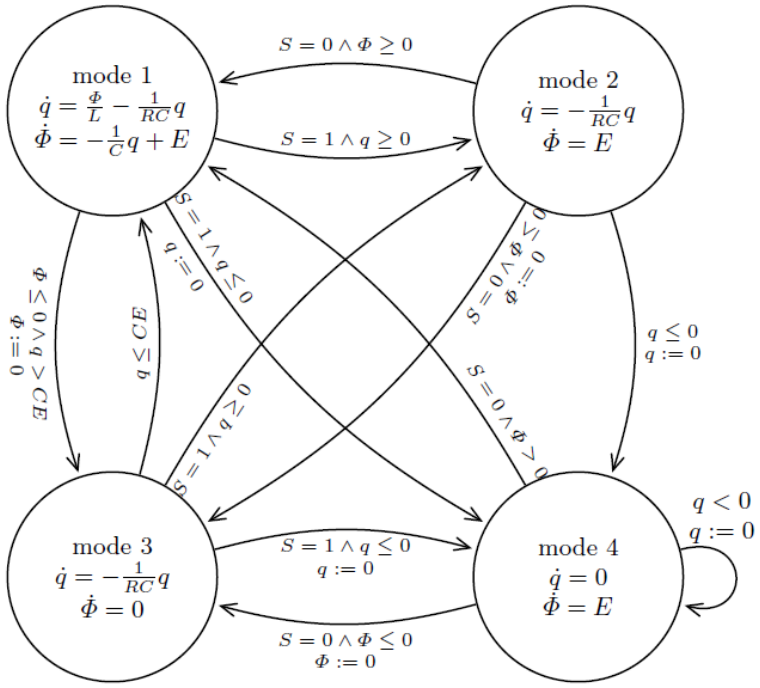

Figure 2.7: Boost converter automata

2.3 Simulation

Advances in technology continually lead to the construction of systems with higher complexity; therefore any change in these systems is made more complicated. A simulation is an approach to understanding and to evaluate the behaviour of these systems. More formally, simulation is “the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behaviour of the system of or evaluating various strategies (within limits imposed by a criterion or a set of criteria) for the operation of the system” (SHANNON, 1975).

Simulation has several advantages, for example, it is used to compress a time frame, a simulation running on a computer describes more quickly the effects of a change in a real word circumstances. It used in engineering design to verify the effects of changes on the product without producing a physical prototype. It is especially valuable for testing conditions that might be difficult to reproduce with simple prototypes, mainly in the early phase of the design process when the system may not be available. Also, it can increase the quality of the systems, potentially decreasing the number of errors found later in the design process (FRAMEWORK FOR SIMULATION OF HYBRID SYSTEMS: INTEROPERATION OF DISCRETE EVENT AND CONTINUOUS SIMULATORS USING HLA/RTI, 2011).

Moving our focus to hybrid systems, they have participation in several areas, including in safety-critical areas, and consequently, there is the need to analyse the behaviour of these systems. A common approach used here is the numeric simulation of such systems. The simulation of pure continuous systems (ROBERTS; SEDRA; SEDRA; SMITH, 1992) and pure discrete systems is thereby well understood (CASSANDRAS; LAFORTUNE, 2009). There exist several numeric simulation methods for systems of Ordinary Differential Equations (ODE). However, the combination of discrete and continuous dynamics leads to challenging problems for simulation (MIXED-SIGNAL SIMULATION CHALLENGES AND SOLUTIONS, 2008).

In this context, the idea of building a simulation and verification environment to fill several gaps in this research area, led to Acumen. The main goal of the Acumen project was developing a semantic foundation that unified the formalism Functional Reactive Programming, which has been used successfully in a wide range of domains, including robotics and computer animation, with real numbers detailed treatment. A peculiarity that enhances the use of Acumen for hybrid systems is its continuous language. The use of this feature allows the use of real-valued variables, derivatives with respect to time, partial derivatives, tuples, families of equations (finite quantifiers), vectors, matrices and recursive functions during the construction of models (TAHA et al., 2015).

Other features which have been taken into account in the construction of Acumen is the accessibility (open-source) and the usability. Acumen is used in this work and a brief introduction is given in the next section, according to TAHA (2012).

2.3.1 Acumen

Acumen is an experimental environment for modelling and simulation of hybrid systems. It is built upon a textual modelling language that has the same name. Hereafter, we highlight the main features of the environment and the language. Using the Acumen GUI, shown in Figure 2.8, the user can load, edit and save textual models in the Acumen language; also the user can easily simulate models pressing a single button. Similarly, the user can view a plot, a table or a 3D visualization of the system variables over time.

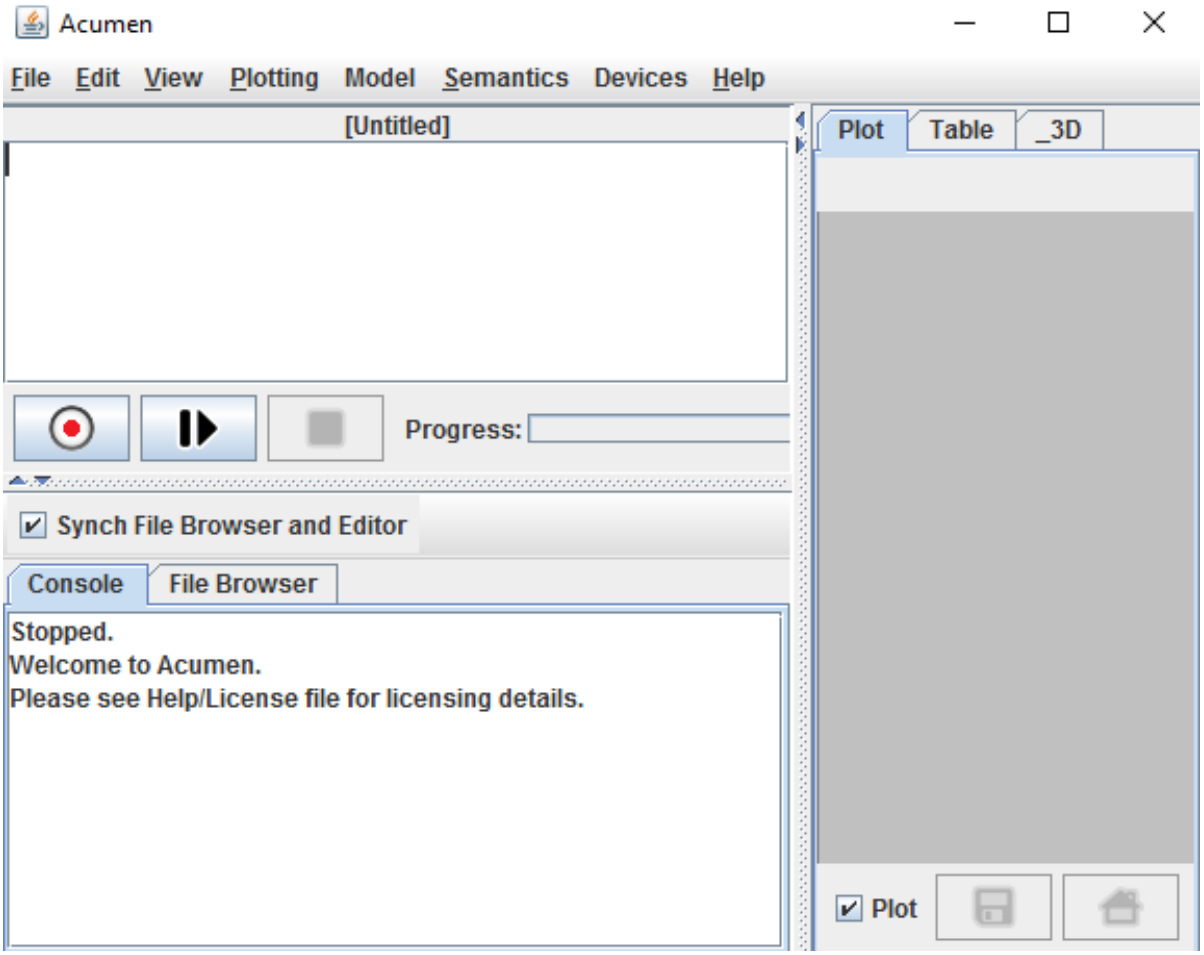

Figure 2.8: Graphical user interface of Acumen

A complete model in Acumen is composed of a set of model declarations, which may appear in any order. A valid model must contain at least a model declaration called *Main*. This main statement should have exactly one parameter, and by convention, this parameter is called *simulator*. The model declaration starts with a name for the model and, right after, a list of formal parameters between brackets, followed by an equal sign (=). After the name and the parameters, the statement may contain a section *initially*, besides an *always* section. Model statements may appear in any order. For example, a typical model has the form presented below. In this case, we are modelling a bouncing ball, whose initial position is 0 (x,y), considering a given mass and size.

```
model Ball (mass, size) =
   initially
      x_position = 0,
      y_position = 0
   always
       //the rest of the body of the declaration of the Ball model
model Main (simulator) =
   // body of the declaration of the model Main
```

Initially sections are responsible for the declaration of the variables that are used throughout the model , besides defining their initial values. *Always* sections contain a sequence of statements, usually made up of assignments and/or conditional assignments. It is important to understand that all these statements are executed at the same time. Thus, the order in which they are introduced does not matter.

In a model statement, it is allowed to instantiate objects declared in another model, via the reserved word *create*. The instantiation is permitted in both *initially* and *always* sections. When in the *initially* section, it is called a static instance, whereas in the *always* section, it is called a dynamic instance. It is shown below how instances differ: the difference between static instances and dynamic ones is that the static ones can be accessible during all the running of the program, while the dynamics ones cannot be referenced because they do not have a bound with a variable. For example, in the code below, b is a static instance, whereas the ball created later is a dynamic one.

```
   model Main (simulator) =
     initially
       b = create Ball (5, 14) // Static instance
     always
       // ...
        create Ball (10, 42)    //Dynamic instance
       // ...
```

Expressions in Acumen can be built with variables, literals, embedded functions, vector generators, and summations.

A variable has a name followed by zero or more apostrophes (’). Such apostrophes indicate that this variable is the derivative with respect to time of the variable without apostrophe. For example, x, x', x'', and x''' represent the variable x, its first, second and third derivative, respectively. Acumen defines five types of statements, namely: continuous assignments, conditional (or guarded) statements, discrete assignments, iteration, and sequences of statements. In a continuous assignment the left-hand side must be a variable or a derivative of a variable, its right side can be any expression, and the assignment operator is $(=)$. The example below shows a continuous assignment where two derivatives, from the running example, are being updated. gc' is the global clock derivative and it is assigned 1.0 to indicate a steady growth time. q' is a variable derivative and it is assigned the corresponding equation of its behaviour.

```
q'=((-q)/((r)*c))
gc'= 1.0
```

Any continuous statements on the same object are evaluated simultaneously after all discrete assignments have been made, provided no change occurs in program state.

Similarly to the continuous assignments, in discrete assignments the left-hand side must be a variable or a derivative of a variable, the right-hand side may be any expression, and the assignment operator is $(+=)$. However, discrete assignments are instantaneous. It is used to indicate that there is a discontinuous shift in a particular variable during simulation.

In order for the simulation to behave in an appropriate manner, any discrete assignment in the model definition must occur within a conditional statement. An if-statement is executed only if certain conditions are valid. The following code illustrates how if-statements are written:

```
if (x>0) then
  x'' = -9.8
else
  x' = 0
```

A match-statement is another type of a conditional statement. It can be seen as a generalisation of an if-statement, or as the switch-statement in imperative programming languages. It allows one to execute specific statements according to different conditions. These conditions are based on the value of a particular expression that is being evaluated. The following example illustrates this statement:

```
  match myCommand with
[ "Fall" ->
  x'' = -9.8
| "Freeze" ->
  x' = 0
| "Reset" ->
  x = 0
]
```

A for-statement allows the execution of an iteration, either for a particular number of times or for each element in a collection. For example:

```
foreach i in 1:10 do x=2*y
foreach c in children do c.x += 15
```

Sequential statements are delimited by a comma (,). For example, the code bellow shows a continuous statement followed by an if-statement using a discrete assignment in its body:

```
gc'=1.0,
if sin(1000000*gc) >= 0 then
  s += 1
else
  s += 0,
```

Initially, the simulation of an Acumen model has only one *Main* object. Due to the dynamic creation of objects, a tree of objects is created, where the *Main* object is always the root of the tree, and the children are the objects dynamically created.

Each simulation step involves the visit of all tree objects, starting from the $Main$. Two kinds of steps are performed, as shown in Figure 2.9, originally presented in (TAHA, 2012). During the discrete step, discrete statements and structural actions (create) are processed. Once all the discrete statements available are collected, the instructions are performed in parallel. For each object, the processing begins with the execution of its the structural actions, and then structural actions of all its children are executed. While there are active actions that change the state, the execution continues making discrete steps. Contrarily, it makes a continuous step. During a continuous step, all continuous assignments and integrations are performed.

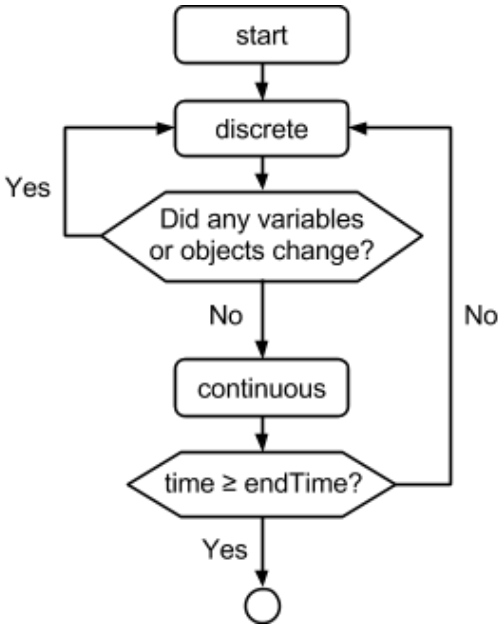

Figure 2.9: Order of evaluation

3

NAT2TEST for hybrid systems

In this chapter, we explain how the SysReq-CNL, proposed in CARVALHO (2016), is extended to enable writing requirements for hybrid systems. All changes are defined to keep the expressiveness and overall structure already provided, besides keeping all the requirements written in the old SysReq-CNL still valid in the new version. We then show how the extended SysReq-CNL is translated into a Data-Flow Reactive System for hybrid systems (h-DFRS), which is itself translated into Acumen for the purpose of simulation.

3.1 The SysReq-CNL grammar for hybrid systems

As described in Section 2.2, to express requirements of hybrid systems it is necessary to use mathematical expressions, variables, functions and derivatives. Some of these features are not supported by the previous version of the SysReq-CNL, as can be seen in Section 2.1. As previously mentioned, the extensions to the SysReq-CNL that support these features are conservative. For a better understanding, each extension is explained in isolation, using a running example (the DC-DC boost converter). For compatibility with the used parser and to facilitate the processing and analysis of the requirements, the grammar has been benefited of techniques to avoid ambiguity (SCOTT, 2005).

Hereafter, only parts of the modified grammar are shown. The whole grammar can be found in the Appendix C. In the new version of SysReq-CNL, the most crucial change was the inclusion of expressions, so that it is now possible to write expressions (see Example 3.1) in conditions instead of writing single comparisons with constant values. In this example, although S and q are being compared with constant values, they could have been compared with other variables and expressions as well. The inclusion of expressions in the grammar permits the construction of logical comparisons, such as those used in the transition guards in Figure 2.7. For instance, one way to represent the guard of transition from *mode 1* to *mode 2* is as follows.

$$S == 1 \;\&\&\; q >= 0 \tag{3.1}$$

With no doubt, this form of writing is very similar to the style adopted to express

requirements. Furthermore, the inclusion of expressions permits the use of variables or values in arithmetic expressions, making it possible to describe the behaviour of the variable q in *mode 1* as follows:

$$(Phi/L) - (1/R*C)*q \tag{3.2}$$

Considering these characteristics, it is possible to write the requirements of the boost converter related to transitions between modes. For example, the requirement representing the transition from *mode 1* to *mode 2* can be written as follows:

- When the system mode is 1, and S == 1 && q >= 0, the DC-DC boost control system shall assign 2 to the system mode.

In what follows, one can see how the original SysReq-CNL was evolved to consider expressions. Now, a *VariableState* can be written as an Expression, whereas it considered noun phrases, adverbs, adjectives, and numbers, previously. The grammar roles ensure the precedence of operators, thus an *Expression* generates a list of *AndExpression* separated by the *OR* operator. An *AndExpression* generates a list of *NotExpression* separated by the *AND* operator. A *NotExpression* generates *ComparativeExpression* or a token *LOGICALNOT* follow by a *NotExpression*. A *ComparativeExpression* generates a list of *ArithmeticExpression* separated by the *ComparativeOperator* operator. A *ComparativeOperator* could be a greater than (*GT*), less than (*LT*), greater than or equal to (*GE*), less than or equal to (*LE*), equals (*EQ*), or not equals (*NE*). An *ArithmeticExpression* generates a list of *Term* separated by the *AdditiveOperator* operators. An *AdditiveOperator* can be a token *PLUS* or *MINUS*. A *Term* generates a list of *Factor* separated by the *MultiplicativeOperator* operators. A *MultiplicativeOperator* could be a token *MULT*, or *SLASH*, or *MOD*. A *Factor* is a *PrimaryExpression*, and it may be preceded by a *PrefixOperator*. A PrefixOperator has the same roles as an *AdditiveOperator*. A *PrimaryExpression* is the core of the grammar; it can generate values, variables, anothers expression between bracket and call functions. The grammar defined four default functions, namely, *SIN*, *COS*, *EXP*, *LOG* and *SQRT*.

VariableState	→	Expression;
Expression	→	AndExpression (OR AndExpression)*;
AndExpression	→	NotExpression (AND NotExpression)*;
NotExpression	→	LOGICALNOT NotExpression | ComparativeExpression;
ComparativeExpression	→	ArithmeticExpression (ComparativeOperator ArithmeticExpression)*;
ComparativeOperator	→	GT | LT | GE | LE | EQ | NE ;
ArithmeticExpression	→	Term (AdditiveOperator Term)*;
Term	→	Factor (MultiplicativeOperator Factor)*;

Factor	→	PrefixOperator? PrimaryExpression;
PrefixOperator	→	AdditiveOperator;
AdditiveOperator	→	PLUS | MINUS;
MultiplicativeOperator	→	MULT | SLASH | MOD;
PrimaryExpression	→	FunctionID LP ArgumentList? RP
		| TRUE | FALSE | NUMBER | NounPhrase
		| LP Expression RP;
FunctionID	→	SIN | COS | EXP | LOG | SQRT | Noun;
ArgumentList	→	VariableState (COMMA VariableState)*;

Table 3.1: Part of the extended SysReq-CNL: – representing expressions

Besides expressions, now it is also possible to declare functions, with or without parameters. After declaring a function, it can be referred to (called) within a requirement. As it can be seen in Table 3.2.

The function declaration begins with an identifier, then a list of the parameters between parentheses and finally the function body. The body of a function can be just a simple expression, or a ternary expression, or can even be defined using pattern matching. These last two are constructs inserted to enhance the flexibility and the expressiveness of the grammar.

The ternary conditional operator is a way to make a simple conditional test, analogously to an if-else structure. This structure is written as follows: *test* ***?*** *expression1* ***:*** *expression2*, where *test* is any boolean expression, *expression1* is an expression evaluated if *test* is true, whereas *expression2* is evaluated otherwise.

The last construct allows the definition of expressions for pattern matching. This form is widely used in the writing of mathematical equations. The structure is as follows, *(test* ***DO*** *expression1)+*. Where test is any boolean expression and *expression1* is an expression evaluated if *test* is true. Using this form, it is possible to define functions by pattern matching.

Since our grammar allows expressions rather than being confined to literals, a peculiarity was added to simplify the writing. For example, the original grammar, a conditional $a > b \; is \; true$ is a valid conditional, in the proposed one this conditional remains valid and the conditional $a > b$ is also valid.

Table 3.2: Part of the extended SysReq-CNL: functions

Sentence	→	Requirement | **FunctionDeclaration**;
FunctionDeclaration	→	Noun LP ParameterList? RP EQUALSSIGN FunctionBody;
ParameterList	→	Noun (COMMA Noun)*;
FunctionBody	→	TernaryExpression | PatternMatching | Expression;
TernaryExpression	→	Expression IN TernaryDefinition COLON TernaryDefinition;
TernaryDefinition	→	Expression | TernaryExpession;
PatternMatching	→	(Expression DO Expression)+;

Returning to the running example, the behaviour of q in mode 1 can be described as the following function:

$$q\_mode1(Phi, L, R, C, q) = (Phi/L) - (1/R * C) * q \tag{3.3}$$

As shown in Section 2.2, in hybrid systems, the behaviour of a variable can involve the computation of derivatives. Therefore, the new version of the SysReq-CNL also allows such a characteristic. For such improvement, the new verb action *set* was incorporated into the dictionary, and the word *derivative* was defined as a reserved word, in order to denote when the derivative of a variable is being considered. In our research, the second, the third and the others derivatives were not necessary for the description of a hybrid system, but they can easily be incorporated by adding new reserved words. Now, it is possible to write a requirement to describe the behaviour of variables using function calls and derivatives. For example, the requirement for the continuous behaviour of q and Phi in the $mode1$ can be written as follows:

- When the system mode is 1, the DC-DC boost control system shall:
  set q_mode1(Phi,L,R,C,q) to the q derivative, set Phi_mode1(C,q,E) to the Phi derivative.

Now, during the syntactic analysis phase of the NAT2TEST strategy, a syntax tree is generated for each grammatically correct requirement, considering the new grammar structure. Syntax trees are also generated for the user-defined functions. To enable the parsing of these new elements, the SysReq-CNL (component of the NAT2TEST tool that parses requirements) was extended.

About the perspective of implementation, the new grammar has introduced into the parser, and the new tokens were included. In the GUI, it was necessary to create a new area responsible for creating and manipulating functions.

3.2 Semantic analysis of hybrid-system requirements

This phase consists of relating syntactic structures of grammar elements with semantic roles according to the theory of Case Grammar (FILLMORE, 1968). The relation of the meaning of each thematic role to a group of words is obtained by analysing the syntax tree generated for each valid requirement according to the new System Requirements Controlled Natural Language (SysReq-CNL) grammar.

The thematic roles considered here are the same shown in Section 2.1. However, now, the words related to each role can comprise expressions and function calls. Therefore, some inference rules considered by the RF-Generator (the NAT2TEST component that relates words to thematic roles) were updated accordingly.

A peculiarity in semantic processing is when the user decides to use one expression as a condition, for example, to write $a > b$ instead of $a > b$ *is true*, the thematic role patient

receives the expression as the value, and the same occurs with toValue. This interpretation does not affect the Data-Flow Reactive System (DFRS) generation (CARVALHO; CAVALCANTI; SAMPAIO, 2016).

For generating the thematic roles it is necessary to analyse an entire requirement. With this in mind, an analysis of the requirement introduced in the previous section is presented. Figure 3.1 shows the Requirement Frames corresponding to the semantic analysis. This requirement presents one conditional, "the system mode is 1", where, $thesystemmode$ receives the role of Patient (PAT) and 1 receives the role of Condition To Value (CTV). This requirement also contains two actions, both being executed by $theDC-DCboostcontrolsystem$ which plays the role of Agent (AGT). As there are two actions, there is also two Patient (PAT), namely, $theqderivative$ and $thePhiderivative$. And each patient receives a value, in this case, q_mode1(Phi,L,R,C,q) and Phi_mode1(C,q,E), respectively.

Meanwhile, looking on the side of development the keywords had a special treatment, the constants were changed by expressions, and consequently, the definition of expressions and types was needed.

Conditions

CPT	CAC	CMD	CFV	CTV
the_system_mode	is		-	1

Actions

AGT	ACT	TOV	PAT
the_dc-dc_boost_control_system	set	q_mode1(phi,l,r,c,q)	the_q_derivative
-	**AND**	-	-
the_dc-dc_boost_control_system	set	phi_mode1(c,q,e)	the_phi_derivative

Figure 3.1: Thematic roles from a requirement

3.3 Hybrid data-flow reactive systems

In the NATural Language Requirements to TEST Cases for hybrid systems (h-NAT2TEST), the DFRS definition, given in Section 2.1, was modified to incorporate user-defined functions. Function declarations were incorporated. As a result, the s-DFRS is now formalised as a 7-tuple: **(I, O, T, gcvar, s0, F, FD)**. Inputs *(I)* and outputs *(O)* are system variables, timers *(T)* are a special kind of variable whose values are non-negative numbers representing a discrete or a dense (continuous) time. The system global clock *(gcvar)* has the same type as the timers. The initial state is *s0*, *F* is a set of functions describing the system behaviour and *FD* are user-defined functions, which can be referred to (called) by definitions in F.

In the same way, changes on the generation of DFRS models from requirement frames were performed. In the previous version, only constants were considered and, thus, the type inference was trivial. Now, with the improvements described in this chapter, the DFRS can

also consider arbitrary expressions. This has generated the need for a new method for type inference and analysis.

Attending to the fact that the inference is mostly the analysis of expressions, this inference resembles a functional language inference. Because of that, the algorithm Hindley–Milner, see (MILNER, 1978), was selected to be implemented.

Hindley–Milner is a traditional type system for the lambda calculus with parametric polymorphism. Modern language processors use type inference techniques that are based on the algorithm W proposed by MILNER (1978). The most relevant properties of the algorithm are completeness and its ability to deduce the most general type of a given expression without the need of any kind of annotations or other suggestions provided by the programmer (HEEREN; HEEREN; HAGE; HAGE; SWIERSTRA; SWIERSTRA, 2002).

Algorithm W is fast, performing type inference in almost linear time on the size of the source, making it practically usable to type large programs. The implementation of the algorithm in the DFRS used the technique exposed in (DIEHL, 2002) and (HEEREN et al., 2002). For the implementation of this algorithm, a behavioural pattern, called visitor, was used. It was necessary to create a class for each grammar construction, responsible for representing the syntax tree, and further two major classes responsible for navigating the syntax tree and infer the types.

Now, after generating a hybrid DFRS, this model is saved in an XML file. This file acts as a possible input to other extensions of the NAT2TEST strategy, for instance, the one described in (TEST CASE GENERATION FROM NATURAL LANGUAGE REQUIREMENTS USING CPN SIMULATION, 2015), where Colored Petri Nets (CPNs) (AALST, 2015) are considered.

3.4 Evolving the NAT2TEST tool

The tool presented in (**?**) implements the concepts and features in a graphical tool. This tool was also evolved to support the characteristics described in sections 3.1, 3.2, and 3.3. In this section, we briefly describe the changes performed.

First, the tool parser receives the new grammar, along with its new tokens. After this modification, the new style of writing can already be processed by the tool. The main result of this phase is the possibility of the generation of syntax trees for requirements of hybrid systems. For instance, Figure 3.2 shows the syntax tree obtained for the following requirement:

- When the system mode is 1, the DC-DC boost control system shall: **set** the q **derivative** to q_mode1(Phi,L,R,C,q), **set** the Phi **derivative** to Phi_mode1(C,q,E).

As the new approach deals with user-defined functions, a new screen dedicated to the creation and manipulation of user-defined functions has been incorporated into the NAT2TEST tool. It works very similarly to the editing of requirements. Figure 3.3 shows the editing of the

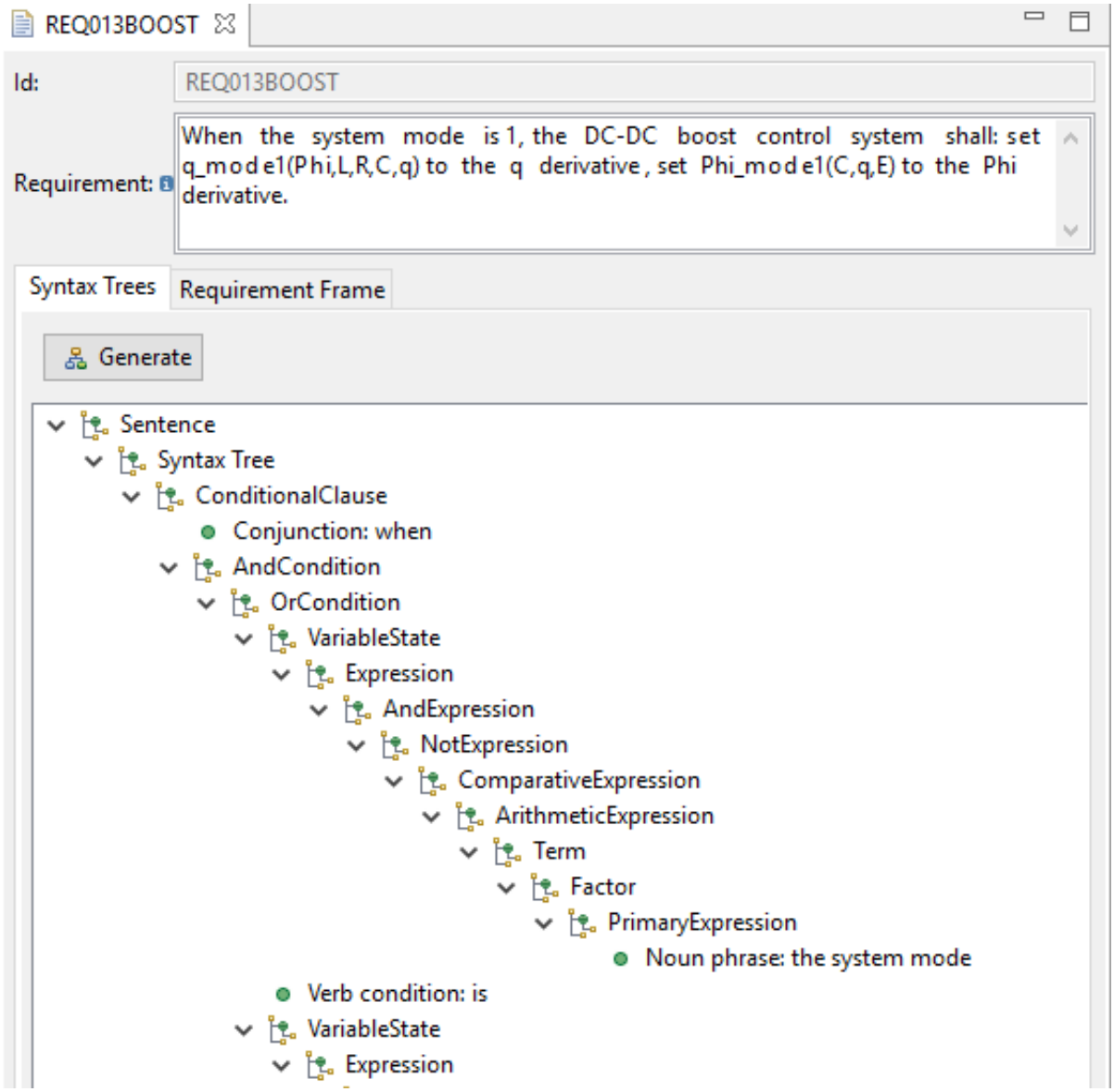

Figure 3.2: Syntax tree obtained for a requirement of a hybrid system

user-defined function q_mode1, which is used in the definition of the continuous behaviour of q in $mode1$. These functions are locally saved in *.fdef files.

The implementation of the inference rules that map words to thematic roles was also updated, since now to and from values, agents and patients can comprise expressions, and not only single constants. Figure 3.4 shows the thematic roles identified for the same requirement presented in Figure 3.2.

As previously explained, the generation of DFRS models was also updated to be able to infer the type of expressions and, thus, the type of the system variables. Moreover, now, the DFRS models are also saved as XML files. The next step of the extension to handle hybrid systems proposed here is to derive an Acumen model, which enables simulation of hybrid systems, from the hybrid DFRS model. This step is described in details in Section 3.5.1.

Figure 3.5 shows a visual representation of the variables that compose the DFRS. In

the original strategy, the kind of a variable could be an *input*, or an *output*, or a *timer*, in the proposed approach *derivative* is add to the set of kinds. Figure 3.6 shows the functions that define the system behaviour. For each statement, representing an action, it is associated a static guard or a dynamic guard, and the requirement name to provide traceability.

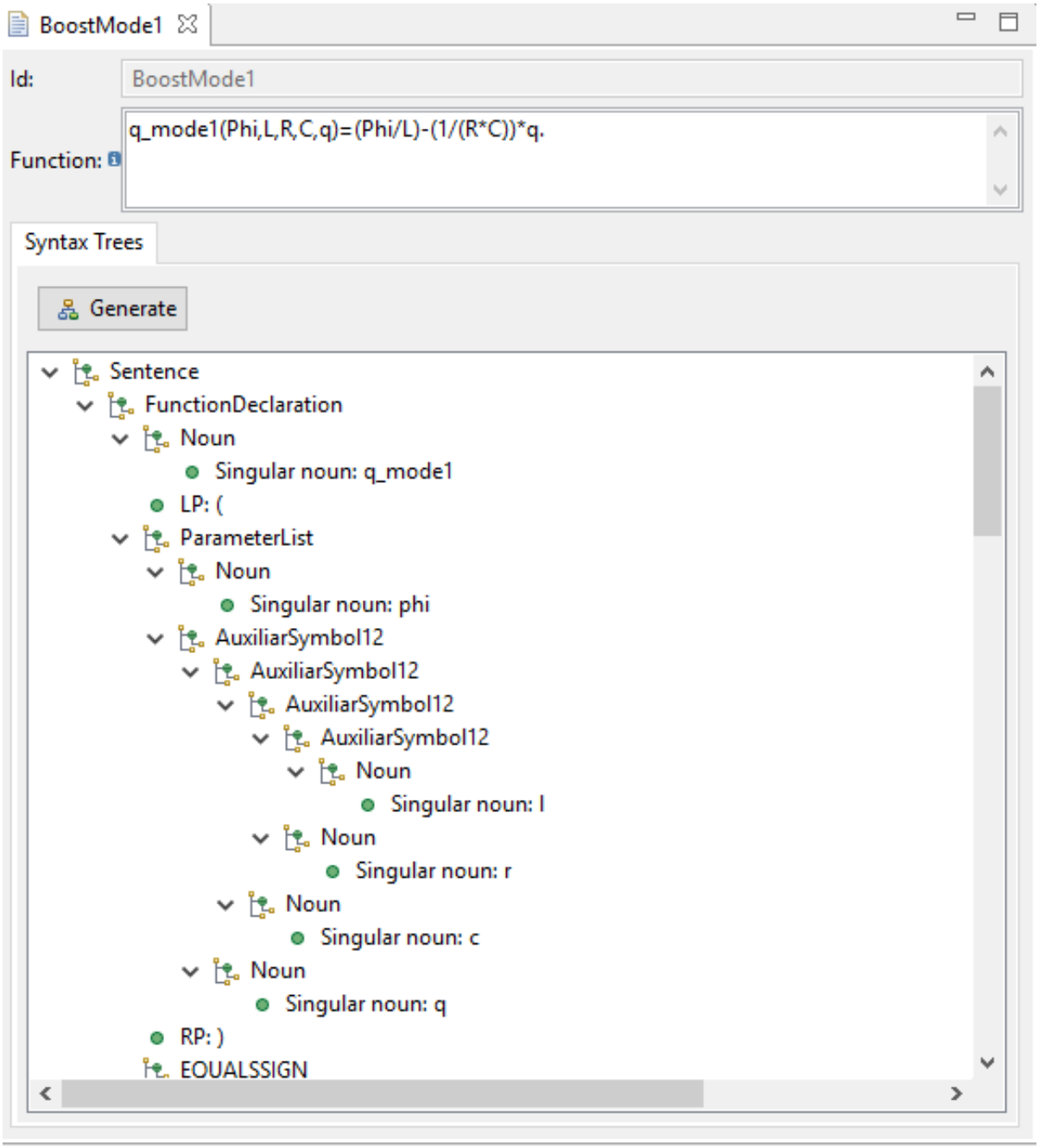

Figure 3.3: User-defined functions

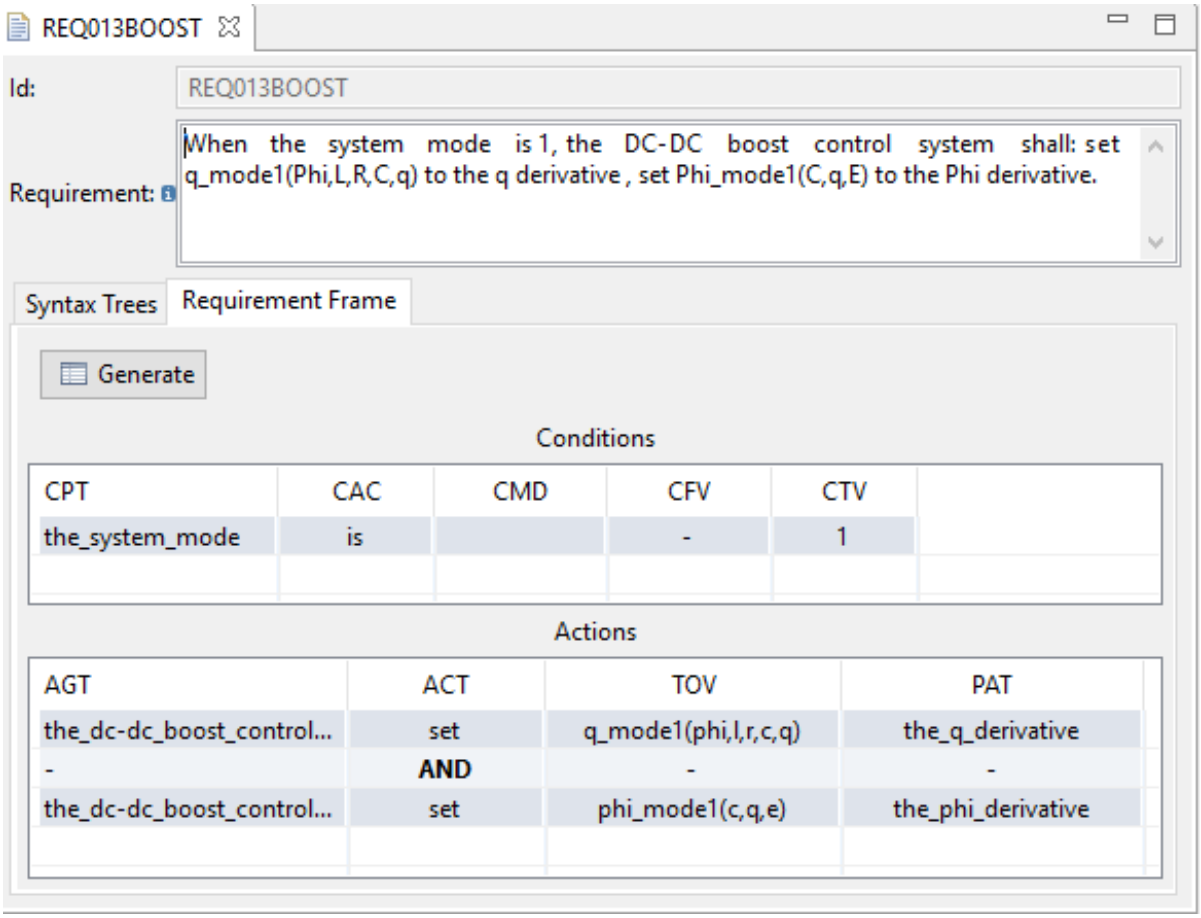

Figure 3.4: Requirement frames obtained from a requirement of a hybrid system

Kind	Name	Type
INPUT	s	INTEGER
INPUT	l	FLOAT
OUTPUT	phi	INTEGER
OUTPUT	q	FLOAT
DERIVATIVE	the_q_derivative	FLOAT
OUTPUT	r	FLOAT
OUTPUT	c	FLOAT
DERIVATIVE	the_phi_derivative	FLOAT
OUTPUT	the_system_mode	INTEGER
OUTPUT	e	FLOAT
GLOBAL CLOCK	gc	FLOAT

Figure 3.5: Variables defined from the DFRS of the running example

Static Guard	Timed Guard	Statements	Requirement Traceability
Function: the_dc-dc_boost_control_system			
the_system_mode = 2		the_q_derivative := q_mode2(r,c,q), the_phi_derivative := phi_mode2(e)	REQ0014BOOST
the_system_mode = 3		the_q_derivative := q_mode3(r,c,q), the_phi_derivative := phi_mode3()	REQ0015BOOST
((s)==1)&&((q)>0) = true AND the_system_mode = 1		the_system_mode := 2	REQ001BOOST
((phi)<=0)&&((q)>((c)*e)) = true AND the_system_mode = 1		the_system_mode := 3, phi := 0	REQ002BOOST
((s)==1)&&((q)>=0) = true AND the_system_mode = 1		the_system_mode := 4, q := 0	REQ003BOOST
((s)==0) = ((phi)>=0) AND the_system_mode = 2		the_system_mode := 1	REQ004BOOST
((s)==0)&&((phi)<=0) = true AND the_system_mode = 2		the_system_mode := 3, phi := 0	REQ005BOOST
((q)<0) = true AND the_system_mode = 2		the_system_mode := 4, q := 0	REQ006BOOST
((q)<=((c)*e)) = true AND the_system_mode = 3		the_system_mode := 1	REQ007BOOST
((s)==1)&&((q)>=0) = true AND the_system_mode = 3		the_system_mode := 2	REQ008BOOST
((s)==1)&&((q)<=0) = true AND the_system_mode = 3		the_system_mode := 4, q := 0	REQ009BOOST
((s)==0)&&((phi)>0) = true AND the_system_mode = 4		the_system_mode := 1	REQ010BOOST
((s)==0)&&((phi)<=0) = true AND the_system_mode = 4		the_system_mode := 3, phi := 0	REQ011BOOST
((q)<0) = true AND the_system_mode = 4		q := 0	REQ012BOOST
the_system_mode = 1		the_q_derivative := q_mode1(phi,l,r,c,q), the_phi_derivative := phi_mode1(c,q,e)	REQ013BOOST
the_system_mode = 4		the_q_derivative := q_mode4(), the_phi_derivative := phi_mode4(e)	REQ016BOOST

Figure 3.6: Functions defined from the DFRS of the running example

3.5 Generating Acumen specifications

After presenting a formal representation of system requirements based on DFRS models to hybrid systems, we now discuss how hybrid DFRSs can be simulated by tools that deal with models of hybrid systems. At first glance, using S-TaLiRo (VISPEC: A GRAPHICAL TOOL FOR ELICITATION OF MTL REQUIREMENTS, 2015) might seem an interesting option, since its input model is a hybrid automata, and we noticed that it is possible to derive such a model from hybrid DFRSs. However, studying in detail this tool has shown that some restrictions apply to the kind of hybrid automata that can be dealt with. Therefore, considering the Acumen language and tool (TAHA, 2012) turned out to be a better option, since its input is similar to a textual and imperative representation of hybrid automata, with less restrictions than S-TaLiRo.

In Section 3.5.1 we explain how Acumen models are derived from hybrid DFRSs. Afterwards, in Section 3.5.2, we describe how Acumen models can be used to simulate the behaviour of hybrid systems.

3.5.1 From hybrid DFRSs to Acumen models

After generating a hybrid DFRS that formally represents the system requirements, it is necessary to translate this model to an Acumen one in order to simulate it. In other words, the elements of a hybrid DFRS are mapped to the elements of an Acumen specification, ideally preserving the underlying semantics. In this work, we do not provide prove that the semantics is preserved, since it is outside the scope of this work. Therefore, it is an interesting and important topic for future work.

In order to generate syntactically correct Acumen models, it is necessary to know its syntactic grammar. The Acumen Backus-Naur Form (BNF) was provided by the authors of the Acumen tool, and can be seen in the Appendix B.

The translation from DFRSs to Acumen models is implemented with the aid of the *Visitor* design pattern; for more details, we refer to ERICH; RALPH; RICHARD; JOHN (1994). The visitor iterates over all nodes of the Extensible Markup Language (XML) representation of a DFRS, and executes the appropriate translation for each node based on the type of node, its parent and the data that it holds. In what follows, we describe the main stages of the translation:

Generating the header Stage that generates the initial settings, among them, we can highlight determining the simulation method (EulerForward by default) and the time step (it is a pre-establish value, but in the future will be edited by the user via the GUI), which is the fixed and discrete increment of time that is considered when evaluating the differential or difference equations. These settings are placed in the Main model. To exemplify, the main model of the BC is as follows:

- model Main(simulator) =
  initially

a = create DFRS()
always
simulator.timeStep+=0.0001,simulator.endTime+=0.01
,simulator.method+="EulerForward"

Generating the Variables Here, all system variables in the *initially* section are declared, along with their initial values. Model variables to represent the system inputs, outputs and timers are identified and declared. When the derivative of a variable is used at some point of the system specification, a second auxiliary variable is also declared. The variables of BC is shown below:

- $s = 0,\ l = 0.000080,\ phi = 0,\ q = 0.0,\ q' = 0.0,\ r = 20.0,\ c = 0.000040,\ phi' = 0.0,\ the\_system\_mode = 1,\ e = 12.0,\ gc = 0.0,\ gc' = 0.0$

The variable gc is the global clock; it possible to note that the variables q', phi' and gc' were created to represent the derivatives of q, phi and gc, respectively. At the moment the initial values are being assigned direct in the source code file, but there is a feature in the GUI responsible for that task, but it is missing the integration.

Generating auxiliary definitions Introduces auxiliary structures that are necessary, but are not explicitly described in the model. For example, the derivative of the *global clock* variable is set to assume a steady growth behaviour.

Generating the functions It is the most important stage of the translation. The system behaviour represented by a hybrid DFRS model (in particular, the F component, see Section 3.3) is represented in an imperative language. Here, we use if-statements, whose conditions are composed by discrete and timed guards. The body of each if-statement comprise continuous and discrete assignments according to the statements defined in the hybrid DFRS. Since Acumen does not allow the declaration of functions defined by the user to structure the specification TAHA; ZENG; DURACZ (2016), to represent the functions written by the user, we generate assignments considering an in-line version of the user-defined functions. If the user-defined function is recursive, it is not possible to consider an in-line version of it. Therefore, although our SysReq-CNL allows for the definition of recursive functions, we restrict ourselves to non-recursive functions due to this restriction of the Acumen tool. To exemplify, the functions of the BC are as follows:

- $if(((s) == 1)\&\&((q) > 0) == true\ \&\&\ the\_system\_mode == 1) then\ the\_system\_mode+ = 2\ noelse,$
- $if(((q) < 0) == true\ \&\&\ the\_system\_mode == 2) then\ the\_system\_mode+ = 4, q+ = 0\ noelse,$
- $if(the\_system\_mode == 3) then\ q' = ((-q)/((r) * c)), phi' = 0\ noelse,$

- $if(((s) == 1)\ \&\&\ ((q) <= 0) == true\ \&\&\ the\_system\_mode == 3) then$
  $the\_system\_mode+ = 4, q+ = 0\ noelse,$
- $if(((s) == 0)\ \&\&\ the\_system\_mode == 2) then$
  $the\_system\_mode+ = 1\ noelse,$
- $if(((q) < 0) == true\ \&\&\ the\_system\_mode == 4) then$
  $q+ = 0\ noelse,$
- $if(((q) <= ((c) * e)) == true\ \&\&\ the\_system\_mode == 3) then$
  $the\_system\_mode+ = 1\ noelse,$
- $if(the\_system\_mode == 2) then$
  $q' = ((((-1)/((r) * c))) * q), phi' = e\ noelse,$
- $if(((s) == 1)\ \&\&\ ((q) <= 0) == true\ \&\&\ the\_system\_mode == 1) then$
  $the\_system\_mode+ = 4, q+ = 0\ noelse,$
- $if(((s) == 0)\ \&\&\ ((phi) < 0) == true\ \&\&\ the\_system\_mode == 4) then$
  $the\_system\_mode+ = 3, phi+ = 0\ noelse,$
- $if(the\_system\_mode == 1) then$
  $q' = ((((phi)/l)) - ((((1)/((r) * c))) * q)), phi' = ((((((-1)/c)) * q)) + e)\ noelse,$
- $if(((s) == 1)\ \&\&\ ((q) >= 0) == true\ \&\&\ the\_system\_mode == 3) then$
  $the\_system\_mode+ = 2\ noelse,$
- $if(the\_system\_mode == 4) then$
  $q' = 0, phi' = e\ noelse,$
- $if(((s) == 0)\ \&\&\ ((phi) >= 0) == true\ \&\&\ the\_system\_mode == 4) then$
  $the\_system\_mode+ = 1\ noelse,$
- $if(((phi) <= 0)\ \&\&\ ((q) > ((c) * e)) == true\ \&\&\ the\_system\_mode == 1) then$
  $the\_system\_mode+ = 3, phi+ = 0\ noelse,$
- $if(((s) == 0)\ \&\&\ ((phi) <= 0) == true\ \&\&\ the\_system\_mode == 2) then$
  $the\_system\_mode+ = 3, phi+ = 0\ noelse$

Generating the changes of the inputs Finally, in order to enable simulation of Acumen models, it is necessary to describe how the input variables evolve over time, considering their expected values. If the input has only one expected value, no function change is created. If the input has two possible values of processing, we use a sine function to provide the variation between these values. The changes of the input s, with two expect values (0 or 1), is shown below:

- $if\ sin(1000000 * gc) >= 0\ then$
  $s+ = 1$

$else$

$s+=0$

3.5.2 Simulating the generated Acumen model

As described in Section 3.5, the tool Acumen allows for model simulation. The Acumen model statically describes the system behaviour, whereas the simulation illustrates how the system behaviour evolve over time. Simulating the Acumen model derived from natural-language requirements is a useful validation activity. Figure 3.7 shows the graphical result of the simulation of the running example; this simulation is explained in details in Section 4.1.

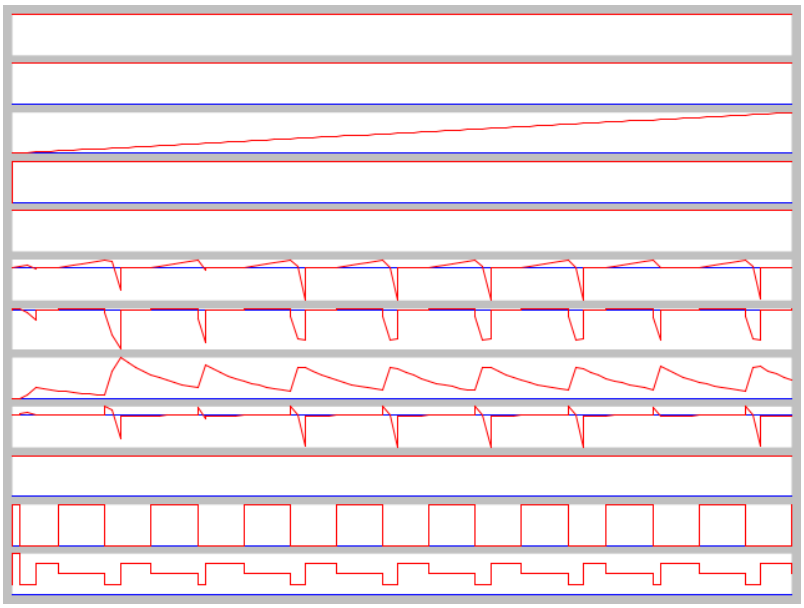

Figure 3.7: Running example simulation

In our approach, after translating a hybrid DFRS into an Acumen model, a file *.acm* is generated with the full Acumen code representing the system behaviour. When this file is opened in the tool, it is possible to simulate the corresponding model. As simulation output, the tool plots the values of each variable over time.

4

Application of the h-NAT2TEST strategy

This chapter describes the application of the proposed hybrid version of the NAT2TEST strategy. The examples considered here were selected in order to cover the maximum possible characteristics of the proposed strategy, whilst being typical examples of hybrid systems.

It is important to note that, although it is not our focus here, it is also possible to use the h-NAT2TEST strategy to perform conformance testing. To this goal, besides deriving an Acumen specification model from natural-language requirements, it is required to have a Matlab implementation model. Here, we focus on generating Acumen models for simulation purposes.

First (Section 4.1), we illustrate all steps of the h-NAT2TEST strategy considering our running example (a DC-DC Boost Converter (BC)). Then (Section 4.2), the same steps are performed but with respect to an Adaptive Cruise Control (ACC).

4.1 A DC-DC boost converter

The general idea of a BC was previously presented in Section 2.2.3. It is used to boost an input DC voltage to an increased output value. Hereafter, we describe the application of each step of the h-NAT2TEST strategy considering this example.

4.1.1 Writing the system requirements

Since natural-language requirements were not available for this example, but only the automaton model showed in Figure 2.7, we manually wrote the requirements taking this model as input.

First, we note that in each one of the four discrete states of the automaton there are equations to update the system variables. These equations have been represented as function declarations. Eight functions were created (one for each output variable with respect to each state).

FUN-001 q_mode1(Phi,L,R,C,q) = (Phi/L) - (1/R*C)*q.

FUN-002 Phi_mode1(C,q,E) = (-1/C)*q + E.

FUN-003 q_mode2(R,C,q) = (-1/R*C)*q.

FUN-004 Phi_mode2(E) = E.

FUN-005 q_mode3(R,C,q) = (-1/R*C)*q.

FUN-006 Phi_mode3() = 0.

FUN-007 q_mode4() = 0.

FUN-008 Phi_mode4(E) = E.

Considering these user-defined functions, requirements were written to describe the discrete transitions between states of the automaton. These transitions update the system mode, besides performing discrete assignments. Twelve requirements were written, and are shown below.

REQ-001 When the system mode is 1, and S == 1 && q >= 0, the DC-DC boost control system shall assign 2 to the system mode.

REQ-002 When the system mode is 1, and Phi <= 0 && q > C*E, the DC-DC boost control system shall: assign 3 to the system mode, assign 0 to Phi.

REQ-003 When the system mode is 1, and S == 1 && q >= 0, the DC-DC boost control system shall: assign 4 to the system mode, assign 0 to q.

REQ-004 When the system mode is 2, and S == 0 && Phi >= 0, the DC-DC boost control system shall assign 1 to the system mode.

REQ-005 When the system mode is 2, and S == 0 && Phi <= 0, the DC-DC boost control system shall: assign 3 to the system mode, assign 0 to Phi.

REQ-006 When the system mode is 2, and q < 0, the DC-DC boost control system shall: assign 4 to the system mode, assign 0 to q.

REQ-007 When the system mode is 3, and q <= C * E, the DC-DC boost control system shall assign 1 to the system mode.

REQ-008 When the system mode is 3, and S == 1 && q >= 0, the DC-DC boost control system shall assign 2 to the system mode.

REQ-009 When the system mode is 3, and S == 1 && q <= 0, the DC-DC boost control system shall: assign 4 to the system mode, assign 0 to q.

REQ-010 When the system mode is 4, and S == 0 && Phi > 0, the DC-DC boost control system shall assign 1 to the system mode.

REQ-011 When the system mode is 4, and S == 0 && Phi <= 0, the DC-DC boost control system shall: assign 3 to the system mode, assign 0 to Phi.

REQ-012 When the system mode is 4, and q < 0, the DC-DC boost control system shall assign 0 to q.

Finally, requirements were written to describe the continuous evolution of the system (the continuous assignments performed within each state). In these requirements, the previously defined functions are called, and the reserved word *derivative* is used. These requirements are shown in what follows.

REQ-013 When the system mode is 1, the DC-DC boost control system shall: **set** the q **derivative** to q_mode1(Phi,L,R,C,q), **set** the Phi **derivative** to Phi_mode1(C,q,E).

REQ-014 When the system mode is 2, the DC-DC boost control system shall: **set** the q **derivative** to q_mode2(R,C,q), **set** the Phi **derivative** to Phi_mode2(E).

REQ-015 When the system mode is 3, the DC-DC boost control system shall: **set** the q **derivative** to q_mode3(R,C,q), **set** the Phi **derivative** to Phi_mode3().

REQ-016 When the system mode is 4, the DC-DC boost control system shall: **set** the q **derivative** to q_mode4(), **set** the Phi **derivative** to Phi_mode4(E).

4.1.2 Inferring thematic roles

After verifying whether each system requirement is correct with respect to the SysReq-CNL grammar, thematic roles are inferred from the obtained syntax trees. To illustrate with an example, the requirement frame of the requirement REQ001 is shown in Figure 4.1. This requirement describes a transition between modes 1 and 2. We note that the expression used in this requirement is properly mapped to the CTV role. In the original NAT2TEST strategy, we only had literals related to this role. Figure 4.2 shows the requirement frame of the requirement RE016, which describes the continuous evolution of q and Phi in the mode 4. Here, the function calls are properly mapped to the TOV role. We note that the expression used in this requirement is properly mapped to the CTV and CPT roles. In the original NAT2TEST strategy, we only had literals related to these roles. This information is useful, since later two specific variables are created to represent the first derivative of q and Phi. The expressions represented here are not evaluated now, but only when performing simulation of the corresponding Acumen model.

4.1.3 Generating hybrid DFRSs and Acumen models

From the inferred requirement frames a hybrid DFRS is generated. The process of generating DFRSs automatically infers the initial value of each system variable: 0 is the default

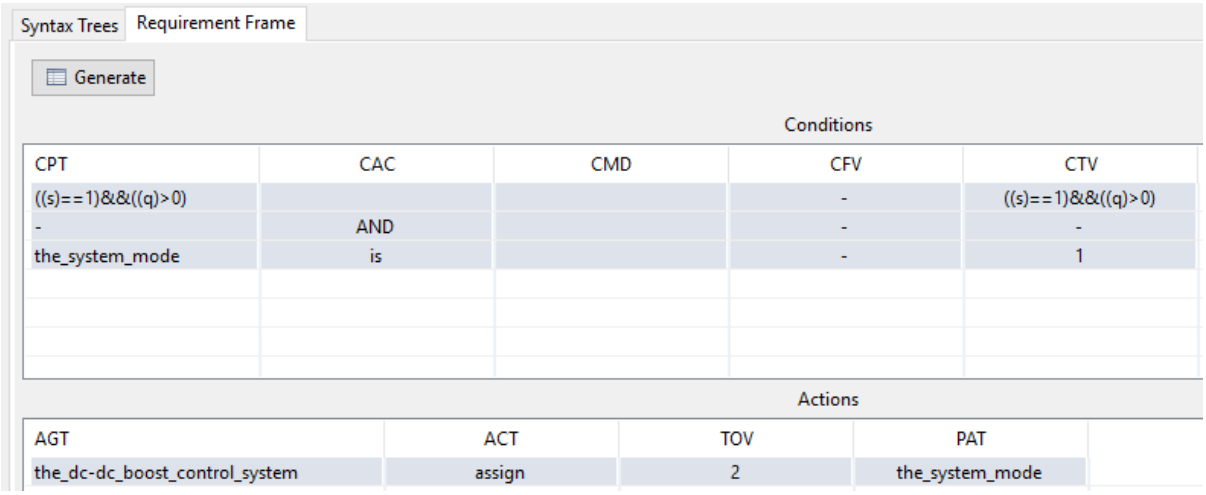

Figure 4.1: Requirement frame of REQ001

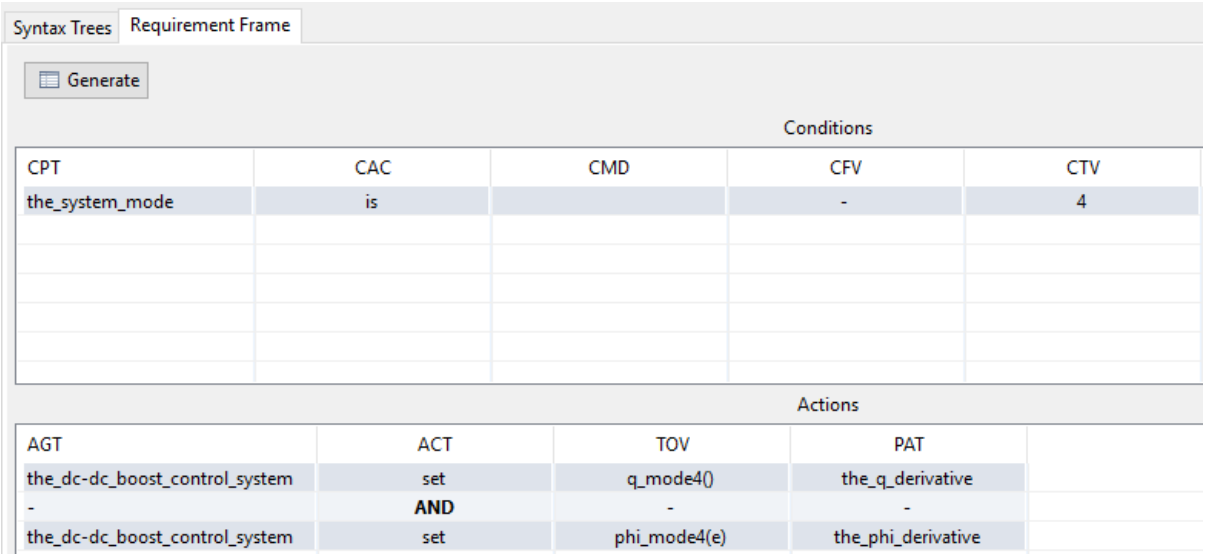

Figure 4.2: Requirement frame of REQ016

initial value for integers, whereas 0.0 and false are the default values for floating numbers and booleans, respectively. However, considering the DC-DC boost converter, other initial values are typically expected: s = 0; l = 0.000080; phi = 0.0; q = 0.0; the q derivative = 0.0; r = 20.0; c = 0.000040; the phi derivative = 0.000040; the system mode = 1; e = 12.0. Therefore, these values were manually set via the NAT2TEST tool (see Figure 4.3). The complete XML file generated for the hybrid DFRS of the BC can be seen in Appendix C.1.

Kind	Name	Type	Expected Values	Initial Value
INPUT	s	INTEGER	{((phi)>=0), ((s)==0)&&((phi)<=0), ((s)==0)&&((phi)>0), ((s)==1)&&((q)<=0), ((s)...	0
INPUT	l	FLOAT	{0.0, q_mode1(phi,l,r,c,q)}	0.0
OUTPUT	phi	INTEGER	{((phi)<=0)&&((q)>((c)*e)), ((s)==0)&&((phi)<=0), ((s)==0)&&((phi)>0), 0, q_mod...	0
OUTPUT	q	FLOAT	{((phi)<=0)&&((q)>((c)*e)), ((q)<0), ((q)<=((c)*e)), ((s)==1)&&((q)<=0), ((s)==1)&...	0.0
DERIVATIVE	the_q_derivative	FLOAT	{0.0, q_mode1(phi,l,r,c,q), q_mode2(r,c,q), q_mode3(r,c,q), q_mode4()}	0.0
OUTPUT	r	FLOAT	{0.0, q_mode1(phi,l,r,c,q), q_mode2(r,c,q), q_mode3(r,c,q)}	0.0
OUTPUT	c	FLOAT	{((phi)<=0)&&((q)>((c)*e)), ((q)<=((c)*e)), 0.0, phi_mode1(c,q,e), q_mode1(phi,l,r,c,...	0.0
DERIVATIVE	the_phi_derivative	FLOAT	{0.0, phi_mode1(c,q,e), phi_mode2(e), phi_mode3(), phi_mode4(e)}	0.0
OUTPUT	the_system_mode	INTEGER	{0, 1, 2, 3, 4}	1
OUTPUT	e	FLOAT	{((phi)<=0)&&((q)>((c)*e)), ((q)<=((c)*e)), 0.0, phi_mode1(c,q,e), phi_mode2(e), phi...	0.0
GLOBAL CLOCK	gc	FLOAT	-	

Figure 4.3: Screen of the DFRS

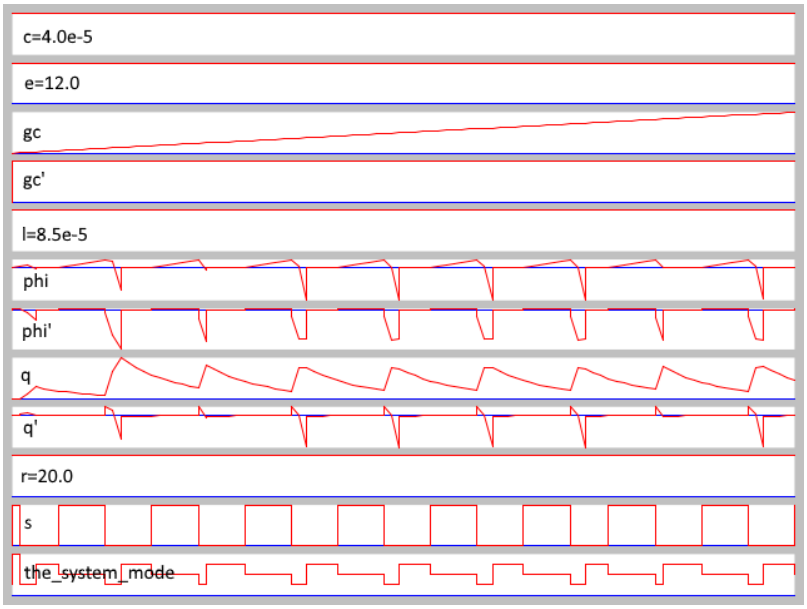

Figure 4.4: Simulation of the Acumen model obtained for the boost converter

The generated XML file, which describes the hybrid DFRS, is used as input to generate an Acumen specification model according to the steps discussed in Section 3.5.1. The complete Acumen model is available in Appendix D.1. After simulating this model, the Acumen tool plots the graphic shown in Figure 4.4. As it can be seen from the figure, the system global clock grows steadily, and, the switch changes discretely and periodically between “on” and “off” and this variation causes changes in the values of other variables, based on the behaviour defined in the system requirements. An important point to note is that the charge q suffers a high oscillation during the simulation, which in real world conditions would be something inconsistent,

but when the frequency of switch is increased, see figure4.5, a remarkable stabilisation of the charge is noticeable.

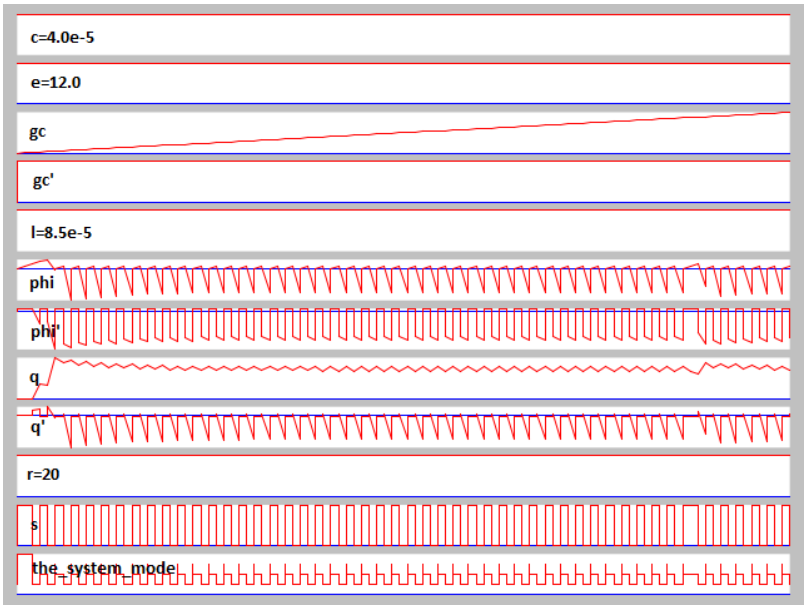

Figure 4.5: Simulation of the Acumen model obtained for the boost converter (2)

4.2 Adaptive Cruise Control

Adaptive Cruise Control (ACC) is an intelligent sort of cruise control, similar to the conventional one in which the driver sets the maximum speed, and the vehicle maintains this pre-set speed. ACC systems have been developed to aid vehicular traffic on highways, when during long drives a motorist can transfer control to an intelligent vehicular controller system. However, unlike conventional cruise control, this new technology can automatically adjust speed to sustain a proper distance between vehicles in the same lane. This distance is measured by a small radar unit behind the front grille or under the bumper.

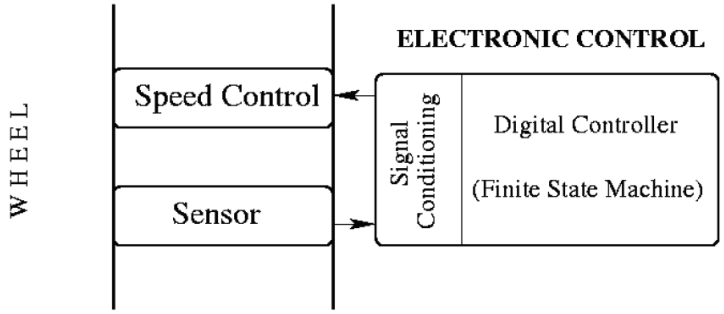

Figure 4.6: Mechanisms of a vehicular speed control

Figure 4.6, reproduced from (VERIFICATION OF A MEMS BASED ADAPTIVE CRUISE CONTROL SYSTEM USING SIMULATION AND SEMI-FORMAL APPROACHES, 2008), shows

a speed control mechanism used in vehicles. In this system, a gyroscope is attached at the wheel base to provide details on the vehicular speed. Speed and proximity sensors are also deployed at the front and back end of the vehicle to continuously observe the speed and distance of the vehicles ahead and behind. All these items are then sampled by an engine controller to take a proper course of action for the vehicle.

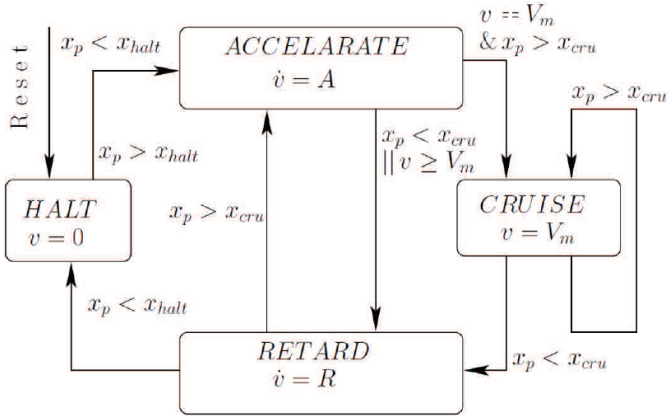

Figure 4.7: A state machine for an adaptive cruise control system

The vehicular control is characterised by a closed loop control chain, which works in a continuous polling mode. It represents a state machine, which is created to interpret the traffic conditions and generate an action for each case. Figure 4.7, originally presented in (VERIFICATION OF A MEMS BASED ADAPTIVE CRUISE CONTROL SYSTEM USING SIMULATION AND SEMI-FORMAL APPROACHES, 2008), describes a state machine for an ACC system. The system behaviour consists of four states: *HALT*, *ACCELARATE*, *CRUISE* and *RETARD*. The variables x_p (for proximity to the front vehicle) and v (for speed) govern the assignments to different situations and the transitions between states. The system is self-explanatory about the behaviour of state changes, its initial state is "HALT" and the differential equations for *ACCELARATE* and *RETARD* are: $\dot{v} = A$ and $\dot{v} = R$ These states are responsible for activating subsystems in charge of acceleration, retardation or maintaining a uniform speed of the vehicle. During all execution, the sensors keep polling for front and back vehicular proximity for a continuous update of the system (VERIFICATION OF A MEMS BASED ADAPTIVE CRUISE CONTROL SYSTEM USING SIMULATION AND SEMI-FORMAL APPROACHES, 2008).

4.2.1 Writing the system requirements

Similarly to the BC example, natural-language requirements were not available for this example, but only the automaton model shown in Figure 4.7 and, thus, we manually wrote the requirements taking this model as a reference. For easier reading of the text, the variables received intuitive names: v was renamed to *velocity*, R to *retardation*, A to *acceleration*, x_p to

vehicle proximity, x_{halt} to *halt proximity* x_{cru} to *cruise proximity*, and, finally, V_m was renamed to *cruise velocity*.

Differently from the BC, since only simple assignments (literals to variables) are performed, auxiliary user-defined functions were not created. In what follows, one can see the requirements that describe the discrete and continuous assignments performed within each state. A point to note is that the system mode is no longer an integer and has become a string showing the freedom of expression that can be used in writing the requirements.

REQ-001 When the system mode is HALT, the ACC shall assign 0 to velocity.

REQ-002 When the system mode is CRUISE , the ACC shall assign cruise speed to velocity.

REQ-003 When the system mode is ACCELARATE , the ACC shall set velocity derivative to acceleration.

REQ-004 When the system mode is RETARD , the ACC shall set velocity derivative to retardation.

Similarly to the boost converter example, requirements were written for each transition between states, except for the self-transition with respect to the *CRUISE* state. The underlying reason is that, according to the DFRS semantics, only transitions that have side effects needs to be explicitly modelled. The requirements created after the automaton transitions are shown below.

REQ-005 When the vehicle proximity < the halt proximity, the ACC shall assign HALT to the system mode.

REQ-006 When the vehicle proximity > the halt proximity, and the system mode is HALT, the ACC shall assign ACCELARATE to the system mode.

REQ-007 When the vehicle proximity > the cruise proximity, and the system mode is RETARD, the ACC shall assign ACCELARATE to the system mode.

REQ-008 When the vehicle proximity < the cruise proximity or velocity >= cruise speed,and the system mode is ACCELARATE, the ACC shall assign RETARD to the system mode.

REQ-009 When the vehicle proximity > the cruise proximity, and velocity > cruise speed , and the system mode is ACCELARATE, the ACC shall assign CRUISE to the system mode.

REQ-010 When the vehicle proximity < the cruise proximity, and the system mode is CRUISE, the ACC shall assign RETARD to the system mode.

4.2.2 Inferring thematic roles

As in the BC exemple, after verifying whether each system requirement is correct with respect to the SysReq-CNL grammar, thematic roles are inferred from the obtained syntax trees. To illustrate this example, the requirement frame of the requirement REQ010 is shown in Figure 4.8. This requirement describes a transition between modes *CRUISE* and *RETARD*. We note that the expression used in this requirement is properly mapped to the CTV and CPT roles. In the original NAT2TEST strategy, we only had literals related to these roles. The expressions represented here are not evaluated at this stage, but only when performing simulation of the corresponding Acumen model.

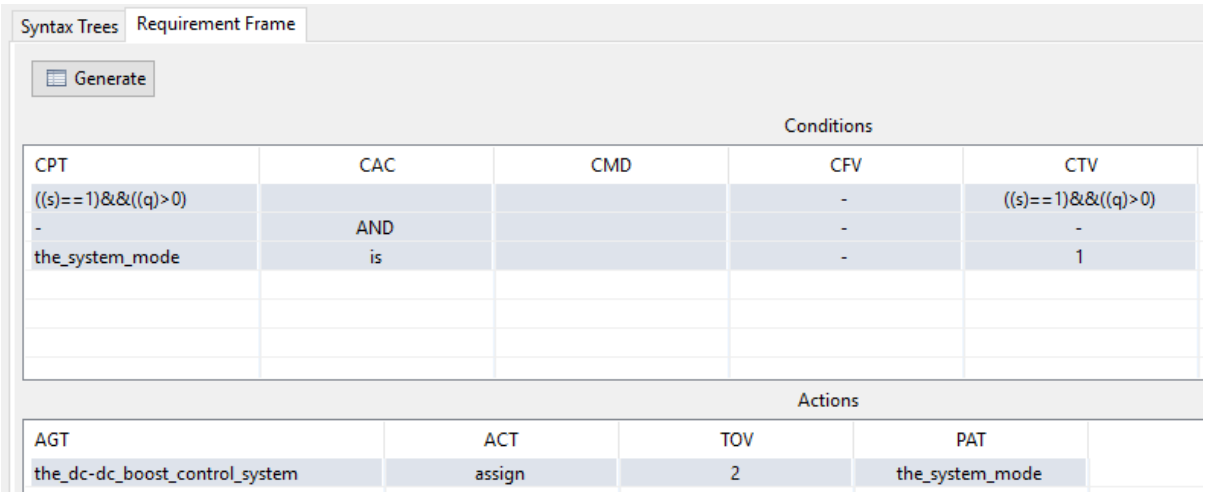

Figure 4.8: Requirement and requirement frame of REQ010

4.2.3 Generating hybrid DFRSs and Acumen models

From the inferred requirement frames a hybrid DFRS is generated. Similarly to the boost converter example, here we also need to provide the initial values for the system variables: *acceleration* = 100, *halt* = “halt”, cruise_speed=50, the_cruise_proximity=15, the_halt_proximity=5, retardation=-5, the_vehicle_proximity=10, the_system_mode=“accelarate”, velocity=0, velocity’=0.

As previously mentioned, the expressions are evaluated only during the simulation, as a consequence, the expected values are composed by the entire expression, see Figure 4.9. Consequently, the expression as a whole is a valid value accepted as the initial value.

We note that in this example we use string literals, which are also supported by the Acumen tool. The complete XML file generated for the hybrid DFRS of the ACC can be seen in Appendix C.2. Similarly, this file is used as input to derive an Acumen specification model (see Appendix D.2). In order to produce an interesting simulation, it was defined manually how the input values should change over time, instead of considering a purely random approach. The simulation of this model yields the graphic shown in Figure 4.10.

Kind	Name	Type	Expected Values	Initial Value
INPUT	acceleration	NUMBER	-	
INPUT	cruise_speed	INTEGER	{((velocity)>=cruise_speed), ((velocity)>cruise_speed), cruise_speed}	((velocity)>=cruise_speed)
INPUT	the_cruise_proximity	INTEGER	{((the_vehicle_proximity)<the_cruise_proximity), ((the_vehicle_proximity...	((the_vehicle_proximity)<the_cruise_proximity)
INPUT	the_halt_proximity	NUMBER	{((the_vehicle_proximity)<the_halt_proximity), ((the_vehicle_proximity)>...	((the_vehicle_proximity)<the_halt_proximity)
INPUT	retardation	NUMBER	-	
INPUT	the_vehicle_proximity	NUMBER	-	
OUTPUT	the_system_mode	INTEGER	{accelarate, cruise, halt, retard}	accelarate
OUTPUT	velocity	INTEGER	{((velocity)>=cruise_speed), ((velocity)>cruise_speed), 0, cruise_speed}	((velocity)>=cruise_speed)
DERIVATIVE	velocity_derivative	INTEGER	{acceleration, retardation}	acceleration
GLOBAL CLOCK	gc	FLOAT	-	

Figure 4.9: Screen of the DFRS

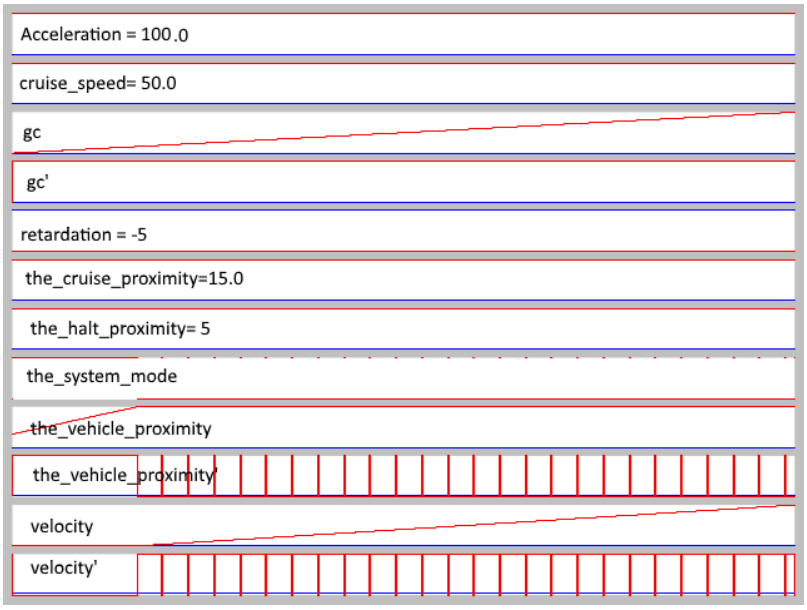

Figure 4.10: Simulation - Adaptive Cruise Control (ACC)

As it can be seen from the figure, the system global clock grows steadily, and the state changes discretely and periodically between “ACCELARATE” and “RETARD”. This variation causes changes in the values of other variables, based on the behaviour defined in the system requirements.

5

Conclusion

The major motivation for this work is to provide means for simulating natural-language requirements of hybrid systems, in order to enable validation of these requirements. In (CARVALHO, 2016), it is proposed a strategy (NAT2TEST) for analysing and testing reactive systems from natural-language requirements. Here, we extend this work to the domain of hybrid systems.

The strategy proposed here (h-NAT2TEST) extends the first three stages of the original NAT2TEST strategy: syntactic and semantic analysis, and generation of DFRS models. In the syntatic analysis, we proposed a new version of the controlled natural language SysReq-CNL, adding to it the manipulation of expressions, besides declaring and referring to differential and difference equations. This new version is a conservative extension of the original one, since requirements for reactive systems are still valid according to the extended SysReq-CNL grammar. In the semantic analysis, the inference rules of thematic roles were updated considering the new elements now supported by the SysReq-CNL grammar. In the DFRS Generation stage, we now consider a hybrid DFRS, extended with user-defined functions, as well as type inference algorithm that is capable of dealing with expressions. Finally, a translation from hybrid DFRSs to Acumen models was implemented, allowing the user to simulate natural-language requirements of hybrid systems via the Acumen tool, without the knowledge of the intermediate notations employed by the h-NAT2TEST strategy.

To demonstrate the feasibility of the h-NAT2TEST strategy, two examples were considered to illustrate the application of the full approach. The first example is a DC-DC boost converter, a classic example in the electronics field, and the second one is an adaptive cruise control system being widely employed by the automotive industry.

5.1 Related Work

Here, we present and discuss works related to our proposal extension of the NAT2TEST strategy.

LUTESS A testing environment that supports highly automated testing of synchronous reactive systems. During the validation process, Lutess requires three elements: the system

under test, its environment description and an oracle. The test is performed on a single action-reaction cycle, operated by the generator. The generator randomly chooses an input vector for the system under test and sends it to the latter. The unit under test reacts with an output vector and feeds back the generator with it. The generator proceeds by producing a new input vector and the cycle is reproduced. The oracle observes the program inputs and outputs and decides whether the software requirements are violated. The test data generator is automatically built by the tool from an environment description written in LUSTRE. The language is detailed in (HALBWACHS; CASPI; RAYMOND; PILAUD, 1991).

LURETTE Is developed by the same team of the LUTESS. It is an automatic test generator for reactive systems. Testing is automated in two principal approaches: (1) realistic input sequences are generated from non-deterministic formal specifications of the SUT environment properties; (2) the test success decision is done with a formal description of the wanted properties (proper behaviours) of the SUT. The environment is modelled using dynamically changing constraints on inputs described using Lucky (RAYMOND; ROUX; JAHIER, 2008) and for the system it is used a higher-level language named stochastic Lutin. The language is detailed in (DESCRIBING AND EXECUTING RANDOM REACTIVE SYSTEMS, 2006).

GATeL Is a tool developed by the French Nuclear Research Agency (CEA). Its focus is not generating plenty of test cases, but test cases converged to a problem. GATeL requires three inputs: a model of the system in Lustre, a model containing the aspects of the environment (also in Lustre), and a declarative definition of desired test cases (MARRE; ARNOULD, 2000).

LBTest Is a tool for requirements testing of embedded and reactive systems based on the principles of learning-based testing (LBT), an emerging technique for black-box requirements testing. The general concept of LBT is to automatically create a large quantity of test cases by combining an incremental automata learning algorithm or a model inference algorithm with a model checking algorithm. LBTest uses as input a formal requirements model written in linear temporal logic (LTL). LTL is an expressive logical language applied in LBTest to describe both safety and liveness properties of embedded systems (LBTEST: A LEARNING-BASED TESTING TOOL FOR REACTIVE SYSTEMS, 2013).

KeYmaera Is a deductive verification tool for hybrid systems. It was developed as a combination of the deductive theorem prover KeY (VERIFYING OBJECT-ORIENTED PROGRAMS WITH KEY: A TUTORIAL, 2007) with the computer algebra system Mathematica (WOLFRAM, 1988). The systems are described in differential dynamic logic that enables the proof of correctness, safety, controllability, reactivity, and liveness properties of hybrid systems.

20-sim Is a modeling and simulation program for mechatronic systems. The model is built graphically by drawing an engineering scheme, and it is possible to create models using equations, block diagrams, physical components and bond graphs. 20-sim contains simulation algorithms for solving ordinary differential equations (ODE) and differential algebraic equations (DAE) (WIKIPEDIA, 2016).

SpaceEx Is a new verification platform for hybrid systems. It uses its languagem as input, the SpaceEx model language (COTTON; FREHSE; LEBELTEL, 2010) that is based on components built as hybrid automata. This language induces a model hierarchy since components can be composed to form new components (SPACEEX: SCALABLE VERIFICATION OF HYBRID SYSTEMS, 2011).

CTE XL (Classification Tree Editor eXtended Logics) is a tool that stands out for using the Classification-Tree Method (GROCHTMANN; GRIMM, 1993). It expresses logical dependencies in formulas of propositional logic, and any logical rules can be related to a classification tree.

MuProD Is an IBM's project that deals with introducing in-line testing techniques to production processes. It aims to develop a model-based test-generation expert system for simulation of production systems at the high system level.

Ulysses Is a test case generator following the model-based mutation testing strategy. Basically, the tool mutates UML models (input) and generate the test cases that would kill a set of mutated models (FAULT-BASED GENERATION OF TEST CASES FROM UML-MODELS–APPROACH AND SOME EXPERIENCES, 2011). To generate test cases for hybrid system models, it accepts as input a hybrid variant of action systems, called Qualitative Action Systems (QAS) (QUALITATIVE ACTION SYSTEMS, 2009).

TorX Is a testing tool for conformance testing of reactive systems. The tool requires as input a real implementation and a formal specification. Then, the tool verifies whether the implementation is correct with respect to the given specification considering a conformance relation (TIMED TESTING WITH TORX, 2006).

Autofocus Is a tool for the graphical specification and validation of embedded systems. The tool uses behaviour models as input in a language quite similar to a subset of the UML-RT (PRETSCHNER; LOTZBEYER; PHILIPPS, 2001).

AutoLink Is to a tool that aims to simplify the test generation process in order to get error-free test suites in less time. Message Sequence Charts (MSC) are used to describe the interaction between the SUT and the test equipment. Based on these MSCs and an Specification and Description Language (SDL) specification, the tool generates the test cases according to the Tree and Tabular Combined Notation (TTCN); see more details in (PROBERT; MONKEWICH, 1992).

S-Taliro Is a tool to check and test Cyber-Physical Systems (CPSs). It is a modular software tool built on the Matlab platform. S-TALIRO can analyse hybrid automata, user-defined functions as BlackBox, arbitrary Simulink models, hardware-in-the-loop and processor-in-the-loop. S-TALIRO performs automated randomised trials based on stochastic optimisation techniques. The system requirements are defined in Metric Temporal Logic (MTL) (VISPEC: A GRAPHICAL TOOL FOR ELICITATION OF MTL REQUIREMENTS, 2015).

(AERTS; MOUSAVI; RENIERS, 2015) Proposes a tool prototype for model-based testing of cyber-physical systems. It is implemented in Matlab and comprises three stages of model-based testing, specifically, test case generation, test case execution, and conformance analysis. The tool requires a hybrid system model in a domain-specific language called Acumen Modeling Language (TAHA; BRAUNER; CARTWRIGHT; GASPES; AMES; CHAPOUTOT, 2010), and an implementation in Matlab.

Simulink is a software for modelling, simulating and analysing dynamic systems. It supports linear and nonlinear systems, modelled in continuous time, sampled time, or a hybrid of the two. It runs on the Matlab platform. For modelling, Simulink provides a graphical user interface (GUI) for building models as block diagrams.

Table 5.1 summarises the primary purpose of each analysed tool, along with the expected input language.

Despite the existence of numerous tools that work with hybrid systems, few are in the context that is pursued in our work. LUTESS, LURETTE, GATEL, LBTest, CTE XL, Autofocus and AutoLink are intended for low-level systems using logic gates. TorX is not capable of representing the system behavior via differential and difference equations. Ulysses is not publicly available. MuProD is a private initiative. 20-sim uses a graphical representation as input, making it difficult to use it in our approach. SpaceEx apparently is a good alternative to be adopted, but was only recently discovered by indication, which prevented a deep study aiming its utilisation. Differently, S-TaLiro seems to be closer related to this work, as it generates test cases for hybrid systems, however its test generation module is not capable of modifying the variables in a discrete manner while changing states.

Despite all the tool compatibility, the module for testing is still under construction and the hybrid automaton, used as input, does not fully support the features used in our strategy. These facts prevent the use of this tool to support our strategy.

However, when we turn our attention to the tool proposed by AERTS; MOUSAVI; RENIERS (2015), we note that it represents hybrid systems as models in the Acumen language, which can be generated from the intermediate notation of our work (hybrid data-flow reactive systems).

- MATLAB is the tool for design and development complex embedded systems most widely used in the industrial area.

Table 5.1: Tools analyzed

Tool	**Input language**	**Primary purpose**
LUTESS	LUSTRE	Test generation
LURETTE	LUSTRE, Lucky, Lutin	Test generation
GATEL	LUSTRE	Test generation and solving constraints
LBTest	Propositional linear temporal logic	Learning-based testing and model checker
KeYmaera	Differential dynamic logic	Conformance verification
CTE XL	Graphical Specifications	Classification-Tree method
RTCAT	Logical components	Test generation
MuProD	Gaussian process model	Test generation
Ulysses	UML models/QAS	Test Generation
20-sim	Graphical Specifications	Simulation
SpaceEx	SpaceEx language	Verification
TorX	LTS (LOTOS, PROMELA, FPS)	Conformance testing
Autofocus	Autofocus	Test generation and solving constraints
AutoLink	SDL	Simulation and validation
S-TaLiRo	MTL Specification, Simulink Model, Hybrid Automata	Simulation and test conformance
(AERTS; MOUSAVI; RENIERS, 2015)	Acumen Model, Matlab Model	Simulation and test conformance
Simulink	Graphical models	Simulation

- The model in Acumen can be generated from the DFRS (model used by our a tool) via a simple translation.

Considering these facts, our adaptation of the NAT2TEST strategy generates models in Acumen. Once the implementation model is already largely used in industry, our goal is to use this common implementation model along with a specification model written in Acumen generated by our strategy from the requirements, and, with these two models, to use as input for conformance verification processed by AERTS; MOUSAVI; RENIERS (2015).

5.2 Future work

Despite the results achieved in this work, we envisage some future work, described bellow.

Enable the user to specify the changing of the input variables Although our strategy automates the generation of Acumen specification models from natural-language requirements, it is still required to specify manually how the input variables evolve over time, since purely random values might not lead to relevant simulations. An interesting approach to solving this problem is used in (AUTOMATED TESTING WITH RT-TESTER - THEORETICAL ISSUES DRIVEN BY PRACTICAL NEEDS, 2000), where a GUI is available for the user to define how the change of values shall occur during the simulation.

Expand the proposed approach to generate test cases For such functionality it is needed to find a tool to give support to the generation of test cases, since this is not covered by the Acumen tool. A possible candidate is the S-TaLiRo tool, previously mentioned. It is capable of generating test cases from hybrid automata. However, since some restrictions apply to the classes of hybrid automata it can deal with, its adoption is not straightforward.

Elaborate a conformance testing approach Conformance testing is characterised by efficiently detecting faults or establishing a level of quality by generating test cases from a model (in our case, Acumen models), and applying the generated test cases to evaluate the behaviour of the system under test (BROY; JONSSON; KATOEN; LEUCKER; PRETSCHNER, 2005). Establishing a conformance relation testing, it can allow the strategy to prove the soundness of the test case generation, similar to what CARVALHO (2016) did, using CSP.

Perform more empirical analyses As showed in Chapter 4, the h-NAT2TEST strategy has been illustrated using two examples from different domains. Nevertheless, to investigate the effective application of the proposed strategy, more empirical analyses need to be conducted, considering bigger and more complex systems.

Prove the translation semantics As mentioned in Section 3.5, the translation from DFRS to Acumen is done with no semantic preservation proof. To establish and prove this semantic translation would be beneficial to the strategy robustness.

Improve the strategy to support nonlinear hybrid systems As mentioned in Chapter 1, hybrid systems have been gaining popularity every day and, likewise, its subclass called nonlinear, it plays an important role in modern mechatronics and robotics (BUSS; GLOCKER; HARDT; VON STRYK; BULIRSCH; SCHMIDT, 2002). For extension of the strategy, a more detailed study of nonlinear hybrid systems should be done in order to identify the necessary changes in the proposed strategy.

References

AALST, W. van der. **CPN A concrete language for high-level Petri nets**. Online; accessed 19 June 2016, http://cpntools.org/_media/book/cpn.pdf.

LEUCKER, M.; RUEDA, C.; VALENCIA, D. F. (Ed.). **Theoretical Aspects of Computing - ICTAC 2015**: 12th international colloquium, cali, colombia, october 29-31, 2015, proceedings. Cham: Springer International Publishing, 2015. p.563–572.

ALUR, R. et al. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: **Hybrid systems**. [S.l.]: Springer, 1993. p.209–229.

ALUR, R. et al. The algorithmic analysis of hybrid systems. **Theoretical Computer Science**, [S.l.], v.138, n.1, p.3 – 34, 1995.

DAHLWEID, M.; MEYER, O.; PELESKA, J. **Automated Testing with RT-Tester - Theoretical Issues Driven by Practical Needs**. [S.l.: s.n.], 2000.

BLACKBURN, M.; BUSSER, R. D.; FONTAINE, J. S. **Automatic generation of test vectors for SCR-style specifications**. [S.l.: s.n.], 1997. p.54–67.

BRANICKY, M. S. Introduction to hybrid systems. In: **Handbook of Networked and Embedded Control Systems**. [S.l.]: Springer, 2005. p.91–116.

BROOKES, S. D.; HOARE, C. A.; ROSCOE, A. W. A theory of communicating sequential processes. **Journal of the ACM (JACM)**, [S.l.], v.31, n.3, p.560–599, 1984.

BROY, M. et al. **Model-Based Testing of Reactive Systems**: advanced lectures (lecture notes in computer science). Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

BUSS, M. et al. Nonlinear hybrid dynamical systems: modeling, optimal control, and applications. In: **Modelling, Analysis, and Design of Hybrid Systems**. [S.l.]: Springer, 2002. p.311–335.

CARVALHO, G. **NAT2TEST**: generating test cases fromnatural language requirements based on csp. 2016. Tese (Doutorado em Ciência da Computação) — Universidade Federal de Pernambuco.

CARVALHO, G.; CAVALCANTI, A.; SAMPAIO, A. Modelling timed reactive systems from natural-language requirements. **Formal Aspects of Computing**, [S.l.], p.1–41, 2016.

CASSANDRAS, C. G.; LAFORTUNE, S. **Introduction to discrete event systems**. [S.l.]: Springer Science & Business Media, 2009.

CHEN, Y.-P. **Course**: dynamic system analysis and simulation. Online; accessed 29 June 2016, http://jsjk.cn.nctu.edu.tw/JSJK/DSAS/DSAS_1_Static%20and%20Dynamic%20Systems_2013.pdf.

COTTON, S.; FREHSE, G.; LEBELTEL, O. **The SpaceEx modeling language**. 2010.

RAYMOND, P.; JAHIER, E.; ROUX, Y. **Describing and executing random reactive systems**. [S.l.: s.n.], 2006. p.216–225.

DIEHL, S. **Hindley-Milner Inference**. Online; accessed 29 June 2016, `http://dev.stephendiehl.com/fun/006_hindley_milner.html`.

ERICH, G. et al. **Design Patterns**: elements of reusable object-oriented software. 1st.ed. Boca Raton, FL, USA: CRC Press, Inc., 1994.

FAHRENBERG, U.; LARSEN, K. G.; LEGAY, A. Model-based verification, optimization, synthesis and performance evaluation of real-time systems. In: **Unifying Theories of Programming and Formal Engineering Methods**. [S.l.]: Springer, 2013. p.67–108.

SCHLICK, R.; HERZNER, W.; JÖBSTL, E. **Fault-based generation of test cases from UML-models–approach and some experiences**. [S.l.: s.n.], 2011. p.270–283.

FILLMORE, C. **The case for case. Universals in linguistic theory, ed. by E. Bach & R. Harms, 1-88**. [S.l.]: New York: Holt, Rinehart & Winston, 1968.

SUNG, C.; KIM, T. G. **Framework for simulation of hybrid systems**: interoperation of discrete event and continuous simulators using hla/rti. [S.l.: s.n.], 2011. p.1–8.

GROCHTMANN, M.; GRIMM, K. Classification trees for partition testing. **Software Testing, Verification and Reliability**, [S.l.], v.3, n.2, p.63–82, 1993.

HALBWACHS, N. et al. The synchronous data flow programming language LUSTRE. **Proceedings of the IEEE**, [S.l.], v.79, n.9, p.1305–1320, 1991.

HEEREN, B. et al. **Generalizing Hindley-Milner Type Inference Algorithms**. [S.l.: s.n.], 2002.

HEITMEYER, C. L.; BHARADWAJ, R. Applying the SCR requirements method to the light control case study. **Journal of Universal Computer Science**, [S.l.], v.6, n.7, p.650–678, 2000.

HUBBARD, J. H.; WEST, B. H. **MacMath 9.2**. [S.l.]: Springer, 1993.

GOLLU, A.; VARAIYA, P. **Hybrid dynamical systems**. [S.l.: s.n.], 1989. p.2708–2712 vol.3.

KHALI, H. K. **Nonlinear Systems**. 3st.ed. Upper Saddle River, NJ, USA: PRENTICE HALL, Inc., 2002.

LARSEN, G. K.; STEFFEN, B.; WEISE, C. Continuous modeling of real-time and hybrid systems: from concepts to tools. **International Journal on Software Tools for Technology Transfer**, [S.l.], v.1, n.1, p.64–85, 1997.

MEINKE, K.; SINDHU, M. A. **LBTest**: a learning-based testing tool for reactive systems. [S.l.: s.n.], 2013. p.447–454.

LUNZE, J.; LAMNABHI-LAGARRIGUE, F. **Handbook of hybrid systems control**: theory, tools, applications. [S.l.]: Cambridge University Press, 2009.

MARRE, B.; ARNOULD, a. Test sequences generation from LUSTRE descriptions: gatel. **Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering**, [S.l.], 2000.

MILNER, R. A Theory of Type Polymorphism in Programming. **JOURNAL OF COMPUTER AND SYSTEM SCIENCES**, Edinburgh, Scotland, p.348–375, 1978.

CHANG, H. et al. **Mixed-signal simulation challenges and solutions**. [S.l.: s.n.], 2008. p.xiii–xiii.

TAN, L. et al. **Model-based testing and monitoring for hybrid embedded systems**. [S.l.: s.n.], 2004. p.487–492.

EFKEMANN, C.; PELESKA, J. **Model-Based Testing for the Second Generation of Integrated Modular Avionics**. [S.l.: s.n.], 2011. p.55–62.

DALAL, S. R. et al. **Model-based Testing in Practice**. New York, NY, USA: ACM, 1999. p.285–294. (ICSE '99).

REZA, H.; LANDE, S. **Model Based Testing Using Software Architecture**. [S.l.: s.n.], 2010. p.188–193.

MURATA, T. Petri nets: properties, analysis and applications. **Proceedings of the IEEE**, [S.l.], v.77, n.4, p.541–580, 1989.

NICOLLIN, X. et al. An approach to the description and analysis of hybrid systems. In: **Hybrid Systems**. [S.l.]: Springer, 1993. p.149–178.

PELESKA, J. et al. Automated model-based testing with RT-Tester. **University of Bremen**, [S.l.], 2011.

PRETSCHNER, a.; LOTZBEYER, H.; PHILIPPS, J. Model based testing in evolutionary software development. **Proceedings 12th International Workshop on Rapid System Prototyping. RSP 2001**, [S.l.], p.155–160, 2001.

PROBERT, R. L.; MONKEWICH, O. TTCN: the international notation for specifying tests of communications systems. **Computer Networks and ISDN Systems**, [S.l.], v.23, n.5, p.417–438, 1992.

BREITMAN, K.; CAVALCANTI, A. (Ed.). **Qualitative Action Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p.206–225.

RAYMOND, P.; ROUX, Y.; JAHIER, E. Lutin: a language for specifying and executing reactive scenarios. **EURASIP Journal on Embedded Systems**, [S.l.], v.2008, p.1–11, 2008.

ROBERTS, G. W. et al. **Microelectronic circuits. Spice for microelectronic circuits**. [S.l.]: Oxford University Press, 1992.

SAVKIN, A.; EVANS, R. A new approach to robust control of hybrid systems over infinite time. **Automatic Control, IEEE Transactions on**, [S.l.], v.43, n.9, p.1292–1296, Sep 1998.

SCHAFT, H. S. Arjan van der. **An Introdution to Hybrid Dynamecal Systems**. 1st.ed. Department of Systems, University of Twente, USA: Springer-Verlag London, 2000.

SCOTT, M. L. **Programming Language Pragmatics**. [S.l.]: Elsevier, 2005, 2005.

SHANNON, R. E. Systems Simulation: the art and science prentice-hall. **Englewood Cliffs, NJ**, [S.l.], 1975.

SKOGESTAD, S.; POSTLETHWAITE, I. **Multivariable Feedback Control**: analysis and design. [S.l.]: John Wiley & Sons, 2005.

FREHSE, G. et al. **SpaceEx**: scalable verification of hybrid systems. [S.l.: s.n.], 2011. p.379–395.

TAHA, W. **Acumen 2015 Reference Manual**. Online; accessed 29 June 2016, https://www.google.com/url?q=http://bit.ly/Acumen-manual-2015&sa=D&ust=1458598730176000&usg=AFQjCNGSJDcWhP5vJGxPWa7To5Vp2gqfYA.

TAHA, W. et al. A Core Language for Executable Models of Cyber Physical Systems. **Rice University**, [S.l.], 2010.

TAHA, W. et al. Acumen: an open-source testbed for cyber-physical systems research. **Proceedings of the IEEE**, [S.l.], v.77, n.4, p.541–580, 2015.

TAHA, W.; ZENG, Y.; DURACZ, A. **Acumen EBNF**. 2016.

SILVA, B. C. F.; CARVALHO, G.; SAMPAIO, A. **Test Case Generation from Natural Language Requirements Using CPN Simulation**. [S.l.: s.n.], 2015. p.178–193.

BOHNENKAMP, H.; BELINFANTE, A. **Timed Testing with TorX**. [S.l.: s.n.], 2006. p.1–10.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A Taxonomy of Model-based Testing Approaches. **Softw. Test. Verif. Reliab.**, Chichester, UK, v.22, n.5, p.297–312, Aug. 2012.

JAIRAM, S. et al. **Verification of a MEMS based adaptive cruise control system using simulation and semi-formal approaches**. [S.l.: s.n.], 2008. p.910–913.

BOER, F. S. de et al. (Ed.). **Verifying Object-Oriented Programs with KeY**: a tutorial. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p.70–101.

HOXHA, B.; MAVRIDIS, N.; FAINEKOS, G. **VISPEC**: a graphical tool for elicitation of mtl requirements. [S.l.: s.n.], 2015. p.3486–3492.

WIKIPEDIA. **20-sim 4.6**. 2016.

WOLFRAM, S. **Mathematica**: a system for doing mathematics by computer. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988.

Y.LI, P. **Course**: advanced control system design. Online; accessed 29 June 2016, http://www.me.umn.edu/courses/me8281/notes_S12/Chapter1_S12.pdf.

ZANDER, J.; SCHIEFERDECKER, I.; MOSTERMAN, P. J. **Model-Based Testing for Embedded Systems**. 1st.ed. Boca Raton, FL, USA: CRC Press, Inc., 2011.

ZANDER-NOWICKA, J. **Model-based testing of real-time embedded systems in the automotive domain**. [S.l.]: Fraunhofer-IRB-Verlag, 2008.

Appendix

A

SysReq-CNL for hybrid system requirements

Table A.1: SysReq-CNL – a grammar for hybrid system requirements

Sentence	→	**Requirement** | **FunctionDeclaration;**
Requirement	→	ConditionalClause COMMA ActionClause PERIOD;
ConditionalClause	→	CONJ AndCondition;
AndCondition	→	AndCondition COMMA AND OrCondition
		| OrCondition;
OrCondition	→	OrCondition OR Condition | Condition;
Condition	→	VariableState VerbPhraseCondition?;
ActionClause	→	VariableState VerbPhraseAction;
VerbPhraseAction	→	SHALL (VerbAction VerbComplement
		| COLON VerbAction VerbComplement
		(COMMA VerbAction VerbComplement)+);
VerbAction	→	VBASE;
VerbPhraseCondition	→	VerbCondition NOT? ComparativeTerm?
		VerbComplement;
VerbCondition	→	VTOBE_PRE3 | VPRE3RD
		| VTOBE_PRE | VTOBE_PAST3;
ComparativeTerm	→	(COMP (OR NOT? COMP)?);
VerbComplement	→	VariableState? PrepositionalPhrase*;
VariableState	→	**Expression**;
PrepositionalPhrase	→	PREP VariableState;
NounPhrase	→	DETER? ADJ* Noun+;
Noun	→	NSING | NPLUR;
Expression	→	AndExpression (OR AndExpression)*;
AndExpression	→	NotExpression (AND NotExpression)*;

NotExpression → LOGICALNOT NotExpression | ComparativeExpression;
ComparativeExpression → ArithmeticExpression
(ComparativeOperator ArithmeticExpression)*;
ComparativeOperator → GT | LT | GE | LE | EQ | NE ;
ArithmeticExpression → Term (AdditiveOperator Term)*;
Term → Factor (MultiplicativeOperator Factor)*;
Factor → PrefixOperator? PrimaryExpression;
PrefixOperator → AdditiveOperator;
AdditiveOperator → PLUS | MINUS;
MultiplicativeOperator → MULT | SLASH | MOD;
PrimaryExpression → FunctionID LP ArgumentList? RP
| TRUE | FALSE | NUMBER | NounPhrase
| LP Expression RP;
FunctionID → SIN | COS | EXP | LOG | SQRT | Noun;
ArgumentList → VariableState (COMMA VariableState)*;

FunctionDeclaration → Noun LP ParameterList? RP EQUALSSIGN FunctionBody;
ParameterList → Noun (COMMA Noun)*;
FunctionBody → TernaryExpression | PatternMatching | Expression;
TernaryExpression → Expression IN TernaryDefinition COLON
TernaryDefinition;
TernaryDefinition → Expression | TernaryExpession;
PatternMatching → (Expression DO Expression)+;

B

Acumen EBNF

```
prog       = model { model }
model      = "model" modelname
             "(" paramlst? ")" "="
             [ "initially" initlst? [ "always" stmtlst? ] ]

modelname  = name
paramlst   = var  { "," var }
initlst    = discrass { "," discrass }
stmtlst    = stmt { "," stmt }

stmt       = claim | discrass | contass | claim | if | match
discrass   = (var | dot) "+ =" (expr | new)
contass    = (var | dot) "="   (expr | new)
claim      = "claim" [ string ] pred
new        = "create" modelname [ "(" exprlst? ")" ]
if         = "if" pred "then" stmtlst
             ( "else" ( stmt | "(" stmtlst ")" )
             | "noelse" )
match      = "match" name "with" "[" clause { "|" clause } "]"
clause     = lit "->" stmtlst?

pred       = "true" | "false" | rel
           | "(" pred ")"
           | unbop pred
           | pred binbop pred
rel        = aexpr relop aexpr
relop      = "<" | ">" | "<=" | ">=" | "==" | "~="
unbop      = "not"
binbop     = "&&" | "||"

lit        = modelname | interval | number | string | const
vector     = "(" exprlst? ")"
const      = "red" | "green" | "blue" | "pi"
let        = "let" initlst "in" expr

expr       = let | vector | aexpr | lit
exprlst    = expr { "," expr }

```

```
39 aexpr      = interval | number | dot | var
40            | "(" aexpr ")"
41            | unaop aexpr
42            | aexpr binaop aexpr
43            | fun "(" aexprlst ")"
44 aexprlst   = aexpr { "," aexpr }
45 dot        = name "." var [ "(" int ")" ]
46 var        = name (#\\*)
47 name       = #[a-zA-Z][a-zA-Z0-9]*
48 number     = float | int
49 int        = #[0-9]+
50 float      = #[0-9]*\\.[0-9]*
51 interval   = "[" number [ .. number ] "]"
52            | number +/- number
53 string     = #\"[a-zA-Z0-9]*\"
54 unaop      = -
55 binaop     = + | - | * | /
56 fun        = sin | cos | tan | atan | atan2 | abs
57            | dot | floor | ceil
```

C

The DFRS representation in XML

C.1 The DFRS of the DC-DC Boost-Converter

```
<?xml version="1.0"?>
<DFRS>

<InVariable>
<VarName>s</VarName>
<VarType>INTEGER</VarType>
<ExpectedValue>0</ExpectedValue>
<ExpectedValue>1</ExpectedValue>
<InitialValue>0</InitialValue>
</InVariable>

<InVariable>
<VarName>I</VarName>
<VarType>FLOAT</VarType>
<ExpectedValue>0.000080</ExpectedValue>
<InitialValue>0.000080</InitialValue>
</InVariable>

<OutVariable>
<VarName>phi</VarName>
<VarType>INTEGER</VarType>
<ExpectedValue>0</ExpectedValue>
<InitialValue>0</InitialValue>
</OutVariable>

<OutVariable>
<VarName>q</VarName>
<VarType>FLOAT</VarType>
<ExpectedValue>0.0</ExpectedValue>
<InitialValue>0.0</InitialValue>
</OutVariable>

<OutVariable>
<VarName>the_q_derivative</VarName>
```

```
35 <VarType>FLOAT</VarType>
36 <ExpectedValue>0.0</ExpectedValue>
37 <ExpectedValue>q_mode1(phi,l,r,c,q)</ExpectedValue>
38 <ExpectedValue>q_mode2(r,c,q)</ExpectedValue>
39 <ExpectedValue>q_mode3(r,c,q)</ExpectedValue>
40 <ExpectedValue>q_mode4()</ExpectedValue>
41 <InitialValue>0.0</InitialValue>
42 </OutVariable>
43
44 <OutVariable>
45 <VarName>r</VarName>
46 <VarType>FLOAT</VarType>
47 <ExpectedValue>0.0</ExpectedValue>
48 <ExpectedValue>q_mode1(phi,l,r,c,q)</ExpectedValue>
49 <ExpectedValue>q_mode2(r,c,q)</ExpectedValue>
50 <ExpectedValue>q_mode3(r,c,q)</ExpectedValue>
51 <InitialValue>0.0</InitialValue>
52 </OutVariable>
53
54 <OutVariable>
55 <VarName>c</VarName>
56 <VarType>FLOAT</VarType>
57 <InitialValue>0.0</InitialValue>
58 </OutVariable>
59
60 <OutVariable>
61 <VarName>the_phi_derivative</VarName>
62 <VarType>FLOAT</VarType>
63 <ExpectedValue>0.0</ExpectedValue>
64 <ExpectedValue>phi_mode1(c,q,e)</ExpectedValue>
65 <ExpectedValue>phi_mode2(e)</ExpectedValue>
66 <ExpectedValue>phi_mode3()</ExpectedValue>
67 <ExpectedValue>phi_mode4(e)</ExpectedValue>
68 <InitialValue>0.0</InitialValue>
69 </OutVariable>
70
71 <OutVariable>
72 <VarName>the_system_mode</VarName>
73 <VarType>INTEGER</VarType>
74 <ExpectedValue>1</ExpectedValue>
75 <ExpectedValue>2</ExpectedValue>
76 <ExpectedValue>3</ExpectedValue>
77 <ExpectedValue>4</ExpectedValue>
78 <InitialValue>1</InitialValue>
79 </OutVariable>
80
81 <OutVariable>
82 <VarName>e</VarName>
83 <VarType>FLOAT</VarType>
84 <ExpectedValue>0.0</ExpectedValue>
85 <InitialValue>0.0</InitialValue>
86 </OutVariable>
87
88 <Function>
```

```
<FuncName>the_dc-dc_boost_control_system</FuncName>

<FuncAttrib>
<GuardName>REQ001BOOST</GuardName>
<StaticGuard>((s)==1)&amp;amp;amp;amp;((q)&gt;0) = true
AND the_system_mode = 1</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>2</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ006BOOST</GuardName>
<StaticGuard>((q)&lt;0) = true
AND the_system_mode = 2</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>4</Expression></Statement>
<Statement>
<VarName>q</VarName>
<Expression>0</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ0015BOOST</GuardName>
<StaticGuard>the_system_mode = 3</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_q_derivative</VarName>
<Expression>q_mode3(r,c,q)</Expression></Statement>
<Statement>
<VarName>the_phi_derivative</VarName>
<Expression>phi_mode3()</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ009BOOST</GuardName>
<StaticGuard>((s)==1)&amp;amp;amp;amp;((q)&lt;=0) = true
AND the_system_mode = 3</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>4</Expression></Statement>
<Statement>
<VarName>q</VarName>
<Expression>0</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ004BOOST</GuardName>
<StaticGuard>((s)==0) = ((phi)&gt;=0)
AND the_system_mode = 2</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>1</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ012BOOST</GuardName>
<StaticGuard>((q)&lt;0) = true
AND the_system_mode = 4</StaticGuard>
```

```
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>q</VarName>
<Expression>0</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ007BOOST</GuardName>
<StaticGuard>((q)&lt;=((c)*e)) = true
AND the_system_mode = 3</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>1</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ0014BOOST</GuardName>
<StaticGuard>the_system_mode = 2</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_q_derivative</VarName>
<Expression>q_mode2(r,c,q)</Expression></Statement>
<Statement>
<VarName>the_phi_derivative</VarName>
<Expression>phi_mode2(e)</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ003BOOST</GuardName>
<StaticGuard>((s)==1)&amp;amp;amp;((q)&lt;=0) = true
AND the_system_mode = 1</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>4</Expression></Statement>
<Statement>
<VarName>q</VarName>
<Expression>0</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ011BOOST</GuardName>
<StaticGuard>((s)==0)&amp;amp;amp;((phi)&lt;=0) = true
AND the_system_mode = 4</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>3</Expression></Statement>
<Statement>
<VarName>phi</VarName>
<Expression>0</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ013BOOST</GuardName>
<StaticGuard>the_system_mode = 1</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_q_derivative</VarName>
<Expression>q_mode1(phi,l,r,c,q)</Expression></Statement>
<Statement>
<VarName>the_phi_derivative</VarName>
<Expression>phi_mode1(c,q,e)</Expression></Statement></FuncAttrib>
```

```
<FuncAttrib>
<GuardName>REQ008BOOST</GuardName>
<StaticGuard>((s)==1)&amp;amp;&amp;amp;((q)&gt;=0) = true
AND the_system_mode = 3</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>2</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ016BOOST</GuardName>
<StaticGuard>the_system_mode = 4</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_q_derivative</VarName>
<Expression>q_mode4()</Expression></Statement>
<Statement>
<VarName>the_phi_derivative</VarName>
<Expression>phi_mode4(e)</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ010BOOST</GuardName>
<StaticGuard>((s)==0)&amp;amp;&amp;amp;((phi)&gt;=0) = true
AND the_system_mode = 4</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>1</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ002BOOST</GuardName>
<StaticGuard>((phi)&lt;=0)&amp;amp;&amp;amp;((q)&gt;((c)*e)) = true
AND the_system_mode = 1</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>3</Expression></Statement>
<Statement>
<VarName>phi</VarName>
<Expression>0</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ005BOOST</GuardName>
<StaticGuard>((s)==0)&amp;amp;&amp;amp;((phi)&lt;=0) = true
AND the_system_mode = 2</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>3</Expression></Statement>
<Statement>
<VarName>phi</VarName>
<Expression>0</Expression></Statement></FuncAttrib></Function>

<FunctionDefinition>
<FunctionName>q_mode1</FunctionName>
<FunctionBody>((((phi)/l))-((((1)/((r)*c)))*q))</FunctionBody>
<Params><Param>phi</Param><Param>l</Param><Param>r</Param><Param>c</Param><
    Param>q</Param></Params>
```

```
</FunctionDefinition>

<FunctionDefinition>
<FunctionName>q_mode2</FunctionName>
<FunctionBody>((((-1)/((r)*c)))*q)</FunctionBody>
<Params><Param>r</Param><Param>c</Param><Param>q</Param></Params>
</FunctionDefinition>

<FunctionDefinition>
<FunctionName>q_mode3</FunctionName>
<FunctionBody>((-q)/((r)*c))</FunctionBody>
<Params><Param>r</Param><Param>c</Param><Param>q</Param></Params>
</FunctionDefinition>

<FunctionDefinition>
<FunctionName>q_mode4</FunctionName>
<FunctionBody>0</FunctionBody>
<Params></Params>
</FunctionDefinition>

<FunctionDefinition>
<FunctionName>phi_mode2</FunctionName>
<FunctionBody>e</FunctionBody>
<Params><Param>e</Param></Params>
</FunctionDefinition>

<FunctionDefinition>
<FunctionName>phi_mode1</FunctionName>
<FunctionBody>((((((-1)/c))*q))+e)</FunctionBody>
<Params><Param>c</Param><Param>q</Param><Param>e</Param></Params>
</FunctionDefinition>

<FunctionDefinition>
<FunctionName>phi_mode4</FunctionName>
<FunctionBody>e</FunctionBody>
<Params><Param>e</Param></Params>
</FunctionDefinition>

<FunctionDefinition>
<FunctionName>phi_mode3</FunctionName>
<FunctionBody>0</FunctionBody>
<Params></Params>
</FunctionDefinition>

</DFRS>
```

C.2 The DFRS of the adaptive cruise control

```
<?xml version="1.0"?>
<DFRS>
```

```
3
4  <InVariable>
5  <VarName>acceleration</VarName>
6  <VarType>INTEGER</VarType>
7  <InitialValue>0</InitialValue>
8  </InVariable>
9
10 <InVariable>
11 <VarName>halt</VarName>
12 <VarType>BOOLEAN</VarType>
13 <ExpectedValue>false</ExpectedValue>
14 <ExpectedValue>true</ExpectedValue>
15 <InitialValue>false</InitialValue>
16 </InVariable>
17
18 <InVariable>
19 <VarName>cruise_speed</VarName>
20 <VarType>INTEGER</VarType>
21 <ExpectedValue>((velocity)&gt;=cruise_speed)</ExpectedValue>
22 <ExpectedValue>cruise_speed</ExpectedValue>
23 <InitialValue>0</InitialValue>
24 </InVariable>
25
26 <InVariable>
27 <VarName>the_cruise_proximity</VarName>
28 <VarType>INTEGER</VarType>
29 <ExpectedValue>((the_vehicle_proximity)&lt;the_cruise_proximity)</
       ExpectedValue>
30 <ExpectedValue>((the_vehicle_proximity)&gt;the_cruise_proximity)</
       ExpectedValue>
31 <InitialValue>0</InitialValue>
32 </InVariable>
33
34 <InVariable>
35 <VarName>cruise</VarName>
36 <VarType>BOOLEAN</VarType>
37 <ExpectedValue>false</ExpectedValue>
38 <ExpectedValue>true</ExpectedValue>
39 <InitialValue>false</InitialValue>
40 </InVariable>
41
42 <InVariable>
43 <VarName>the_halt_proximity</VarName>
44 <VarType>INTEGER</VarType>
45 <ExpectedValue>((the_vehicle_proximity)&lt;the_halt_proximity)</ExpectedValue>
46 <ExpectedValue>((the_vehicle_proximity)&gt;the_halt_proximity)</ExpectedValue>
47 <InitialValue>0</InitialValue>
48 </InVariable>
49
50 <InVariable>
51 <VarName>retardation</VarName>
52 <VarType>BOOLEAN</VarType>
53 <ExpectedValue>false</ExpectedValue>
54 <ExpectedValue>true</ExpectedValue>
```

```
<InitialValue>false</InitialValue>
</InVariable>

<InVariable>
<VarName>the_vehicle_proximity</VarName>
<VarType>NUMBER</VarType>
<ExpectedValue>false</ExpectedValue>
<ExpectedValue>true</ExpectedValue>
<InitialValue>false</InitialValue>
</InVariable>

<OutVariable>
<VarName>the_system_mode</VarName>
<VarType>INTEGER</VarType>
<ExpectedValue>accelarate</ExpectedValue>
<ExpectedValue>cruise</ExpectedValue>
<ExpectedValue>halt</ExpectedValue>
<ExpectedValue>retard</ExpectedValue>
<InitialValue>0</InitialValue>
</OutVariable>

<OutVariable>
<VarName>velocity</VarName>
<VarType>INTEGER</VarType>
<ExpectedValue>((velocity)&gt;=cruise_speed)</ExpectedValue>
<ExpectedValue>0</ExpectedValue>
<ExpectedValue>cruise_speed</ExpectedValue>
<InitialValue>0</InitialValue>
</OutVariable>

<OutVariable>
<VarName>accelarate</VarName>
<VarType>BOOLEAN</VarType>
<ExpectedValue>false</ExpectedValue>
<ExpectedValue>true</ExpectedValue>
<InitialValue>false</InitialValue>
</OutVariable>

<OutVariable>
<VarName>velocity_derivative</VarName>
<VarType>INTEGER</VarType>
<ExpectedValue>acceleration</ExpectedValue>
<ExpectedValue>retardation</ExpectedValue>
<InitialValue>0</InitialValue>
</OutVariable>

<OutVariable>
<VarName>retard</VarName>
<VarType>BOOLEAN</VarType>
<ExpectedValue>false</ExpectedValue>
<ExpectedValue>true</ExpectedValue>
<InitialValue>false</InitialValue>
</OutVariable>

```

```
<Function>
<FuncName>the_acc</FuncName>

<FuncAttrib>
<GuardName>REQ008</GuardName>
<StaticGuard>the_system_mode = accelarate
AND ((velocity)&gt;=cruise_speed) = true
OR ((the_vehicle_proximity)&lt;the_cruise_proximity) = true</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>retard</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ007</GuardName>
<StaticGuard>the_system_mode = retard
AND ((the_vehicle_proximity)&gt;the_cruise_proximity) = true</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>accelarate</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ001</GuardName>
<StaticGuard>the_system_mode = halt</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>velocity</VarName>
<Expression>0</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ006</GuardName>
<StaticGuard>the_system_mode = halt
AND ((the_vehicle_proximity)&gt;the_halt_proximity) = true</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>accelarate</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ010</GuardName>
<StaticGuard>the_system_mode = cruise
AND ((the_vehicle_proximity)&lt;the_cruise_proximity) = true</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>retard</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ002</GuardName>
<StaticGuard>the_system_mode = cruise</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>velocity</VarName>
<Expression>cruise_speed</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ009</GuardName>
<StaticGuard>the_system_mode = accelarate
AND ((velocity)&gt;=cruise_speed) = true
```

```
AND ((the_vehicle_proximity)&gt;the_cruise_proximity) = true</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>cruise</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ003</GuardName>
<StaticGuard>the_system_mode = accelarate</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>velocity_derivative</VarName>
<Expression>acceleration</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ004</GuardName>
<StaticGuard>the_system_mode = retard</StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>velocity_derivative</VarName>
<Expression>retardation</Expression></Statement></FuncAttrib>
<FuncAttrib>
<GuardName>REQ005</GuardName>
<StaticGuard>((the_vehicle_proximity)&lt;the_halt_proximity) = true</
    StaticGuard>
<TimedGuard>null</TimedGuard>
<Statement>
<VarName>the_system_mode</VarName>
<Expression>halt</Expression></Statement></FuncAttrib></Function>

</DFRS>
```

D

Acumen representation of DFRS models

D.1 The DFRS of the DC-DC Boost-Converter

```
model Main(simulator) =
initially
a = create DFRS()
always
simulator.timeStep+=0.0001,simulator.endTime+=0.01
, simulator.method+="EulerForward"
model DFRS() =
initially
s=0,l=0.000080,phi=0,q=0.0,q'=0.0,r=20.0,c=0.000040,phi'=0.0,the_system_mode
    =1,e=12.0,gc=0.0,gc'=0.0
always
gc'=1.0,
if sin(1000000*gc) >= 0 then
   s += 1
  else
   s += 0,

 if(((s)==1)&&((q)>0) == true && the_system_mode == 1)then
the_system_mode += 2 noelse ,
 if(((q)<0) == true && the_system_mode == 2)then
the_system_mode += 4,q += 0 noelse ,
 if(the_system_mode == 3)then
q'=((-q)/((r)*c)),phi'=0 noelse ,
 if(((s)==1)&&((q)<=0) == true && the_system_mode == 3)then
the_system_mode += 4,q += 0 noelse ,
 if(((s)==0) && the_system_mode == 2)then
the_system_mode += 1 noelse ,
 if(((q)<0) == true && the_system_mode == 4)then
q += 0 noelse ,
 if(((q)<=((c)*e)) == true && the_system_mode == 3)then
the_system_mode += 1 noelse ,
 if(the_system_mode == 2)then
q'=((((-1)/((r)*c)))*q),phi'=e noelse ,
 if(((s)==1)&&((q)<=0) == true && the_system_mode == 1)then
```

```
34 the_system_mode += 4,q += 0 noelse ,
35  if(((s)==0)&&((phi)<0) == true && the_system_mode == 4)then
36 the_system_mode += 3,phi += 0 noelse ,
37  if(the_system_mode == 1)then
38 q'=((((phi)/l))-((((1)/((r)*c)))*q)),phi'=((((((-1)/c))*q))+e) noelse ,
39  if(((s)==1)&&((q)>=0) == true && the_system_mode == 3)then
40 the_system_mode += 2 noelse ,
41  if(the_system_mode == 4)then
42 q'=0,phi'=e noelse ,
43  if(((s)==0)&&((phi)>=0) == true && the_system_mode == 4)then
44 the_system_mode += 1 noelse ,
45  if(((phi)<=0)&&((q)>((c)*e)) == true && the_system_mode == 1)then
46 the_system_mode += 3,phi += 0 noelse ,
47  if(((s)==0)&&((phi)<=0) == true && the_system_mode == 2)then
48 the_system_mode += 3,phi += 0 noelse
```

D.2 The DFRS of the adaptive cruise control

```
1  model Main(simulator) =
2  initially
3  a = create DFRS()
4  always
5  simulator.timeStep+=0.001,simulator.endTime+=0.5
6  , simulator.method+="EulerForward"
7  model DFRS() =
8  initially
9  acceleration=100,cruise_speed=50,the_cruise_proximity=15,the_halt_proximity=5,
       retardation=-5,the_vehicle_proximity=5,the_system_mode="accelarate",
       velocity=0,velocity'=0,gc=0.0,gc'=0.0
10 ,the_vehicle_proximity'=0.0
11 always
12 gc'=1.0,
13
14  if(the_system_mode == "accelarate" && (((velocity)>cruise_speed) == true ||
       (((the_vehicle_proximity)<the_cruise_proximity) == true)))then
15 the_system_mode += "retard" noelse ,
16  if(the_system_mode == "retard" && ((the_vehicle_proximity)>
       the_cruise_proximity) == true)then
17 the_system_mode += "accelarate" noelse ,
18  if(the_system_mode == "halt")then
19 velocity += 0 noelse ,
20  if(the_system_mode == "halt" && ((the_vehicle_proximity)>the_halt_proximity)
       == true)then
21 the_system_mode += "accelarate" noelse ,
22  if(the_system_mode == "cruise" && ((the_vehicle_proximity)<
       the_cruise_proximity) == true)then
23 the_system_mode += "retard" noelse ,
24  if(the_system_mode == "cruise")then
25 velocity += cruise_speed noelse ,
```

```
26  if(the_system_mode == "accelarate" && (((velocity)==cruise_speed) == true &&
        (((the_vehicle_proximity)>=the_cruise_proximity) == true)))then
27 the_system_mode += "cruise" noelse ,
28  if(the_system_mode == "accelarate")then
29 velocity'=acceleration noelse ,
30  if(the_system_mode == "retard")then
31 velocity'=retardation noelse ,
32  if(((the_vehicle_proximity)<the_halt_proximity) == true && the_system_mode ==
        "retard")then
33 the_system_mode += "halt" noelse,
34
35
36
37
38 if the_vehicle_proximity < the_cruise_proximity then
39    the_vehicle_proximity' = acceleration
40   else
41    the_vehicle_proximity' = retardation
```