NAT2TEST: Generating Test Cases from Natural Language Requirements based on CSP

By

Gustavo Henrique Porto de Carvalho

PhD Thesis

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

RECIFE/2016

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GUSTAVO HENRIQUE PORTO DE CARVALHO

NAT2TEST: GENERATING TEST CASES FROM NATURAL LANGUAGE REQUIREMENTS BASED ON CSP

PHD THESIS PRESENTED TO THE CENTRO DE INFORMÁTICA OF UNIVERSIDADE FEDERAL DE PERNAMBUCO IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR IN COMPUTER SCIENCE.

SUPERVISOR: AUGUSTO SAMPAIO (UFPE, BRAZIL)
CO-SUPERVISOR: ANA CAVALCANTI (U. OF YORK , UK)

RECIFE/2016

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

C331n Carvalho, Gustavo Henrique Porto de
NAT2TEST: generating test cases from natural language requirements based on CSP / Gustavo Henrique Porto de Carvalho. – 2016.
248 f.: il., fig., tab.

Orientador: Augusto Cezar Alves Sampaio.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.

Inclui referências e apêndices.

1. Engenharia de software. 2. Métodos formais. 3. Verificação de sistemas. I. Sampaio, Augusto Cezar Alves (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2016-069

Gustavo Henrique Porto de Carvalho

NAT2TEST: Generating Test Cases from Natural Language Requirements based on CSP

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutora em Ciência da Computação

Aprovado em: 26/02/2016.

Acknowledgements

The results presented here were obtained with the support of many people and, thus, I would like to give them all my deepest thanks. I would like to thank the *Centro de Informática* at *Universidade Federal de Pernambuco* for supporting this work by providing an excellent study and research environment. Professor Flávia Barros for her essential help on natural-language processing techniques, and Professor Alexandre Mota for discussing the formal aspects of our research. The discussions and advices of other members of the Software Reliability Group were very important, as well as, the reviews provided by anonymous referees of the submitted papers that significantly contributed to improve this work.

I would like to thank Embraer, for providing us with challenging research problems, as well as examples from the aerospace domain. Professor Jan Peleska, for providing us with an interesting case from Mercedes-Benz (Daimler). Thanks to the financial support provided by FACEPE, CAPES, CNPq, and the European Commission via the Seventh Framework Programme.

Special and deep thanks to my supervisors, Professor Augusto Sampaio and Professor Ana Cavalcanti, for their professional and personal attention, always available for listening and helping, for the enlightening opportunities they afforded me. I have no proper words to thank them for their support.

I would also like to thank all my friends and my family for sharing the difficulties and the joys. Maria do Rosário Bezerra (my mother-in-law) and Laís Alves (my sister-in-law) for the nice moments. Luiz Carvalho Neto (my brother) for the delightful violin lessons. Luiz Carvalho Junior (my father) and Ione Carvalho (my mother) for their unconditional support and love. Ana Paula Carvalho (my eternal girlfriend, wife and my best friend) for her understanding, support, friendship, love, cherish, and for our baby boys (Lucas and Pedro). Last but not least, to God.

Resumo

Testes baseados em modelos (MBT) consiste em criar modelos para especificar o comportamento esperado de sistemas e, a partir destes, gerar testes que verificam se implementações possuem o nível de confiabilidade esperado. No contexto de sistemas críticos, estes modelos são normalmente (semi)formais e deseja-se uma definição precisa das condições necessárias para garantir que uma implementação é correta em relação ao modelo da especificação. Esta definição caracteriza uma relação de conformidade, que pode ser usada para provar que uma estratégia de MBT é consistente (*sound*). Apesar dos benefícios, aqueles sem familiaridade com a sintaxe e a semântica dos modelos empregados podem relutar em adotar estes formalismos.

Aqui, propõe-se uma estratégia de MBT para gerar casos de teste a partir de linguagem natural controlada (CNL). Esta estratégia (NAT2TEST) dispensa a necessidade de conhecer a sintaxe e a semântica das notações formais utilizadas internamente, uma vez que os modelos intermediários são gerados automaticamente a partir de requisitos em linguagem natural. Esta estratégia é apropriada para sistemas reativos baseados em fluxos de dados: uma classe de sistemas embarcados cujas entradas e saídas estão sempre disponíveis como sinais. Estes sistemas também podem ter comportamento dependente do tempo (discreto ou contínuo).

Na estratégia NAT2TEST, inicialmente, os requisitos são analisados sintaticamente de acordo com a CNL proposta neste trabalho para descrever sistemas reativos. Em seguida, a semântica informal dos requisitos é caracterizada utilizando a teoria de gramática de casos. Posteriormente, deriva-se uma representação formal dos requisitos considerando um modelo definido neste trabalho para sistemas reativos. Finalmente, este modelo é traduzido em uma especificação em *communicating sequential processes* (CSP) para permitir a geração de testes. Este trabalho prova que a estratégia de testes proposta é consistente considerando a relação de conformidade temporal baseada em entradas e saídas também definida aqui: csptio. Além de CSP, foi explorada a geração de outras notações formais (SCR e IMR), a partir das quais é possível gerar casos de teste usando ferramentas comerciais (T-VEC e RT-Tester, respectivamente). Todo o processo é automatizado pela ferramenta NAT2TEST.

A estratégia NAT2TEST foi avaliada considerando exemplos da literatura, da indústria aeroespacial (Embraer) e da automotiva (Mercedes). Foram analisados o desempenho e a capacidade de detectar defeitos gerados através de operadores de mutação. Em geral, a estratégia NAT2TEST apresentou melhores resultados do que a referência adotada: testes aleatórios. A estratégia NAT2TEST também foi comparada com ferramentas comerciais relevantes.

Palavras-chave: Testes baseados em modelos. Linguagem natural controlada. Gramática de casos. Sistemas reativos baseados em fluxos de dados. Processos sequenciais comunicantes. Relação de conformidade temporal baseada em entradas e saídas.

Abstract

High trustworthiness levels are usually required when developing critical systems, and model-based testing (MBT) techniques play an important role generating test cases from specification models. Concerning critical systems, these models are usually created using formal or semi-formal notations. Moreover, it is also desired to clearly and formally state the conditions necessary to guarantee that an implementation is correct with respect to its specification by means of a conformance relation, which can be used to prove that the test generation strategy is sound. Despite the benefits of MBT, those who are not familiar with the models syntax and semantics may be reluctant to adopt these formalisms. Furthermore, most of these models are not available in the very beginning of the project, when usually natural-language requirements are available. Therefore, the use of MBT is postponed.

Here, we propose an MBT strategy for generating test cases from controlled natural-language (CNL) requirements: NAT2TEST, which refrains the user from knowing the syntax and semantics of the underlying notations, besides allowing early use of MBT via natural-language processing techniques; the formal and semi-formal models internally used by our strategy are automatically generated from the natural-language requirements. Our approach is tailored to data-flow reactive systems: a class of embedded systems whose inputs and outputs are always available as signals. These systems can also have timed-based behaviour, which may be discrete or continuous.

The NAT2TEST strategy comprises a number of phases. Initially, the requirements are syntactically analysed according to a CNL we proposed to describe data-flow reactive systems. Then, the requirements informal semantics are characterised based on the case grammar theory. Afterwards, we derive a formal representation of the requirements considering a model of data-flow reactive systems we defined. Finally, this formal model is translated into communicating sequential processes (CSP) to provide means for generating test cases. We prove that our test generation strategy is sound with respect to our timed input-output conformance relation based on CSP: csptio. Besides CSP, we explore the generation of other target notations (SCR and IMR) from which we can generate test cases using commercial tools (T-VEC and RT-Tester, respectively). The whole process is fully automated by the NAT2TEST tool.

Our strategy was evaluated considering examples from the literature, the aerospace (Embraer) and the automotive (Mercedes) industry. We analysed performance and the ability to detect defects generated via mutation. In general, our strategy outperformed the considered baseline: random testing. We also compared our strategy with relevant commercial tools.

Keywords: Model-based testing. Controlled natural language. Case grammar. Data-flow reactive system. Communicating sequential processes. Timed input-output conformance relation.

List of Figures

List of Tables

List of Acronyms

ACT	Action	18
AGT	Agent	18
AIAA	American Institute of Aeronautics and Astronautics	22
BO	Business Object	132
BNF	Backus-Naur Form	133
CF	Case Frame	30
CAC	Condition Action	31
CFG	Context Free Grammar	27
CFV	Condition From Value	31
CMD	Condition Modifier	31
CNF	Conjunctive Normal Form	27
CNL	Controlled Natural Language	16
CPN	Coloured Petri Net	151
CPT	Condition Patient	31
CSP	Communicating Sequential Processes	15
CTV	Condition To Value	31
DAO	Data Access Object	132
DFRS	Data-Flow Reactive System	17
e-DFRS	Expanded Data-Flow Reactive System	38
EBNF	Extended Backus-Naur Form	27
FDR	Failures-Divergences Refinement	19
GLR	Generalized LR	133
GUI	Graphical User Interface	21
IOLTS	Input-Output Labelled Transition System	75
IMR	Intermediate Model Representation	19
ITML	IMA Test Modelling Language	14
LALR	Look-Ahead LR	133
LOC	Lines of Code	148

LTL	Linear Temporal Logic 154
LTS	Labelled Transition System 15
MBT	Model-Based Testing 14
MC/DC	Modified Condition/Decision Coverage 152
NAT2TEST	NATural Language Requirements to TEST Cases 17
NL	Natural Language 154
NLP	Natural Language Processing 15
NPP	Nuclear Power Plant 21
PAT	Patient 18
PC	Priority Control 21
POS	Parts of Speech 26
SCR	Software Cost Reduction 14
SMT	Satisfiability Modulo Theories 20
SUT	System Under Test 14
SysReq-CNL	System Requirements Controlled Natural Language 17
RF	Requirement Frame 31
s-DFRS	Symbolic Data-Flow Reactive System 38
TE	Test Environment 142
TIS	Turn Indicator System 21
TIOTS	Timed Input-Output Transition System 22
TOV	To Value 18
TR	Thematic Role 30
UC	Use Case 156
UML	Unified Modeling Language 14
VM	Vending Machine 17
VGS	Vector Generation System 142

Contents

1 Introduction

During the last fifty years, there has been a significant increase of embedded HW-SW components in critical systems. A report from NASA (WEST, 2009) highlights that, from 1960 to 2000, the amount of functionalities provided to military aircrafts by embedded software has grown from 8% to 80%. This scenario is not restricted to the aeronautical industry. The automotive industry, for instance, has become even more dependent on embedded components. In 2009 some cars already consisted of about 100 million lines of code (CHARETTE, 2009).

Clearly, this trend increases software size and complexity, and strongly impacts embedded critical systems safety and reliability. Currently, many researches are focusing on how to achieve the trustworthiness levels required for these systems. To this end, Model-Based Testing (MBT) techniques play an important role generating test cases from specification models.

1.1 Model-based testing

One of the goals of MBT is to make the testing process more agile, less susceptible to errors, and less dependent on human interaction. This goal is usually reached by means of automatic generation (and execution) of test cases, besides automatic generation of test data, from specification models.

As such, the quality of these specification models is crucial for an effective testing campaign. Thus, it is desirable to describe the expected system behaviour via some formal or semi-formal notation, which may concern different abstraction levels. Examples of notations are Unified Modeling Language (UML) (OMG, 2015), Lustre (BERGERAND, 1986), Software Cost Reduction (SCR) (HEITMEYER; BHARADWAJ, 2000), IMA Test Modelling Language (ITML) (EFKEMANN; PELESKA, 2011), among others. The main aim here is to avoid inconsistent and incomplete specifications: contradictory and unspecified behaviour, respectively.

When MBT is applied to analyse critical systems, it is also desired to clearly and formally state the conditions necessary to guarantee that the System Under Test (SUT) is correct with respect to its specification by means of a conformance relation, which can be used to prove that the test generation strategy is sound. A conformance relation assumes that, besides the specification, the implementation behaviour can also be described using the same notation of the specification model. This requirement is known as the testability hypothesis. Conformance is then defined as a mathematical relation between the specification and implementation models. In this situation, we say we have a formal MBT strategy (GAUDEL, 1995).

Initially, most conformance relations proposed in the literature addressed only functional (qualitative) system behaviour, and thus are unable to tackle non-functional real-time (quantitative) properties. One prominent relation is ioco (TRETMANS, 1999) that relates input and output events of Labelled Transition Systems (LTSs).

More recently, time-based relations have been devised: ioco$_{DTA}$ (KHOUMSI; JÉRON; MARCHAND, 2003), tioco$_{TTG}$ (KRICHEN; TRIPAKIS, 2004), rtioco (LARSEN; MIKUCIONIS; NIELSEN, 2004), tioco$_{M}$ (BRIONES; BRINKSMA, 2005), tioco$_{TorX}$ (BOHNENKAMP; BELINFANTE, 2005), tioco$_{Sch}$, tioco$^{R}_{M}$, and tioco$_{\zeta}$ (SCHMALTZ; TRETMANS, 2008), besides tioco$_{And}$ (ANDRADE et al., 2011). Some of these relations have a similar name (*tioco*) and, thus, we added subscripts to differentiate each of them.

Since the majority of these relations assumes LTSs as specification models, some limitations need to be considered when adopting them. Although it is possible to represent data operations using events as an abstraction, an LTS is not directly suitable for this purpose. Besides that, LTSs are not the best choice for compositional analyses and compositional test generation, which might be an important characteristic when dealing with larger and more complex systems. For instance, opposed to LTSs, process algebras such as Communicating Sequential Processes (CSP) allow that observations of a program may be deduced from the observations of its components with the aid of their denotational semantics, with no need to refer to the operational semantics directly (ROSCOE, 2010).

Despite the benefits of formal MBT, those who are not familiar with the models syntax and semantics may be reluctant to adopt theses formalisms. Moreover, most of these models are not available in the very beginning of the project, when usually natural-language requirements are available. One possible alternative to overcome these limitations is to employ Natural Language Processing (NLP) techniques to derive the required models from natural-language specifications automatically.

1.2 Natural-language processing

The demand of stating the desired system behaviour using formal models may sometimes be an obstacle for adopting formal MBT techniques, despite all its benefits. The model notations may be not easy to interpret by, for instance, aerospace and automotive development engineers. Hence, a specialist (usually mathematicians, logicians, computer engineers and scientists) is required when such languages, and their corresponding techniques, are used in business contexts.

Furthermore, most of these models are not yet available in the very beginning of the system development project. In the initial phases, only high-level requirement descriptions are usually available. According to the Federal Aviation Administration (FAA), which published a report (FAA, 2009) that discusses current practices concerning requirements engineering management, "*... the overwhelming majority of the survey respondents indicated that requirements are being captured as English text...*". This supports the thesis that, at the very beginning of system development, typically only natural-language requirements are documented. Therefore, due to these limitations, the need to be familiar with the models syntax and semantics and the absence of specification models in the very beginning of the system development, the use of MBT is postponed or neglected.

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

NLP has been studied since the 1950s, initially focusing on machine translation. Currently, we can identify a vast diversity of NLP applications, such as question-answering systems, natural-language interfaces, automated services over the telephone, tutoring systems, information extraction, text summarisation, among others (ALLEN, 1995). A complete NLP system usually counts on five processing levels, depending on its aim: morphological analysis, syntactic analysis, semantic mapping, discourse analysis and pragmatic analysis. Concerning MBT, the works (BODDU et al., 2004; SNEED, 2007; SANTIAGO JUNIOR; VIJAYKUMAR, 2012) address the joint use of MBT and NLP based on the second and third NLP levels cited before.

There is a trade-off concerning the application of NLP in MBT. Some studies are able to analyse a broad range of sentences, whereas others rely on controlled versions of natural language. The works that adopt the former approach usually depend on a higher level of user intervention to derive models and to generate test cases. Differently, the restrictions imposed by a Controlled Natural Language (CNL) might allow a more automatic approach when generating models and test cases. Ideally, a compromise between these two possibilities should be sought to provide a useful degree of automation along with a natural-language specification feasible to be used in practice.

1.3 Research question

In the light of the discussion presented so far, the main research question of this work is: how to automatically and formally generate test cases from natural-language requirements? As disccussed in Section 1.6, this question has already been addressed within more restricted contexts, where no data operations or time aspects are considered. In this work, we are particularly interested in systems whose behaviour can be described via data operations and that is time-dependent.

Here, we propose a formal MBT strategy for generating test cases from natural-language requirements: NATural Language Requirements to TEST Cases (NAT2TEST). Actually, it is a general approach that can be specialised via the adoption of different formal models and test generation tools. Nevertheless, we focus on a formal approach based on the CSP (ROSCOE, 2010) process algebra: NAT2TEST$_{CSP}$.

We dispense the need to know the syntax and semantics of the underlying notations, besides allowing early use of MBT, by means of NLP. In this way, the formal and semi-formal models internally used by the NAT2TEST strategy are automatically generated from the natural language requirements.

1.4 NAT2TEST – an overview

Hereafter, we provide a general explanation of the NAT2TEST strategy interleaved with an example to illustrate its application. As a running example, we consider a Vending Machine (VM), which is an adaptation of the coffee machine presented in (LARSEN; MIKUCIONIS; NIELSEN, 2004).

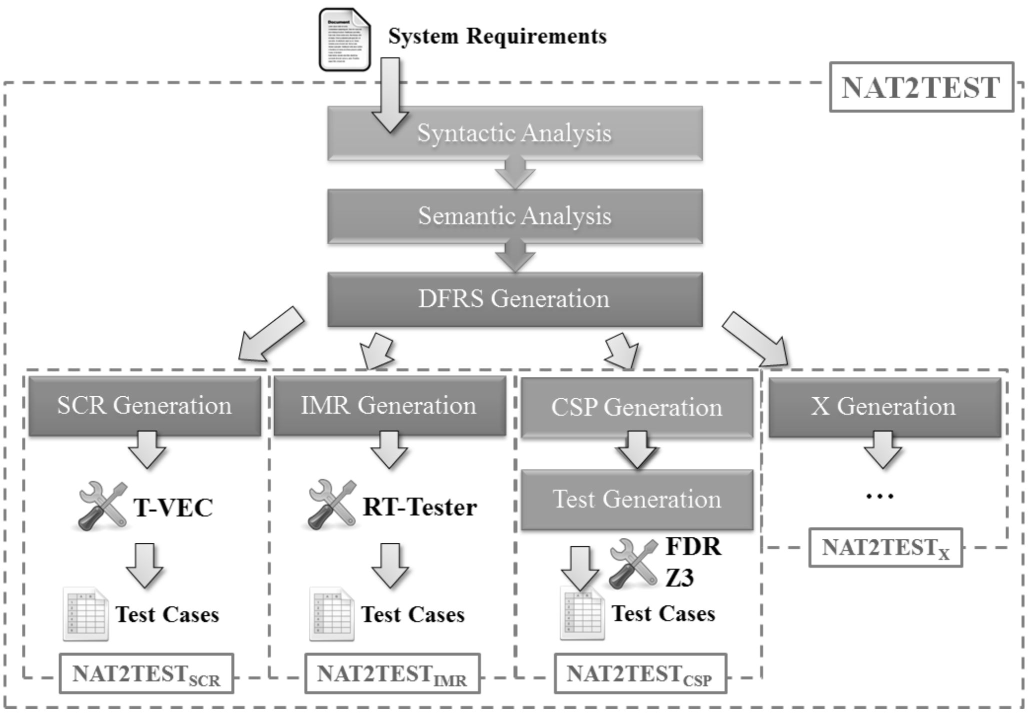
Running example Initially, the VM is in an *idle* state. When it receives a coin, it goes to the *choice* state. When the coffee option is selected, the system goes to the *weak* or *strong* coffee state. If coffee is selected within 30 seconds after inserting the coin, the system goes to the *weak coffee* state. Otherwise, it goes to the *strong coffee* state. Therefore, if the user selects the coffee option too quickly, a weak coffee is dispensed instead of a strong one.

The NAT2TEST strategy is tailored to generate tests for Data-Flow Reactive Systems (DFRSs): a class of embedded systems whose inputs and outputs are always available as signals. The input signals can be seen as data provided by sensors, whereas the output data are provided to system actuators. These systems can also have timed-based behaviour, which may be discrete or continuous.

Our test-generation strategy comprises a number of phases, as illustrated by Figure 1.1. The three initial phases are fixed: (1) syntactic analysis, (2) semantic analysis, and (3) DFRS generation; the remaining phases depend on the target formalism.

In this work, requirements are written according to a CNL based on English: the System Requirements Controlled Natural Language (SysReq-CNL), specially designed for editing requirements of data-flow reactive systems. The first phase of the NAT2TEST strategy (syntactic analysis) is responsible for verifying whether the requirements are in accordance with the SysReq-CNL grammar. For each valid requirement, its corresponding syntax tree is identified.

Figure 1.1: Phases of the NAT2TEST strategy



[Source: author]

Example The following requirement (VM) adheres to the SysReq-CNL grammar.

- REQ001 – *When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode.* □

In the second phase (semantic analysis), the requirements are semantically analysed using the case grammar theory. In this theory, a sentence is not analysed in terms of the syntactic categories or grammatical functions, but in terms of the semantic (thematic) roles played by each word/group of words in the sentence. Therefore, for each syntax tree the group of words that correspond to a thematic role is identified. The collection of thematic roles for a requirement is called the requirement frame.

Example Table 1.1 shows the requirement frame for REQ001. We note that the thematic roles are grouped into conditions and actions. The roles that appear in actions are the following: Action (ACT) – the action performed if the conditions are satisfied; Agent (AGT) – entity who performs the action; Patient (PAT) – entity who is affected by the action; and To Value (TOV) – the patient value after action completion. Similar roles are used in the conditions. □

Afterwards, the third phase (DFRS generation) derives DFRS models – an intermediate formal characterisation of the system behaviour from which other formal notations can be

Table 1.1: Example of requirement frame for REQ001 (vending machine)

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

[Source: author]

derived, such as SCR, Intermediate Model Representation (IMR), and CSP. The possibility of exploring different formal notations allows analyses from several perspectives, using different languages and tools (e.g., T-VEC (BLACKBURN; BUSSER; FONTAINE, 1997) for SCR, RT-Tester (PELESKA et al., 2011) for IMR). Besides that, it makes our strategy extensible.

Example A DFRS model comprises disjoint sets of input and output variables, as well as timers. The system behaviour is represented as functions, which have guards, along with the corresponding system reaction (represented as assignments). In what follows, one can see the guards, as well as the corresponding assignments, obtained from the requirement frame of REQ001. Some syntax sugar is used to simplify the explanation.

$$\text{current(the\_system\_mode) = idle} \wedge \text{current(the\_coin\_sensor) = true} \wedge$$
$$\text{previous(the\_coin\_sensor) = false} \rightarrow \text{the\_request\_timer := 0; the\_system\_mode := choice}$$

The predicate states that when the current value of the variable *the_system_mode* is idle, the current value of *the_coin_sensor* is true, but it was false previously (it changed to true, as specified in the requirement), then the system should react resetting the timer *the_request_timer*, besides assigning *choice* to the variable *the_system_mode*. □

In this work, we focus on the use of the process algebra CSP to generate test cases. In this context, the NAT2TEST strategy (NAT2TEST$_{CSP}$) comprises two additional phases. First, the DFRS model is encoded as CSP processes (CSP generation). With the aid of the Failures-Divergences Refinement (FDR)[1] and Z3 tools[2], symbolic and concrete test cases are generated.

[1]FDR tool – http://www.cs.ox.ac.uk/projects/fdr/
[2]Z3 tool – http://z3.codeplex.com/

Table 1.2: Example of test case for REQ001 (vending machine)

TIME	request	coin	mode	output
0.0	false	false	idle	strong
1.0	false	true	choice	strong

[Source: author]

Example To generate test cases, we use FDR to enumerate traces (sequences of events) from CSP specifications. From the traces, we obtain the test input data, as well as the expected output data. Time information is symbolically encoded in these traces. In what follows, one can see an excerpt from a trace yielded by FDR for the VM. The symbol "..." abstracts some events.

$$\begin{array}{l} < output.the\_system\_mode.I.1.the\_coffee\_machine\_output.I.0, ..., \\ \quad input.the\_coffee\_request\_button.B.false.the\_coin\_sensor.B.true, ..., \\ \quad delay.eta1.B.false.eta2.B.false.eta3.B.false.eta4.B.false, ..., \\ \quad output.the\_system\_mode.I.0.the\_coffee\_machine\_output.I.0, ... > \end{array}$$

The first event represents the initial output values of the system. The following events inform that the coin sensor should be set to true, and the system reacts going to the *choice* state (represent by the value *0*). The *delay* event comprises boolean values that, when true, denote symbolically that a corresponding time constraint (eta_i) has been satisfied. In this case, there is no constraint restricting the amount of time that might advance and, thus, all values are false. Regarding the system mode, the value 1 denotes the *idle* state. Concerning the machine output, 0 represents strong coffee (the standard initial value of this output signal). To obtain concrete test cases, it is necessary to use Z3 to instantiate the values related to the time-based behaviour. Table 1.2 shows part of a concrete test case obtained from the previous trace. □

Considering a CSP timed input-output conformance relation we defined (*csptio*), we prove that our testing theory is sound. In other words, if the execution of a generated test case leads to a fail verdict, it implies that the SUT is not correct with respect to its specification model, considering the definition of the aforementioned relation.

Moreover, the verification that an implementation conforms to its specification, based on csptio, is mechanised in terms of a high-level strategy by reusing successful techniques and tools: refinement checking (FDR) and Satisfiability Modulo Theories (SMT) solving (Z3). The former tool is used to analyse the conformance of data-related aspects as refinement checking, whereas the latter is employed to analyse the temporal aspects as constraint satisfaction problems. Therefore, its mechanisation does not require the implementation of complex algorithms or the manipulation of complex data structures. Furthermore, our mechanisation is sound with respect to the csptio definition.

The NAT2TEST strategy is automated by the NAT2TEST tool,[3] which is written in Java, and its Graphical User Interface (GUI) is built using the Eclipse RCP framework. The tool can be easily installed and it runs on multiple platforms.

1.5 Empirical evaluations

The NAT2TEST strategy, via its supporting tools, is evaluated considering examples from the literature, but also from the aerospace and the automotive industry: (i) a vending machine (VM – the toy example previously mentioned); (ii) a simplified control system for safety injection in a nuclear power plant (Nuclear Power Plant (NPP) – publicly available), (iii) the priority command function (Priority Control (PC) – provided by Embraer[4], our industrial partner); and (iv) part of the turn indicator system of Mercedes vehicles (Turn Indicator System (TIS) – publicly available).

For each example, we analyse two aspects: (i) performance, and (ii) the ability to detect defects by means of mutation analysis. As a baseline we consider the generation and execution of random tests using Randoop (PACHECO et al., 2007). Although random testing is mainly designed for unit testing, it fits into our analysis purposes as we execute the generated test cases on reference implementations in Java, since we do not have access to the embedded implementations due to security policies.

Moreover, to provide an empirical argument to whether the DFRS models are expressive enough to represent the behaviour of a timed reactive system defined using natural language, we assess whether test cases, either independently written by domain specialists from industry or generated by a commercial tool (RT-Tester) from the same set of requirements, are compatible with the corresponding DFRS models. By being compatible, we consider that there is a sequence of transitions in the DFRS model that illustrate the delays, the system inputs and the expected outputs described in the test case.

1.6 Scientific and technological contributions

In summary, the two main scientific contributions of this work are the following: (i) a formal testing theory based on a CSP timed input-output conformance relation (csptio), which extends with time aspects the relation cspio (NOGUEIRA; SAMPAIO; MOTA, 2014); and (ii) a formal model (DFRS) for timed reactive systems that can be automatically derived from natural-language requirements.

Regarding our first contribution, it differs from other formal testing theories (for instance, see the ones mentioned in Section 1.1), since, as far as we know, it is the first CSP-based

[3]Available for download at: `http://www.cin.ufpe.br/~ghpc/`

[4]*Empresa Brasileira de Aeronáutica*
`http://www.embraer.com/en-us/pages/home.aspx`

testing theory to support control, data and time information. In particular, we consider a symbolic codification of time that allows us to use standard CSP; we are not using Timed CSP (REED; ROSCOE, 1988) or tock-CSP (ROSCOE; HOARE; BIRD, 1997) to deal with discrete and continuous time. These notations were not considered for two main reasons. First, as time is not symbolically encoded, it is easier to face state-explosion problems when analysing models. Second, the respective tool support for timed analyses is not mature enough. Moreover, as we use CSP as our underlying notation, opposed to state-based notations such as Timed Input-Output Transition Systems (TIOTSs), compositional aspects of conformance testing might be pursuit as future work, since compositionality is a central aspect of CSP.

Concerning our second contribution, the formal model proposed here (DFRS) stands out by the richness of the model generated solely from natural-language requirements. The DFRS can represent input-output variables, besides discrete and continuous time information. In contrast, other works only cover one of these two aspects (ACEITUNA; DO; SRINIVASAN, 2014; BACKES et al., 2015; BODDU et al., 2004; NOGUEIRA; SAMPAIO; MOTA, 2014; ESSER; STRUSS, 2007; SIEGL; HIELSCHER; GERMAN, 2010) or rely on user intervention (AMBRIOLA; GERVASI, 2006; ILIC, 2007; LEVESON et al., 1994; MILLER et al., 2006; SCHNELTE, 2009; SANTIAGO JUNIOR; VIJAYKUMAR, 2012), when it is necessary to identify and classify entities, besides extracting information by hand.

Besides these two scientific contributions, it is worth mentioning the technological outcome of this research: the NAT2TEST tool – a fully operational tool that supports all phases of the strategy proposed here.

To accomplish these scientific and technological contributions, we have achieved the following specific results:

1. A CNL (SysReq-CNL) for specifying requirements of data-flow reactive systems;
2. An application of NLP and of the case grammar theory with respect to MBT;
3. Algorithms for deriving formal specifications (CSP, SCR, IMR) from DFRSs;
4. A CSP timed input-output conformance relation (csptio);
5. A sound mechanisation of csptio conformance verification using FDR/Z3;
6. A formal testing theory that is proved sound with respect to csptio;
7. Empirical evaluations of the proposed MBT strategy for different domains;
8. Tool support for the MBT strategy proposed here.

In 2014, this study was mentioned as a highlight in the Year in Review issue of the Aerospace America Magazine, a publication of the American Institute of Aeronautics and Astronautics (AIAA) (CARVALHO et al., 2014d).

1.7 Thesis structure

The remainder of this work is structured within the following chapters.

- Chapter 2 describes the first two phases of the NAT2TEST strategy. It presents the lexicon (dictionary) and the grammar of our CNL for data-flow reactive systems (SysReq-CNL). Afterwards, it presents the second phase of our strategy, by explaining the case grammar theory, how the content of the thematic roles are inferred from syntax trees, and contextual rules that are verified concerning the system requirements.
- Chapter 3 formalises the DFRS model. In particular, its symbolic and its expanded representations. Besides that, it presents the algorithms that derive DFRS models from thematic roles. Moreover, it performs a theoretical validation of these models by connecting such a representation to established ones in the literature.
- Chapter 4 explains how the process algebra CSP can be used to represent DFRSs. Then, it defines our formal testing theory, along with our CSP timed input-output conformance relation, which is used to prove the soundness of the testing theory.
- Chapter 5 describes the tool support for the NAT2TEST strategy by going through each of its constituent components. Furthermore, it discusses the empirical evaluations performed in this work.
- Chapter 6 discusses the state-of-the-art related to the two main scientific contributions of this study: timed conformance relations and formal models for natural-language requirements.
- Chapter 7 presents our conclusions and discusses future work.

2 Syntactic and semantic analysis

The first phase (Syntactic Analysis) of the NAT2TEST strategy verifies whether the system requirements are written according to a particular CNL. In the following phase (Semantic Analysis) an informal semantics are given to the requirements.

The ideas presented in this chapter are first discussed in (CARVALHO et al., 2013a), and later detailed in (CARVALHO et al., 2014c). In this chapter, we present these two phases: syntactic analysis (Section 2.1), semantic analysis (Section 2.2).

2.1 Phase I – syntactic analysis

To be automatically processed by the NAT2TEST strategy, the system requirements must be written according to the grammar of the SysReq-CNL. For each valid requirement, the corresponding syntax tree is generated and used as input by the subsequent phase (Semantic Analysis).

2.1.1 The SysReq-CNL – a CNL for system requirements

In general, a CNL consists of a subset of a particular natural language (e.g., English), aiming to avoid textual ambiguity and complexity (WYNER et al., 2010). Some organisations adopt simple CNLs to provide standardization to technical documentation. These CNLs consist of guideline rules, such as: "write short sentences", and "avoid using passive voice".

Examples of such CNLs are: Avaya Controlled English[1], for technical publications in the telecommunication and computing industry (O'BRIEN, 2003); Caterpillar Technical English (CTE), used by Caterpillar Inc. (a heavy equipment manufacturing company) to support consistent, high-quality authoring and translation of technical documents from English into a variety of target languages (KAMPRATH et al., 1998); and the prominent ASD-STE[2] (AeroSpace and Defence Simplified Technical English), a CNL created for the aerospace in-

[1] `http://www.avaya.com/usa/`
[2] `http://www.boeing.com/boeing/phantom/sechecker/se.page`

dustry which aims for people to write clear documentation, making texts easier to understand, especially for non-native English speakers (ASD, 2005).

A different approach involves more formal CNLs, which can be used to write system specifications in order to improve the reliability of the overall software development process. These CNLs are based on a predefined vocabulary and on a restricted set of grammar rules, used to identify the sentences underlying syntactic structure, which can be further mapped into some formal representation (HEITMEYER; BHARADWAJ, 2000). The challenge is to define a syntactic structure that allows automatic processing of requirements, but without losing writing naturalness.

In this light, we designed the SysReq-CNL, specially tailored for editing requirements of data-flow reactive systems: a class of embedded systems whose inputs and outputs are always available as signals. The input signals can be seen as data provided by sensors, whereas the output data are provided to system actuators. These systems can also have timed-based behaviour, which may be discrete or continuous.

Although the SysReq-CNL prevents some sources of ambiguity, such as the use of pronouns, it is still possible to write ambiguous requirements due to lexical ambiguity (see Section 5.2.1). This way, there might be more than one syntax tree per input requirement. In this case, the NAT2TEST strategy terminates, and the requirement analyst shall manually remove the ambiguity, and thus define which syntax tree shall be considered as the correct interpretation of the requirement.

In what follows, one can see an example of an unambiguous requirement that adheres to the SysReq-CNL grammar. The tool support of our strategy guides the user when writing requirements according to the SysReq-CNL grammar (see Section 5.2.1). Requirements that do not adhere to this grammar cannot be analysed, and the tool indicates the source of problem such that the user can correct it.

- *When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode.*

The development of the SysReq-CNL was monitored by our industrial partner (Embraer), who considered it simple to understand and easy to use, besides imposing standardization to requirements without losing naturalness. Despite the close connection with Embraer, since the creation of this CNL was influenced by examples provided by Embraer, the SysReq-CNL has proved to be suitable to express requirements in other domains, as well illustrated by the examples carried out in this work.

Our CNL is defined by a phrase structure context free grammar, and a lexicon containing the application domain vocabulary. The current version of SysReq-CNL (CARVALHO et al., 2014c) is an enhancement of the one described in (CARVALHO et al., 2013a). The main difference is an improved grammar structure to cover new writing scenarios. Furthermore, now we also consider a pre-processing step to enable text reuse. These differences are better explained later on.

When extending the SysReq-CNL via the definition of new grammar elements and rules it suffices to update the phases directly related to the change. Depending on the change, the impact might be restricted to the syntactic analysis phase or other phases might be updated as well, for instance, to define new contextual and inference rules, and how this new information should be used when deriving the requirements formal model.

2.1.1.1 The SysReq-CNL lexicon

As already mentioned, the SysReq-CNL lexicon conveys the vocabulary of the application domain. Its entries are classified into lexical categories, also known as Parts of Speech (POS) (ALLEN, 1995). In this work, we consider the following frequently used lexical categories:

- determiners (*DETER*), words that determine (limit) the meaning of a noun (e.g., a number, an article or a personal pronoun) (CRYSTAL, 2008);
- nouns (*NSING* for singular and *NPLUR* for plural);
- adjectives (*ADJ*);
- adverbs (*ADV*);
- verbs with inflections, for example, *VBASE* - base form, *VPRE3RD* - for the 3^{rd} person in present form;
- conjunctions (*CONJ*);
- prepositions (*PREP*).

In order to simplify the SysReq-CNL grammar rules, we create two special categories: *NUMBER* for numbers, and *COMP* for comparisons (e.g., less than). Yet, we have special entries to identify keywords that are used in the grammar definition: “and” (AND), “or” (OR), “not” (NOT), “shall” (SHALL), “:” (COLON), and “,” (COMMA).

Finally, we highlight that, as the lexicon is domain dependent, it must be manually created and maintained considering the system current domain. Despite the initial effort, the vocabulary tends to become stable, which minimizes the maintenance effort. This is a natural assumption for NLP systems that rely on a pre-defined set of lexical entries. Yet, it is possible to reuse part of an existing lexicon for a new application domain (e.g., prepositions, conjunctions, etc.).

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

[Source: author]

2.1.1.2 The SysReq-CNL grammar

We follow the phrase structure grammar theory, originally introduced by Noam Chomsky, which assumes a binary division of the clause into a noun phrase and a verb phrase. These phrases can be further divided into other constituents, until we reach the word level (CRYSTAL, 2008). A constituent can be a word or a phrase (group of words) that occurs as a unit in the grammar rewrite rules. Note that these constituents will be nodes in the syntax tree.

This way, the SysReq-CNL grammar consists of phrase structure rewrite rules which define the syntactic structures accepted by our parser. Our grammar was defined as a Context Free Grammar (CFG), represented in the Extended Backus-Naur Form (EBNF) notation; see Figure 2.1 – words in uppercase denote terminal symbols, and a ";" delimits the end of each production. In this work, terminal symbols correspond to lexical categories.

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*. A *VerbPhraseC-*

ondition begins with a *VerbCondition* (the *to be* verb 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*.

This concise grammar is able to represent requirements written using several different sentence formations, and it is not restricted to one specific application domain. We have successfully applied the SysReq-CNL in the different domains considered in this work: VM, NPP, PC, and TIS (see Section 5.1).

Bellow, we present a typical requirement as it was originally written by the Embraer requirements team, and the corresponding form, rewritten to adhere to the SysReq-CNL. This requirement is part of the specification of the PC. As it can be seen, the rewritten requirement is similar to the original version.

- *Original*: The Priority Logic Function shall assign value 0 (zero) to Command In-Control output when: left Priority Button is not pressed AND right Priority Button is not pressed AND left Command is on neutral position AND right Command is on neutral position.

- *Rewritten*: When the left priority button is not pressed, and the right priority button is not pressed, and the left command is on neutral position, and the right command is on neutral position, the Priority Logic Function shall assign 0 to the Command-In-Control output.

Now we present other examples of typical requirement adhering to the SysReq-CNL.

- REQ001 - *When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode.* [VM]

- REQ002 - *When the water pressure becomes higher than or equal to 900, and the pressure mode is low, the Safety Injection System shall assign permitted to the pressure mode.* [NPP]

- REQ003 - *When the voltage is greater than 80, and the flashing timer is greater than or equal to 220, and the left indication lights are off, and the right indication lights are off, and the flashing mode is left flashing or the flashing mode is left tip flashing,* *the lights controller component shall: assign on to the left indication lights, assign off to the right indication lights, reset the flashing timer.* [TIS]

- REQ004 - *When the voltage is greater than 80, and the flashing timer is greater than or equal to 220, and the left indication lights are off, and the right indication lights are off, and the flashing mode is right flashing or the flashing mode is right tip flashing, the lights controller component shall: assign off to the left indication lights, assign on to the right indication lights, reset the flashing timer.* [TIS]

The current version of the SysReq-CNL (CARVALHO et al., 2014c) has some improvements with respect to its first version (CARVALHO et al., 2013a). Initially, it was not possible to have an adverb derived from a *VariableState*. This restriction prevents recognizing requirements, such as the third example above mentioned (REQ003), since the words "off" and "on" are adverbs in the context of a *VariableState*.

Besides this expressiveness change, now we allow more concise specifications by means of text reuse. To support text reuse, we offer aliases and condition prefixes. To exemplify a possible usage of aliases, consider the third and the fourth examples. They might be rewritten considering the definition of an alias (e.g., *ACTION_1*) to refer to *reset the flashing timer*, which appears in both requirements. To refer to an alias, one just needs to refer to the alias name within the requirement. Besides that, the writer can reuse condition prefixes, which are identical for two or more different requirements, using a special keyword (...).

Let *ACTION_1* refer to *reset the flashing timer*, *ACTION_5* to *assign on to the left indication lights, assign off to the right indication lights*, and *ACTION_6* to *assign off to the left indication lights, assign on to the right indication lights*, then REQ003 and REQ004 can be rewritten as a single requirement as follows.

- *When the voltage is greater than 80, and the flashing timer is greater than or equal to 220, and the left indication lights are off, and the right indication lights are off,*
  ... and the flashing mode is left flashing or the flashing mode is left tip flashing, the lights controller component shall: ACTION_5, ACTION_1.
  ... and the flashing mode is right flashing or the flashing mode is right tip flashing, the lights controller component shall: ACTION_6, ACTION_1.

Despite the reuse benefits of these two features, they are optional. Moreover, our parser is not aware of them as they are just pre-processed. Therefore, the parser input are requirements without aliases or condition prefixes.

2.2 Phase II – semantic analysis

The second processing phase of the NAT2TEST strategy receives as input the generated syntax tree, and delivers a requirement semantic representation. In this work, we adopt the case grammar theory (FILLMORE, 1968) to represent meaning. In this theory, a sentence is analysed in terms of the semantic (thematic) roles played by each word/group of words in the sentence (e.g., agent, patient, location, object, and instrument, among others). A simple example of this system is: *“Mary (agent) broke (action) the window (patient) with a stone (instrument)”*. The thematic roles used in this work are detailed in the following section.

Thematic roles are organised into case frames (detailed below), a structure which represents the semantic meaning of the sentence. The obtained case frame based semantic representation, which is informal, can then be mapped later into an internal formal model to provide a formal semantics for the system requirements. Note that this informal semantic representation decouples the formal model generation process from the SysReq-CNL syntactic rules. Thus, the formal model generation process will not be affected by updates in the CNL (e.g., to capture new syntactic structures) that do not cause changes to the underlying semantics.

2.2.1 Thematic roles, case frames, and requirement frames

In the case grammar theory, the verb is the main element of the sentence, and it determines its possible semantic relations with words in the sentence; that is, the role that each word plays with respect to the action or state described by the sentence’s verb. Considering the previous example regarding *“Mary”*, the possible relationships with the verb *“break”* are represented by the agent, patient and instrument Thematic Roles (TRs).

The verb’s associated TRs (or *cases*) are aggregated into a Case Frame (CF). In practical terms, a CF can be viewed as a structure consisting of slots (each one representing a verb’s thematic role) to be filled in by sentence elements. So, the TRs in a CF specify the structural context of the verb.

Note that this classification system may resemble the functional syntactic analysis, which classifies the sentence elements by their function in a sentence (e.g., subject, object, predicative, etc.). It is even possible to establish a correspondence between some of our TRs and these functional classes (e.g., agent $\mapsto$ subject; action $\mapsto$ verb and patient $\mapsto$ object). However, the functional classification does not provide the fine grained (semantic) analysis needed in our approach to generate formal models of the specification. For instance, concerning some verbs, the indirect object might refer to the patient (e.g., *assign 10 to x.* – *x* is the patient), whereas it might refer to different roles (e.g., *change from 10 to 20*), regarding other verbs. Thus, we use TRs instead of functional classes.

Continuing our explanation about case frames, it is important to say that each verb in the language vocabulary corresponds to only one CF. However, one CF may be related to several verbs (those which bear the same TRs – e.g., *“add”* and *“subtract”*). In a CF, thematic roles may be obligatory or optional (e.g., the CF for the verb *“break”* has *“agent”* and *“patient”* as obligatory roles, whereas the TR *“instrument”* is optional).

In this work, we consider nine thematic roles (the adopted nomenclature was inspired by (ALLEN, 1995)). The *Condition Modifier* role was defined by us, whereas the other TRs were obtained from the related literature.

As seen before, requirements have the form of action statements guarded by conditions. The underlying semantics is that when the conditions are true, the corresponding actions shall be performed. Note that the verbs (or verbal phrases) used in the condition clauses are different from the ones used in the action statements. As such, we consider specific TRs for condition clauses, and other TRs for the action statements.

Below, we can find the TRs associated with action statements.

- *ACT*: the action performed if the conditions are satisfied;
- *AGT*: entity who performs the action;
- *PAT*: entity who is affected by the action;
- *TOV*: the Patient value after action completion.

Now we list the TRs associated with conditions.

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

In our CNL, as a requirement comprises more than one verb (at least one condition and one action), we have more than one CF related to each phrase. All derived CFs are joined afterwards to compose what we call a Requirement Frame (RF). In other words, a RF is a structure to encode data such as the one presented in Table 2.1: a collection of case frames for conditions and action statements.

Considering the requirement REQ003, previously presented in Section 2.1.1.2 and reproduced below, Table 2.1 shows the corresponding case frames.

- REQ003 - *When the voltage is greater than 80, and the flashing timer is greater than or equal to 220, and the left indication lights are off, and the right indication lights are off, and the flashing mode is left flashing or the flashing mode is left tip flashing, the lights controller component shall: assign on to the left indication lights, assign off to the right indication lights, reset the flashing timer.* [TIS]

Table 2.1: Example of requirement frame for REQ003 (turn indicator system)

Condition #1 - Main Verb (CAC): is			
CPT:	the voltage	CFV:	-
CMD:	greater than	CTV:	80
Condition #2 - Main Verb (CAC): is			
CPT:	the flashing timer	CFV:	-
CMD:	greater than or equal to	CTV:	220
Condition #3 - Main Verb (CAC): are			
CPT:	the left indication lights	CFV:	-
CMD:	-	CTV:	off
Condition #4 - Main Verb (CAC): are			
CPT:	the right indication lights	CFV:	-
CMD:	-	CTV:	off
Condition #5 - Main Verb (CAC): is			
CPT:	the flashing mode	CFV:	-
CMD:	-	CMD:	left flashing
OR - Main Verb (CAC): is			
CPT:	the flashing mode	CFV:	-
CMD:		CTV:	left tip flashing
Action - Main Verb (ACT): assign			
AGT:	the lights controller component	TOV:	on
PAT:	the left indication lights		
Action - Main Verb (ACT): assign			
AGT:	the lights controller component	TOV:	off
PAT:	the right indication lights		
Action - Main Verb (ACT): reset			
AGT:	the lights controller component	TOV:	-
PAT:	the flashing timer		

[Source: author]

2.2.2 Contextual and inference rules

In this section, we explain how we infer the contents of each thematic role from the syntax trees obtained after the first phase of the NAT2TEST strategy. Before the inference process, some contextual rules are verified according to the verbs being used. These rules inspect the syntax trees for semantic errors.

Currently, we consider the verbs: *to add*, *to assign*, *to be*, *to become*, *to change*, *to reset*, and *to subtract*. It is worth mentioning that they are sufficient to write all requirements of the examples considered in this work. Nevertheless, one can extend the NAT2TEST strategy incorporating new verbs. It suffices to update the phases directly related to the change: define new contextual and inference rules, besides how this new information should be used when deriving the requirements formal model. The contextual rules are defined considering the expected grammar structure of the aforementioned verbs, while the inference rules take into considering the structure of the SysReq-CNL grammar. In what follows, we discuss the NAT2TEST contextual and inference rules.

Rule 1. *The verbs* to add, to assign, to reset, *and* to subtract *shall be used only in action statements.*

Rule 2. *The verbs* to be, to become, *and* to change *shall be used only in the conditions of a requirement.*

Rule 1 restricts the use of verbs that express an action to the context of action statements. Analogously, Rule 2 restricts the use of verbs that express a possible value or a modification between values to the conditions of a requirement.

Depending on the verb, some thematic roles are mandatory, others shall not be used, whereas some are optional. In what follows, we define rules that state which thematic roles are related to each verb.

Rule 3. *Concerning the verbs* to add, to assign, *and* to subtract, *all thematic roles associated with action statements are mandatory.*

Rule 4. *Concerning the verb* to reset, *the thematic role TOV is not used, and the other thematic roles associated with action statements are mandatory. We assume* to reset *to be equivalent to assigning a standard value. This standard value may be dependent on the context (e.g., in the context of boolean values, reset is equivalent to assign false, whereas the default value is 0 when dealing with numerical values).*

Rule 5. *Concerning the verbs* to be *and* to become, *the thematic role CFV is not used, the thematic role CMD is optional, and the other thematic roles associated with conditions are mandatory. We assume* to become X *to be equivalent to* changes from not X to X.

Rule 6. *Concerning the verb* to change, *the thematic roles CFV and CMD are optional, and the other thematic roles associated with conditions are mandatory. The use of* to change *without the CFV is equivalent to the use of* to become.

For instance, *changes to X* is equivalent to write *becomes X*. We also have one rule to avoid an incoherent use of the CMD thematic role.

Rule 7. *Concerning the thematic role CMD, incoherent modifiers shall not be used. Thus, it is not possible to have* greater than *and* lower than *simultaneously for the same thematic role as it does not make sense.*

Finally, the last rules concern the prepositions and number of complements that shall be used with each verb.

Rule 8. *Concerning the verb* to add, *the structure is:* to add X to Y.

Rule 9. *Concerning the verb* to assign, *the structure is:* to assign X to Y.

Rule 10. *Concerning the verb* to be, *the structure is:* to be X.

Rule 11. *Concerning the verb* to become, *the structure is:* to become X.

Rule 12. *Concerning the verb* to change, *the structure is:* to change from X to Y *or* to change to Y.

Rule 13. *Concerning the verb* to reset, *the structure is:* to reset X.

Rule 14. *Concerning the verb* to subtract, *the structure is:* to subtract X from Y.

To exemplify the need for these last rules, consider the phrase "When input1 becomes from 10, and input2 changes from 20, the system shall: add 30 to output1 to output2, assign 40, subtract 50, reset from output2.". Despite being syntactically correct, the phrase does not make sense since it does not respect the expected verb structure. It violates the Rules 11, 12, 8, 9, 14, and 13, in this order.

It is worth emphasising that these rules need to be updated if new verbs are considered to write the requirements. However, as previously said, this set of verbs and rules were sufficient to describe requirements from different domains and structures.

The contents of each thematic role are inferred from the syntax trees generated after the first phase of the NAT2TEST strategy. This is done by visiting the syntax trees searching for particular patterns. The mapping process starts by identifying the requirement's verbs.

We have verbs in conditions, which are mapped to the CAC role, and in actions, which correspond to the ACT role. The CAC and ACT verbs will sometimes guide the selection of the appropriate thematic roles to compose the requirement's CF. This happens when the syntax tree structure is not enough to determine the correct semantic mapping. In what follows, we present the patterns related to each thematic role. These patterns are applied individually to each requirement's condition and action.

Rule 15. *ACT: comprises the terminals of* VBASE *(a verb on its base form).*

Rule 16. *AGT: comprises the terminals of the* NounPhrase *that is a direct descendant of* ActionClause.

Rule 17. *PAT: comprises the terminals of*
(i) the VariableState *that is a descendant of the* PrepositionalPhrase *that is a descendant of* VerbPhraseAction, *if the corresponding ACT is equal to* add, assign, *or* subtract*;*
(ii) the VariableState *that is a descendant of* VerbPhraseAction, *if the corresponding ACT is equal to* reset.

Rule 18. *TOV: comprises the terminals of the* VariableState *that is a direct descendant of the* VerbComplement *that is a descendant of* VerbPhraseAction, *if the corresponding ACT is equal to* add, assign, *or* subtract.

Figure 2.2: Example of inference of the thematic roles content within actions

```
Syntax Tree:
...
|−ActionClause
 |−NounPhrase
  |−DETER
   |−the)
  |−Noun
   |−NSING
    |−system
 |−VerbPhraseAction
  |−SHALL
   |−shall
  |−COLON
   |− :
  |−VerbAction
   |−VBASE
    |−assign
  |−VerbComplement
   |−VariableState
    |−NUMBER
     |−30.0
   |−PrepositionalPhrase
    |−PREP
     |−to
    |−VariableState
     |−NounPhrase
      |−Noun
       |−NSING
        |−output1
```

ACT: assign [Rule 15]
AGT: the system [Rule 16]
PAT: output1 [Rule 17.(i)]
TOV: 30.0 [Rule 18]

[Source: author]

To exemplify these rules, consider the phrase: “When input1 is not lower than 31.5, and input2 changes from false to true, the system shall: assign 30.0 to output1, reset output2.”. Figure 2.2 shows the syntax tree obtained from the first action, as well as the corresponding thematic roles along with the patterns (rules) applied.

The following inference rules are related to thematic roles that appear within conditions.

Rule 19. *CAC: comprises the terminals of* VerbCondition.

Rule 20. *CPT: comprises the terminals of the* NounPhrase *that is a direct descendant of* Condition.

Rule 21. *CFV: comprises the terminals of the* VariableState *that is a direct descendant of the* PrepositionalPhrase *that is a descendant of* VerbPhraseCondition, *if the corresponding CAC is*

Figure 2.3: Example of inference of the thematic roles content within conditions

```
Syntax Tree:
...
|−Condition
 |−NounPhrase
  |−Noun
   |−NSING
    |−input1
 |−VerbPhraseCondition
  |−VerbCondition
   |−VTOBE_PRE3
    |−is
  |−NOT
   |−not
  |−ComparativeTerm
   |−COMP
    |−lower than
  |−VerbComplement     CAC: is [Rule 19]
   |−VariableState     CPT: input1 [Rule 20]
    |−NUMBER           CMD: not lower than [Rule 23]
     |−31.5,           CFV: - (no rule)
                       CTV: 31.5 [Rule 22.(i)]
```

[Source: author]

equal to change, *and if the preposition associated with the* PREP *of this* PrepositionalPhrase *is* from.

Rule 22. *CTV: comprises the terminals of:*
(i) the VariableState *that is a direct descendant of the* VerbComplement *that is a descendant of* VerbPhraseCondition, *if the corresponding CAC is equal to* be *or* become*;*
(ii) the VariableState *that is a direct descendant of the* PrepositionalPhrase *that is a descendant of* VerbPhraseCondition, *if the corresponding CAC is equal to* change, *and if the preposition associated with the* PREP *of this* PrepositionalPhrase *is* to.

Rule 23. *CMD: the terminals of* ComparativeTerm *appended to a* not, *if there is a node* NOT *that is a direct descendant of* VerbPhraseCondition.

To exemplify some of the last rules, consider again the phrase: “When input1 is not lower than 31.5, and input2 changes from false to true, the system shall: assign 30.0 to output1, reset output2.”. Figure 2.3 shows the syntax tree obtained from the first condition, as well as the corresponding thematic roles along with the patterns (rules) applied.

2.3 Concluding remarks

This chapter detailed the first two steps necessary to derive test cases from system requirements written in natural language. First, the system requirements are analysed to assert whether they comply to the grammar of the SysReq-CNL – a CNL specially tailored for editing requirements of data-flow reactive systems. This CNL allows writing requirements that have the form of action statements guarded by conditions. Therefore, it might not be suitable for modelling natural-language requirements outside the domain of reactive systems.

Afterwards, the case grammar theory is used to provide an informal semantic interpretation for the system requirements. Moreover, rules tailored for the verbs being used are considered to identify potential semantic errors in the requirements. If no errors are found, we are ready to assign a formal interpretation to the system requirements based on the inferred thematic roles. This is the main goal of the third phase of the NAT2TEST strategy.

3
A formal model for requirement frames

In the semantic analysis, thematic roles are used to provide an informal semantics to the system requirements. Now we derive a formal representation for the requirements. Therefore, we define here a symbolic, timed and state-rich automata-based notation for representation of natural-language requirements: Data-Flow Reactive System (DFRS) (CARVALHO et al., 2014b), which is derived automatically from the requirement frames.

DFRS models can be translated to more concrete notations (SCR, IMR, CSP, among others) and, thus, reuse the tool support available for these notations. Therefore, a DFRS allows exploring the original requirements from different perspectives, besides being independent of a specific tool. Moreover, translating requirement frames to an intermediate formal notation such as DFRSs is a promising alternative, since it was devised considering particularities of the specific domain of this work: reactive systems. The direct translation from requirement frames to more concrete notations is a more elaborate task, since it would be necessary to consider the particularities of the chosen notation when representing the behaviour of reactive systems.

DFRSs have two representations: a more abstract (symbolic) representation – Symbolic Data-Flow Reactive Systems (s-DFRSs), which inherently avoids an explicit representation of possibly infinite sets of states and, thus, the state space explosion problem; and an expanded representation – Expanded Data-Flow Reactive System (e-DFRS), which is built dynamically from its symbolic counterpart, possibly limited to some bound, and then used to bounded analyses such as requirements reachability, determinism, and completeness. To avoid confusion, consider that, hereafter, s-DFRS and e-DFRS refer to symbolic and expanded DFRSs, respectively, while DFRS refers to both of them.

In this chapter, besides presenting the formal definition in Z (ISO, 2002) of s-DFRSs (Section 3.1) and e-DFRSs (Section 3.3), we also describe the algorithms related to the third phrase of the NAT2TEST strategy, when s-DFRSs models are derived from RFs (Section 3.2).

Furthermore, we also prove that an e-DFRS can be characterised as a TIOTS – a labelled transition system extended with time, which is widely used to characterise conformance relations for timed reactive systems (Section 3.4). Being more abstract than a TIOTS, a DFRS comprises a more concise representation of timed requirements and, thus, easier to represent and to process computationally. For instance, a TIOTS can have an infinite number of states, whereas s-DFRSs are characterised by finite elements.

3.1 Definition and properties of an s-DFRS

In this section, first, we give an informal overview of the definition of DFRSs, and then we provide a formal definition for its symbolic representation. It is important to say that all definitions in Z presented here are syntactically correct and typed checked with the CZT plug-in for Eclipse[1].

3.1.1 Overview of DFRSs

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

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

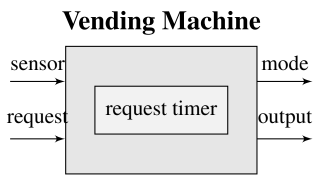
The states of an e-DFRS are connected by *delay* and *function* transitions. A delay transition represents the observation of the input signal values after a given delay, whereas the function transition represents how the system reacts to the input signals: the observed values of the output signals. The transitions are encoded as assignments to input and output variables as well as timers.

As a running example, we consider the VM briefly discussed in the introduction and fully presented in Chapter 5.1. Briefly recapping its main behaviour, after receiving a coin and a coffee request, the VM produces weak or strong coffee depending upon the time elapsed between inserting the coin and requesting coffee. The time required to produce weak coffee is also different from that of strong coffee.

As shown in Figure 3.1, in this example we have two input signals related to the coin sensor (*sensor*) and the coffee request button (*request*). A *true* value means that a coin was inserted and the coffee request button was pressed. There are two output signals related to the system mode (*mode*) and the vending machine output (*output*). The values communicated by these signals reflect the system possible states (*idle*, *choice*, *weak*, *strong*, and *reset*) and the possible outputs (*undefined*, *weak*, and *strong*).

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

Figure 3.1: The vending machine specification – abstract representation



[Source: author]

The VM has just one timer: the *request* timer, which is used to register the moments when a coin is inserted, when the coffee request button is pressed, and when the coffee is produced. Figure 3.2 illustrates a scenario where there is continuous observation of the input and output signals. If we had chosen to observe the system discretely, we would have a similar scenario, but with a discrete number of samples over time.

In Figure 3.2, a coin is inserted 2 seconds after starting the vending machine (the signal *sensor* changes to 1 – *true*). Immediately, the system state changes from *idle* to *choice*. Here, the system states are encoded as follows: *idle* $\mapsto$ 1, *choice* $\mapsto$ 0, *weak* $\mapsto$ 3, *strong* $\mapsto$ 2, and *reset* $\mapsto$ 4. Therefore, this change is represented by changing the value of the signal *mode* from 1 to 0. In this example, the signal *sensor* remains true for 3 seconds.

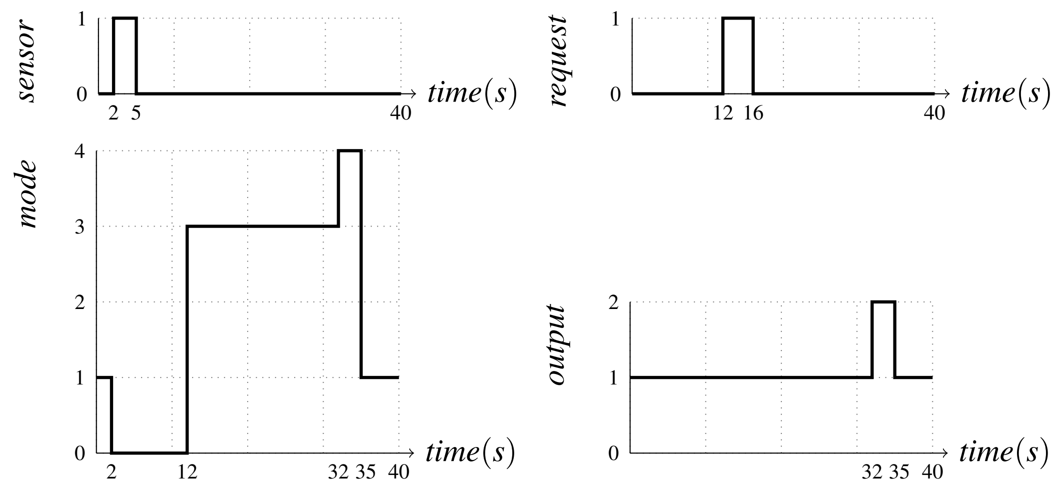
When 10 seconds have elapsed since the coin was inserted, which happens 2 seconds after starting the vending machine, the user requests a coffee (the signal *request* becomes true when the system global clock is equal to 12). At this moment, the system state changes to *weak* coffee (the signal *mode*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

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

As a state-based notation, the example illustrated in Figure 3.2 is represented in a an e-DFRS as a set of states and transitions (see Figure 3.3). The states are related by *delay* (D) and *function* (F) transitions. As already mentioned, the former represents time elapsing along with input stimuli, whereas the latter describes an instant reaction of the system. It is important to emphasize that the diagram presented in Figure 3.3 is just an illustration of part of an e-DFRS based on the particular scenario depicted in Figure 3.2.

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

Figure 3.2: Example of signals for the vending machine



[Source: author]

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

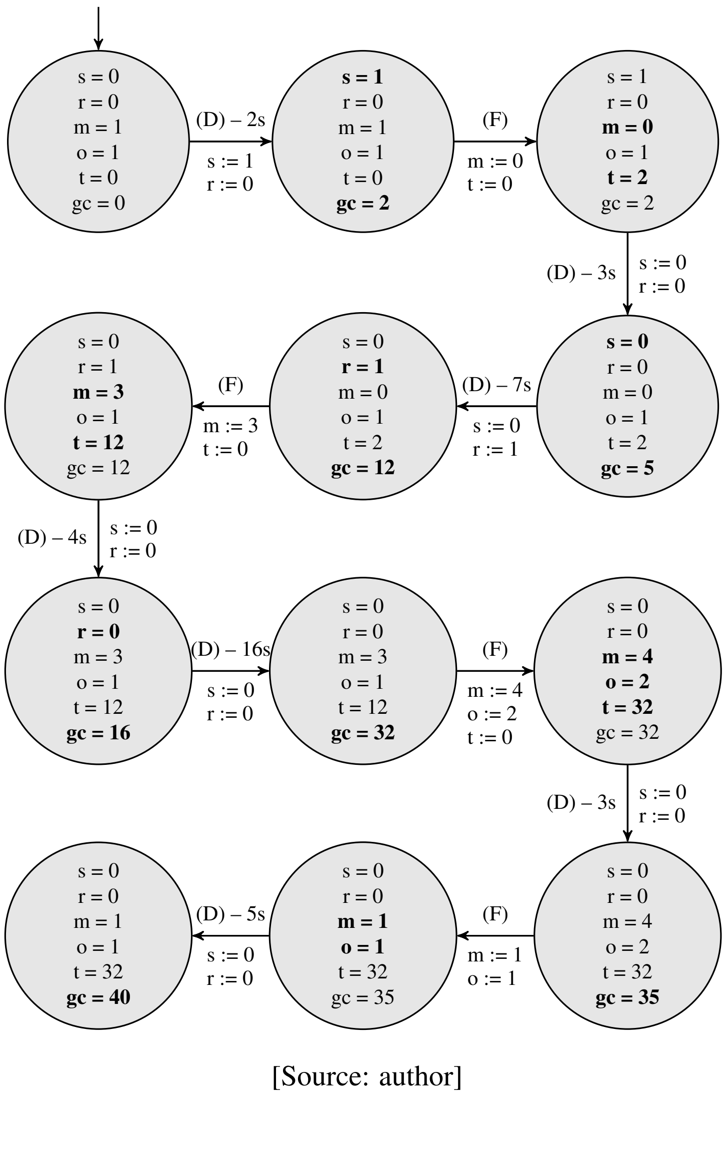
The reset of a timer is represented by assigning 0 to it, but it is encoded as assigning the current value of the system global clock to the corresponding timer. This is possible as we have a single and global clock source (the system global clock). Otherwise, we would need to update the value of all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

We note that the changes produced by the transitions are highlighted in bold in Figure 3.3. Moreover, each delay transition comprises the values of all input signals, whereas each function transition considers a subset of the output signals, besides the internal request timer, when appropriate.

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

The main difference between an s-DFRS and its expanded version, characterised as just explained and illustrated in Figure 3.3, is that the characterisation of an s-DFRS defines the initial state and means of calculating the next states via a set of functions. Differently, an e-DFRS comprises the set of all states and how they are related by delay and function transitions. Now, after presenting an informal discussion of DFRS models, we define the s-DFRS precisely.

Figure 3.3: The vending machine specification – e-DFRS representation
[s ≡ coin sensor, r ≡ the coffee request button, m ≡ the system mode,
o ≡ the system output, t ≡ the request timer, and gc ≡ the system global clock]



s = 0 r = 0 m = 1 o = 1 t = 0 gc = 0
(D) – 2s
s := 1 r := 0
s = 1 r = 0 m = 1 o = 1 t = 0 **gc = 2**
(F)
m := 0 t := 0
s = 1 r = 0 **m = 0** o = 1 **t = 2** gc = 2
(D) – 3s
s := 0 r := 0
s = 0 r = 0 m = 0 o = 1 t = 2 **gc = 5**
(D) – 7s
s := 0 r := 1
s = 0 **r = 1** m = 0 o = 1 t = 2 **gc = 12**
(F)
m := 3 t := 0
s = 0 r = 1 **m = 3** o = 1 **t = 12** gc = 12
(D) – 4s
s := 0 r := 0
s = 0 **r = 0** m = 3 o = 1 t = 12 **gc = 16**
(D) – 16s
s := 0 r := 0
s = 0 r = 0 m = 3 o = 1 t = 12 **gc = 32**
(F)
m := 4 o := 2 t := 0
s = 0 r = 0 **m = 4** **o = 2** **t = 32** gc = 32
(D) – 3s
s := 0 r := 0
s = 0 r = 0 m = 4 o = 2 t = 32 **gc = 35**
(F)
m := 1 o := 1
s = 0 r = 0 **m = 1** **o = 1** t = 32 gc = 35
(D) – 5s
s := 0 r := 0
s = 0 r = 0 m = 1 o = 1 t = 32 **gc = 40**

[Source: author]

3.1.2 Formal model of an s-DFRS

Formally, an s-DFRS is a 6-tuple: (I, O, T, $gcvar$, s_0, F). Inputs (I) and outputs (O) are system variables, whereas timers (T) are a distinct kind of variable, which can be used to model temporal behaviour. The system global clock is *gcvar*, a variable whose values are non-negative numbers representing a discrete or a dense (continuous) time. The initial state is s_0, and F is a set of functions. In what follows, we describe in Z the constituent components of an s-DFRS.

3.1.2.1 Inputs, outpus and timers

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

$$[NAME]$$

$$gc : NAME$$

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

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

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

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

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

More complex types are not needed since we are dealing with systems whose inputs and outputs are signals. As float numbers are not part of Standard Z, we provide an axiomatisation that fulfils our needs. For a more comprehensive axiomatisation, we refer, for instance, to ProofPower-Z[2].

The schema *DFRS_VARIABLES* defines the variables of a DFRS as a set of inputs (I), outputs (O), timers (T) and a global clock (*gcvar*). In Z, a schema is a named element used

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

to structure and encapsulate definitions for reuse. As I and O are distinct and non-empty sets, we have that a DFRS has at least one input and one output variable. Differently, one can have a system with no timers: a DFRS whose behaviour is not dependent on time elapsing. These three sets (I, O and T) are disjoint.

```
DFRS_VARIABLES
  I, O : SVARS
  T : STIMERS
  gcvar : NAME × TYPE
  ─────────────
  gcvar = (gc, nat) ∨ gcvar = (gc, ufloat)
  disjoint ⟨dom I, dom O, dom T⟩
  ran T ⊆ {gcvar.2}
```

We note that our model can represent discrete, $gcvar = (gc, nat)$, or continuous time, $gcvar = (gc, ufloat)$, systems. Besides that, the type of all timers must be the same ($\operatorname{ran} T \subseteq \{gcvar.2\}$): one can analyse the behaviour of the system discretely or continuously, but not in both ways simultaneously. In Z, the notation $.i$ allows us to access the i-th element of a tuple. Therefore, $gcvar.2$ stands for the type of the system global clock.

Example 1 Besides the global clock, five variables are identified in the VM example (see Figure 3.3): two system inputs (*the_coin_sensor—s*, *the_coffee_request_button—r*), two outputs (*the_system_mode—m*, *the_coffee_machine_output—o*), and one timer (*the_request_timer—t*). The sensor and the button are modelled by booleans that indicate whether a coin has been inserted or the button has been pressed. The system mode and the output of the VM are non-negative numbers. The request timer is modelled as a non-negative natural number since the temporal properties of the VM are defined in terms of discrete values (e.g., "... 30 seconds ..." instead of "... 30.0 seconds ..."). □

3.1.2.2 Initial state

A state is a relation between names and values, which include boolean and numerical values. The letter R refers to $\mathbb{R}$, and R^+ to the positive elements of $\mathbb{R}$. The element *VALUE* is a free type: a set with explicit structuring information – constructors $\langle\!\langle\rangle\!\rangle$ are used to represent boolean and numerical values ($b\langle\!\langle ... \rangle\!\rangle$ and $i\langle\!\langle ... \rangle\!\rangle, n\langle\!\langle ... \rangle\!\rangle, f\langle\!\langle ... \rangle\!\rangle, uf\langle\!\langle ... \rangle\!\rangle$, respectively).

$$BOOL\_VALUES ::= TRUE \mid FALSE$$

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

Each name within a state is mapped to two values: the first one represents the previous value, and the second one the current value. Therefore, Figure 3.3 shows a simplified and not the actual representation of states. For instance, in the first state, $s = 0$ should be $s \mapsto (b(\mathit{false}), b(\mathit{false}))$, and, in the second state, $s = 1$ should be $s \mapsto (b(\mathit{false}), b(\mathit{true}))$. Note that in Figure 3.3 we use numbers to represent boolean values.

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

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

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

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

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

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

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

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

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

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

Example 2 Considering the example shown in Figure 3.3, its initial state is:

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

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

3.1.2.3 Functions

The system behaviour is defined as a non-empty finite set ($\mathbb{F}_1$) of functions (see schema *DFRS_FUNCTIONS*) that describe how the system reacts in a given context. There is one function per system component; if the system comprises parallel components, we are going to have one function describing the behaviour of each component.

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

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

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

When both guards evaluate to true in a given state, the system reacts instantly performing the corresponding assignments. These reactions are the *function transition* (F) shown in Figure 3.3. An s-DFRS does not capture this dynamic behaviour (occurrence of reactions explicitly), but only includes the definition of the function that symbolically characterises the reactions.

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

$$EXP == CONJ$$
$$CONJ == \mathbb{F}\, DISJ$$
$$DISJ == \mathbb{F}_1\, BEXP$$

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

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

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

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

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

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

$$var\_consistent\_be : \mathbb{P}(BEXP \times (VNAME \rightarrow\!\!\!\!\!+ TYPE) \times (VNAME \rightarrow\!\!\!\!\!+ TYPE))$$

$$\begin{aligned}
&\forall be : BEXP; f, T : VNAME \rightarrow\!\!\!\!\!+ TYPE; n : VNAME \mid varName(be) = n \bullet\\
&\quad (be, f, T) \in var\_consistent\_be \Leftrightarrow\\
&\quad (n \in \mathrm{dom} f) \land be.3 \in values(f(n)) \land\\
&\quad (be.2 = le \lor be.2 = lt \lor be.2 = gt \lor be.2 = ge \Rightarrow be.3 \notin values(bool)) \land\\
&\quad (n \in \mathrm{dom}\, T \Rightarrow be.1 \in \mathrm{ran}\, current)
\end{aligned}$$

To get the name referenced by a binary expression, we rely on the auxiliary function *varName*, which projects the *VNAME* within the constructors *current* or *previous*. This concept of consistency is lifted to guards, which are said to be consistent if, and only if, all of its binary expressions are consistent.

The last component of a function entry is a finite and non-empty set of assignments (*ASGMTS*). The right-hand side of an assignment (*ASGMT*) is a value (*VALUE*), and the left-hand side is the name of a variable (*VNAME*). Note that it is not possible to define a set of assignments that considers different values to the same variable (e.g., $\{(x,n(0)),(x,n(1))\}$). If such a scenario were allowed (non-deterministic assignments), it would not be clear what would be the value of x after the assignments.

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

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

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

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

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

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

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

3.1.2.4 Complete definition of an s-DFRS

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

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

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

3.2 Phase III – generation of s-DFRS

An s-DFRS is derived from requirement frames according to three consecutive steps. First, the system variables are identified. Then, the functions that describe the system behaviour are defined. Finally, an s-DFRS is created from these two pieces of information. The following sections detail each step. To illustrate them, we consider the following requirement of the VM, as well as the corresponding requirement frame (see Table 3.1).

- *When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode.*

3.2.1 Identifying variables

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

Table 3.1: Example of requirement frame for REQ001 (vending machine)

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

[Source: author]

word “timer” as a suffix. Timers can appear both in conditions and in statements. Our algorithm for identifying variables (Algorithm 1) receives as input a list of requirement frames and yields a list of variables.

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

For example, Table 3.1 shows that *“the system mode”* is the CPT of the first condition. Thus, if the corresponding variable has not yet been identified (lines 6–7), we create a new variable considering the CPT content, replacing white spaces by an underscore (lines 8–9), which is done by the *“toString”* function (line 5). So, in this case, we create the variable *the_system_mode*. Then, we verify whether the variable has the word *“timer”* as a suffix; if so, it is classified as a timer, otherwise it is an input (lines 10–11). Then we add the created variable to the list of identified variables (line 12).

To infer the type of the variable we analyse the value associated with it in the case frame, which is the content of the CTV role. For instance, the variable *the_system_mode* is associated with the value *“idle”* in the first condition of Table 3.1. Thus, the algorithm extracts the CTV content (line 13), and uses it to infer the variable type, which is done by the *inferType* function (line 14) that is later explained (see Algorithm 2).

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

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

Algorithm 1: Identify Variables

```
input  : reqCFList
output : varList
varList = new List();
for reqCF ∈ reqCFList do
    for andCond ∈ reqCF do
        for orCond ∈ andCond do
            varName = toString(orCond.CPT);
            var = varList.find(varName);
            if var == null then
                var = new Var(varName);
                var.type = undefined;
                if varName.endsWith("timer") then var.kind = timer;
                else var.kind = input;
                varList.add(var);
            value = toString(orCond.CTV);
            inferType(var, value, varList);
    for action ∈ reqCF do
        varName = toString(action.PAT);
        var = varList.find(varName);
        if var == null then
            var = newVar(varName);
            var.type = undefined;
            if varName.endsWith("timer") then var.kind = timer;
            else var.kind = output;
            varList.add(var);
        else if var.kind = input then
            var.kind = output;
        value = toString(action.TOV);
        if value ≠ null then inferType(var, value, varList);
for var ∈ varList do
    if var.type = enum then
        if var.possibleValuesList.size() = 1 then var.type = boolean;
        else var.type = integer;
    else if var.kind = timer ∧ var.type = undefined then
        var.type = float
gcVar = newVar(gc);
allDiscrete = true;
allContinuous = true;
for var ∈ varList do
    if var.kind = timer ∧ var.type = integer then allContinuous = false;
    if var.kind = timer ∧ var.type = float then allDiscrete = false;
if allDiscrete then gcVar.type = integer;
else if allContinuous then gcVar.type = float;
else throw Exception("timers: incompatible types");
varList.add(gcVar);
```

[Source: author]

- The variable value is the content of the TOV role, excluding the case (line 27) when the *"reset"* verb is used (see the first action of Table 3.1). In this case, the TOV is empty and what is assigned to the timer is the system global clock, which is an integer or a float. In such a situation, we do not try to infer the type of the timer. If this timer is also mentioned in a condition, its type is determined by the value associated with it in this condition. If this timer is never mentioned within a condition, its type is left undefined (lines 32–33), and then we assume that its type will be float, representing continuous time.

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

3.2.1.1 Type inference

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

Algorithm 2: Infer Type

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

[Source: author]

If the variable is a timer, the associated values need to be numbers (float or integer), otherwise an exception is raised (lines 5–8). If the variable is an input or an output, its type might be boolean (if the value is the boolean constants “true” or “false” – line 10), a float or an integer (lines 11-12), or an *enum* (e.g., if the value is a string such as "idle" – see Table 3.1). It is worth mentioning that the *enum* type is not expected within DFRS models. Therefore, later it is mapped to an integer.

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

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

3.2.2 Identifying functions

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

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

With this information, we check whether each conjunction concerns the same type of guards (static or timed). If it is not the case, an exception is raised (lines 13, 16). This is necessary, since we want to divide the conditions into two disjoint categories (static and timed) without performing boolean algebra manipulation. As examples, we consider the following abstract cases: $c_1 : T \wedge (c_2 : T \vee c_3 : S) \wedge c_4 : S$ and $c_1 : T \wedge (c_2 : S \vee c_3 : S) \wedge c_4 : S$, where c_i

Algorithm 3: Identify Functions

```
input  : $reqCFList, varList$
output : $functionMap$
$functionMap = new\ Map()$;
for $reqCF \in reqCFList$ do
    $staticGuard, timedGuard = null$;
    for $andCond \in reqCF$ do
        $guardType = undefined$;
        $newTerm = null$;
        for $orCond \in andCond$ do
            $exp = generateConditionExpression(orCond, varList)$;
            $varName = toString(orCond.PAT)$;
            $var = varList.find(varName)$;
            if $var.kind = timer$ then
                if $guardType = undefined$ then $guardType = timed$;
                else if $guardType = static$ then $throw\ Exception($“format error”$)$;
            else
                if $guardType = undefined$ then $guardType = static$;
                else if $guardType = timed$ then $throw\ Exception($“format error”$)$;
            if $newTerm = null$ then $newTerm = exp$;
            else $newTerm = newTerm +$ “$\vee$” $+ exp$;
        if $guardType = static$ then
            if $staticGuard = null$ then $staticGuard = (newTerm)$;
            else $staticGuard = staticGuard +$ “$\wedge$” $+ (newTerm)$;
        else
            if $timedGuard = null$ then $timedGuard = (newTerm)$;
            else $timedGuard = timedGuard +$ “$\wedge$” $+ (newTerm)$;
    $actionList = new\ List()$;
    for $action \in reqCF$ do
        $actionStatement = generateStatement(action, varList)$;
        $actionList.add(actionStatement)$;
    $componentName = toString(reqCF.actions.get(0).AGT)$;
    $function = functionMap.find(componentName)$;
    if $foundFunction = null$ then
        $function = new\ Function()$;
        $functionMap.add(componentName, function)$;
    $previousActionList = function.find(staticGuard, timedGuard)$;
    if $previousActionList \neq null$ then
        $previousActionList.add(actionList)$
    else
        $function.add(staticGuard, timedGuard, actionList)$
```

[Source: author]

denotes the i-th condition, and "*:S*" and "*:T*" indicates whether the condition concerns a static or a timed guard, respectively. The first expression does not comprise two disjoint sets of static and timed guards, whereas the second one does (timed: c_1; static: $(c_2 \vee c_3) \wedge c_4$). Lines 17–18 group each disjunction in *newTerm*, and lines 19–24 group the disjunctions in *staticGuard* or *timedGuard* depending on the type of the disjunctions.

After identifying the static and timed guards, the algorithm iterates over the list of actions of the requirement frame and creates a list of action statements using the function *generateStatement* (lines 25–28). Then, the algorithm checks whether a function is already created for the current agent. If not, it creates a new function and maps it to the current agent (lines 29–33). Finally, the element $staticGuard \times timedGuard \times actionList$ is added to the corresponding function (lines 37–38). If an entry for the pair $staticGuard \times timedGuard$ already exists, the list of actions is added to this entry (lines 35–36). In what follows, we explain the auxiliary functions: *generateConditionExpression* and *generateStatement*.

3.2.2.1 Generating condition expressions

Algorithm 4 yields a boolean expression from a single case frame, which comprises the condition thematic roles. The variable name is obtained from the CPT role (line 1). Initially (lines 2–7), the algorithm verifies whether the verb being used, which is obtained from the CAC role, denotes the previous value of a variable. This is the case when the verbs *"was"* and *"were"* are used. In this situation, the boolean expression concerns not the current value of a variable, but its previous one. As explained in Section 3.1.2.3, we use the predicate $previous(v)$ to denote the previous value of v, and $current(v)$, to denote its current value.

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

The next step is to obtain the value, which is compared to the variable. First, the value is obtained from the CTV role (line 8). If the value is a string, we consider as value the index of this string within the list of possible values of the corresponding variable (line 9). For a concrete example, see the one shown in the end of Section 3.1.2.2.

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

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

Finally, lines 28–45 deal with a special case that occurs when the verbs *"change"* or *"become"* are used. When *"change"* is used, as explained in depth in Section 2.2.2, we expect

Algorithm 4: Generate Condition Expression

```
input  : $cond, varList$
output : $exp$
$varName = toString(cond.CPT)$;
$var = varList.find(varName)$;
$verb = cond.CAC$;
if $verb.equals(“was”) \vee verb.equals(“were”)$ then
    if $var.kind = timer$ then $throw\ Exception$(“previous cannot be used with timers”);
    else $varName$ =“previous(” + $varName$ + “)”;
else $varName$ =“current(” + $varName$ + “)” ;
$value = toString(cond.CTV)$;
if $\neg\ isInteger(value) \wedge \neg\ isFloat(value) \wedge \neg\ isBoolean(value)$ then
$value = var.possibleValuesList.getIndex(value)$ ;
$modifier = cond.CMD$;
$negation, lesserThan, greaterThan, equalTo = false$;
if $modifier.contains(“not”)$ then $negation = true$;
if $modifier.contains(“lesser\ than”)$ then $lesserThan = true$;
if $modifier.contains(“greater\ than”)$ then $greaterThan = true$;
if $modifier.contains(“equal\ to”)$ then $equalTo = true$;
if $(lowerThan \vee greaterThan) \wedge var.type = boolean$ then
$throw\ Exception$(“lt/le/gt/ge cannot be used with booleans”) ;
$operator = new\ String()$;
if $lesserThan$ then
    if $equalTo$ then $operator$ =“$le$”;
    else $operator$ =“$lt$”;
else if $greaterThan$ then
    if $equalTo$ then $operator$ =“$ge$”;
    else $operator$ =“$gt$”;
else
    $operator$ =“$eq$”;
$exp = varName\ +\ operator\ +\ value$;
if $negation$ then $exp$ =“$\neg$ (” + $exp$ + “)” ;
if $verb.contains(“change”) \vee verb.contains(“become”)$ then
    $prevExp = null$;
    if $cond.CFV \neq null$ then
        $auxiliaryCond = new\ OrCond()$;
        $auxiliaryCond.CPT = cond.CPT$;
        $auxiliaryCond.CAC$ =“was”;
        $auxiliaryCond.CTV = cond.CFV$;
        $previousExp = generateConditionExpression(auxiliaryCond)$;
    else
        $auxiliaryCond = new\ OrCond()$;
        $auxiliaryCond.CPT = cond.CPT$;
        $auxiliaryCond.CAC$ =“was”;
        $auxiliaryCond.CTV = cond.CTV$;
        $auxiliaryCond.CMD = cond.CMD$;
        $previousExp = generateConditionExpression(auxiliaryCond)$;
        $previousExp = \neg\ previousExp$;
        $previousExp$ =“$\neg$ (” + $previousExp$ + “)”;
    $exp = prevExp$ + “$\wedge$” + $exp$;
```

[Source: author]

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

This algorithm is tightly dependent on the verbs used. However, the verbs currently supported by our approach are sufficient to express requirements from different examples and domains. If more verbs are used, one just needs to extend this function, informing how to form an expression from its thematic roles. No extra change is needed, since the DFRS model is not dependent on the verbs being used in the requirements.

3.2.2.2 Generating action statements

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

Algorithm 5: Generate Statement

```
   input  : $action, varList$
   output : $actionStatement$

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

[Source: author]

The next step concerns the identification of the value being assigned to the involved variable (lines 5–9). If the verb is *"reset"*, the value that is assigned to the timer is 0 or 0.0, depending on its type (integer or float). As already mentioned, and detailed when describing how an s-DFRS is used to produce an expanded one in Section 3.3.2, this assignment actually means assigning to the timer the system global clock. If the variable is not a timer, the value is the content of the TOV role (line 8). If the content of TOV is not an integer, a float or a boolean, it is a string. Therefore, we consider as value the index of this string within the list of possible values of the corresponding variable (line 9). Finally, the action statement is created assembling the variable and the assigned value (lines 10–11).

3.2.3 Creating an s-DFRS

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

Algorithm 6: Derive s-DFRS

```
input  : $reqCFList$
output : $dfrs$
1  $varList = identifyVariables(reqCFList)$;
2  $inputList, outputList, timerList = new\ List()$;
3  $gc = null$;
4  $initialBinding = new\ Map()$;
5  for $var \in varList$ do
6      if $var.kind = input$ then $inputList.add(var)$;
7      else if $var.kind = output$ then $outputList.add(var)$;
8      else if $var.kind = timer$ then $timerList.add(var)$;
9      else $gc = var$;
10     if $var.type = integer$ then $initialBinding.add(var.name, 0)$;
11     else if $var.type = float$ then $initialBinding.add(var.name, 0.0)$;
12     else $initialBinding.add(var.name, false)$;
13 $functionMap = identifyFunctions(reqCFList, varList)$;
14 $dfrs = new\ s\_DFRS()$;
15 $dfrs.I = inputList$;
16 $dfrs.O = outputList$;
17 $dfrs.T = timerList$;
18 $dfrs.s_0 = initialBinding$;
19 $dfrs.gcvar = gc$;
20 $dfrs.F = functionMap$;
```

[Source: author]

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

The algorithms presented here were tested considering examples from the literature and from the industry, when test cases, independently written or generated, were analysed to check whether they are compatible with the corresponding DFRS models, which were obtained by the application of the previously presented algorithms. See Section 5.3.3 for further details.

3.3 Definition and properties of an e-DFRS

Here, we formalise e-DFRSs (Section 3.3.1), and show how they can be obtained from their symbolic counterpart (Section 3.3.2). Although s-DFRS models have the information we need to generate test cases (see Chapter 4), we introduce here another notation for three main reasons. First, e-DFRSs provides a formal semantics of s-DFRSs. Second, they can be used to verify properties of the system requirements (Section 3.3.3). Finally, as a state-based notation, it allows us to connect such a semantic representation to established ones in the literature and, thus, characterise the expressiveness of DFRS models.

3.3.1 Formal model of an e-DFRS

An e-DFRS differs from the symbolic one as it encodes the system behaviour as a state-based machine, whereas an s-DFRS does that symbolically via definition of functions. As we detail later, states are obtained from an s-DFRS by applying its functions to states where the corresponding guards evaluate to true, but also letting the time evolve.

3.3.1.1 Transition relation

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

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

A transition relation (an element of *TRANSREL* defined below) comprises a set of transitions (*TRANS*). A transition relates two states by a label (*TRANS_LABEL*). As shown in Figure 3.3, this label can be of a *delay* (*del*) or a *function* (*fun*) transition.

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

A function transition represents the system instantaneous reaction as assignments (*ASGMTS*), which are performed atomically. It is worth noting that, although the function transition describes an instantaneous reaction, it is possible to model system reactions that occur after some time elapsing. We just need to consider a timer, which is reset when the event of interest happens, and then use it later to check the elapsed time and to decide on what event to engage next. For instance, this approach is used in the VM example (see Figure 3.3). When the coffee request button is pressed, the request timer is reset (see the first state on the second row). Afterwards, when a specific time has elapsed, the system reacts producing coffee (see the last state on the third row).

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

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

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

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

$$delayTransition : TRANS\_LABEL \rightarrow\!\!\!\!\!\!+\ DELAY \times ASGMTS$$

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

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

To be well typed, a function transition must modify only values of outputs and timers.

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

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

$$
\begin{array}{|l}
\mathit{well\_typed\_function\_transition} : \mathbb{P}(\mathit{TRANS\_LABEL} \times \\
\qquad (\mathit{VNAME} \rightarrow\!\!\!\!\!+ \mathit{TYPE}) \times (\mathit{VNAME} \rightarrow\!\!\!\!\!+ \mathit{TYPE})) \\
\hline
\forall \mathit{label} : \mathit{TRANS\_LABEL}; O, T : \mathit{VNAME} \rightarrow\!\!\!\!\!+ \mathit{TYPE} \mid \\
\qquad \mathit{label} \in \operatorname{ran} \mathit{fun} \bullet (\mathit{label}, O, T) \in \mathit{well\_typed\_function\_transition} \Leftrightarrow \\
\qquad (\operatorname{dom}(\mathit{functionTransition}(\mathit{label})) \subseteq (\operatorname{dom} O \cup \operatorname{dom} T))
\end{array}
$$

$$
\begin{array}{|l}
\mathit{well\_typed\_delay\_transition} : \mathbb{P}(\mathit{TRANS\_LABEL} \times (\mathit{VNAME} \rightarrow\!\!\!\!\!+ \mathit{TYPE})) \\
\hline
\forall \mathit{label} : \mathit{TRANS\_LABEL}; I : \mathit{VNAME} \rightarrow\!\!\!\!\!+ \mathit{TYPE} \mid \mathit{label} \in \operatorname{ran} \mathit{del} \bullet \\
\qquad (\mathit{label}, I) \in \mathit{well\_typed\_delay\_transition} \Leftrightarrow \\
\qquad \operatorname{dom}(\mathit{delayTransition}(\mathit{label})).2 = \operatorname{dom} I
\end{array}
$$

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

$$
\begin{array}{|l}
\mathit{clock\_compatible\_transition} : \mathbb{P}(\mathit{TRANS\_LABEL} \times (\mathit{NAME} \times \mathit{TYPE})) \\
\hline
\forall \mathit{label} : \mathit{TRANS\_LABEL}; \mathit{gcvar} : \mathit{NAME} \times \mathit{TYPE} \bullet \\
\qquad (\mathit{label}, \mathit{gcvar}) \in \mathit{clock\_compatible\_transition} \Leftrightarrow \\
\qquad \mathit{label} \in \operatorname{ran} \mathit{del} \wedge \\
\qquad ((\mathit{delayTransition}(\mathit{label})).1 \in \operatorname{ran} \mathit{discrete} \Rightarrow \mathit{gcvar}.2 = \mathit{nat}) \wedge \\
\qquad ((\mathit{delayTransition}(\mathit{label})).1 \in \operatorname{ran} \mathit{dense} \Rightarrow \mathit{gcvar}.2 = \mathit{ufloat})
\end{array}
$$

The delay transitions also need to be compatible with the system global clock in the sense that if the delay is discrete (an element of ran *discrete*), the type of the system global time must be *nat*, whereas if the delay is dense (an element of ran *dense*), the type of the clock must be *ufloat*.

As a consequence, all delay transitions share the same type of delay, meaning that they are all discrete or dense. This is captured by the *clock_compatible_transition* property.

Now, we define in the schema *DFRS_TRANSITION_RELATION* the transition relation (*TR*) of an e-DFRS as an element of *TRANSREL*.

As previously said, we assume that when the system is ready to react it does so instantaneously. Therefore, it would not make sense to have both delay and function transitions from the same state, since the system always react (performing the function transition), instead of letting the time evolve (performing the delay transition). This invariant is formalised in what follows by the first predicate: for every two transitions ($trans1$ and $trans2$) emanating from the same state ($trans1.1 = trans2.1$), they are either function (they belong to $\operatorname{ran} fun$) or delay transitions (they belong to $\operatorname{ran} del$).

DFRS_TRANSITION_RELATION

$TR : TRANSREL$

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

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

For a concrete example, we refer to the second delay transition presented in Figure 3.3: after the delay of 3s, there is no reaction by the system (there is no function transition), and the system state remains the same until the following delay transition. Moreover, we note that the possibility of adding a function transition to this state (the last state on the second row) would violate the invariant that requires that only function or delay transitions fire from the same state.

3.3.1.2 Complete definition of an e-DFRS

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

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

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

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

We note that there is a different definition when dealing with timers ($n \in \operatorname{dom} T$). In this case, the reset of a timer, which is represented by assigning 0, is encoded as an assignment of the current value of the global clock, $(s(gc)).2$. As the system has a single clock, it is easier to encode time reset by assigning the current value of the global clock, instead of assigning 0 and updating its value every time a delay transition is performed.

$$nextState : (STATE \times (NAME \nrightarrow TYPE) \times ASGMTS) \rightarrow STATE$$

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

Therefore, when evaluating timed guards such as $t < v$, where t is a timer and v a value, we actually evaluate the result of $(gc - t) < v$, where gc is the current value of the system global clock. Despite this representation, the previous value of the timer remains unchanged.

3.3.2 From s-DFRSs to e-DFRSs

The function *expandedDFRS* defines how an e-DFRS can be obtained from a symbolic one. The inputs (I), outputs (O), timers (T), the global clock ($gcvar$), and the initial state (s_0) are the same within both representations ($dfrs.I = symDFRS.I \land dfrs.O = symDFRS.O \land dfrs.T = symDFRS.T \land dfrs.gcvar = symDFRS.gcvar \land dfrs.s_0 = symDFRS.s_0$). The transition relation (TR) is obtained via the auxiliary function *buildTR*.

$$dfrs.TR = buildTR(\{dfrs.s_0\}, \emptyset, dfrs.I, dfrs.O, dfrs.T, symDFRS.F)$$

The states of an e-DFRS (S) are defined as the states related by this transition relation ($trans.1$ and $trans.3$), besides the initial state.

$$expandedDFRS : s\_DFRS \rightarrow e\_DFRS$$

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

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

As an inductive function, the base case for *builtTR* happens when *toVisit* is empty. For this value of *toVisit*, we have that $buildTR(toVisit, visited, I, O, T, F)$ is an empty transition relation. In the inductive case, *toVisit* is not empty and, thus, there is at least one state *s* in the states to visit ($s : toVisit$).

The result of *buildTR* is then defined as the union of the relation transition ($tr1$) obtained via *genTransitions*, which considers the emanating transitions from *s*, with the result of the recursive application of *buildTR*. This recursive application considers the not yet visited states, and also the new states reached by $tr1$ $(toVisit \cup \{trans : tr1 \bullet trans.3\}) \setminus (visited \cup \{s\})$. We note that we also need to add *s* to the set of visited states ($visited \cup \{s\}$).

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

The function *genTransitions* identifies either function or delay transitions from a given state *s*. Delay transitions are performed from stable states, whereas function transitions occur in non-stable states.

A state *s* is stable, $(s, ...) \in is\_stable$, when it does not represent a situation that triggers a system reaction: for all entries of the functions ($entry \in f$) of an s-DFRS ($f \in F$), their static ($entry.1$) and timed guards ($entry.2$) evaluate to false. The predicates *static_guards_true* and *timed_guards_true*, which are not presented here (see Appendix B), are defined as the set of all static and timed guards that evaluate to true in a given state.

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

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

If a state *s* is stable, there are delay transitions emanating from *s* for all possible delays, $delay \in possibleDelays(...)$, which is formalised later, and all possible valid assignments, those whose values are consistent with the variable types ($asgmts.2 \in values(I(asgmts.1))$). These assignments also range over the complete set of inputs ($\mathrm{dom}\, asgmts = \mathrm{dom}\, I$). The reached state is defined by the function *nextState*, but also updating the system global clock based on the performed delay ($nextState(...) \oplus \{(gc, ... + ...)\}$).

These three information (the given state – s; the delay transition considering a delay value and assignments – $del((delay, assigmts))$; and the reached state – $nextState(...) \oplus \{(gc, ...)\}$) are used to define the delay transition part of the result of *genTransitions*.

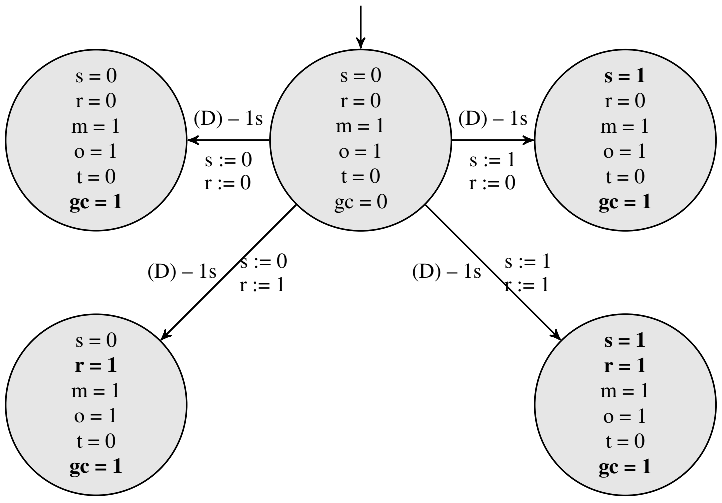
Figure 3.4 shows a concrete example of delay transitions emanating from the initial state of the VM. We note that we have a transition for each valid combination of input values: the coin sensor and the request button remain false (first state on first row), only the coin sensor becomes true (third state on first row), only the request button becomes true (first state on second row), and both signals become true (second state on second row).

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

Although only the transitions with delay equal to 1 second are shown, there are transitions with greater delays (2s, 3s, ...) emanating from the initial state. In this case, all delays are possible and, thus, there is no upper bound. This leads to an infinite number of delay transitions emanating from the initial state.

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

Figure 3.4: The vending machine specification – example of delay transitions



[Source: author]

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

The situation described in the end of the last paragraph (reaching a non-stable state by just letting the time advance) happens in the VM when the system is producing coffee. After pressing the coffee request button, if a weak coffee is going to be produced, we observe this system reaction within 10 to 30 seconds.

Therefore, if we are in the first state on the third row (Figure 3.3), for delays greater than or equal to 10 and lower than or equal to 30, we observe a system reaction leading the system to the reset state, besides changing accordingly the system output. In such a state, for instance, it would not make sense to have a delay transition, whose delay is 31, since we would be modelling an input received after elapsing 31 seconds, but before this input being received we should have observed a system reaction. This captures the principle of *delayable* transitions: the time might advance an arbitrary amount as long as it does not disable an enabled transition.

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

$10..\infty$.

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

To define the set of possible delays that we need to consider when generating delay transitions, we rely on the function *genPossibleDelays*. For a given state s, if the result of the application of *maxDelays* is empty ($maxDelays(...) = \emptyset$), it means that there is no upper bound we need to consider: all delays are possible, $genPossibleDelays(...) = \{delay : DELAY\}$.

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

Otherwise, we can perform all delays that are lower than or equal to the lowest upper bound defined by *maxDelays*, $(discrete^{\sim})(delay) \leq miniumDelay(...)$. The function *mininumDelay* yields this lowest upper bound, whose definition is not shown here as it is straightforward (see Appendix B).

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

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

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

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

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

The other elements of an e-DFRS are directly obtained from the corresponding symbolic ones. It is important to note that, when obtaining an e-DFRS from an s-DFRS, some structural information is lost, namely, the element F (set of functions), since this information is diluted over the labels of delay and function transitions.

3.3.2.1 Soundness of generation of an e-DFRS

Besides presenting a function that yields an e-DFRS from a symbolic one, it is important to show that this function is sound: for all s-DFRSs, all invariants of *e_DFRS* hold in the obtained e-DFRS (Theorem 3.3.1). Although presenting the complete proof is not within the scope of this work, we provide here a proof sketch that the function *expandedDFRS* is sound.

Theorem 3.3.1. *Soundness of expandedDFRS*

$$\forall d : \mathit{s\_DFRS} \bullet \mathit{expandedDFRS}(d) \in \mathit{e\_DFRS}$$

The three invariants of *DFRS_VARIABLES* (reproduced below) trivially hold as the elements I, O, T, and *gcvar* are the same of the corresponding s-DFRS, and these properties are also invariants of valid s-DFRSs.

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

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

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

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

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

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

Considering the definition of *buildTR*, the states of an e-DFRS are reachable from its initial state, which is well typed based on the definition of s-DFRSs, performing delay and function transitions. Each transition only changes the values mapped to the variables, but not the set of variables. Therefore, all states consider the same set of variables, which are all system variables. Concerning the consistency of values, the assignments performed by the transitions also need to be consistent with the variable types and, thus, this consistency is respected in all states.

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

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

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

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

3.3.3 Verifying properties of requirements via e-DFRSs

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

3.3.3.1 Consistent requirements

The system requirements are said to be consistent if, and only if, they do not describe different system reactions for the same context (state). Definition 3.3.1 formalises this concept.

Definition 3.3.1. *Consistent requirements: let reqs be an arbitrary set of requirements, and symDFRS the corresponding s-DFRS obtained via Algorithm 6; the following predicate defines* *when these requirements are said to be consistent:*

$$\begin{array}{l}
consistent(reqs) \Leftrightarrow \exists dfrs : e\_DFRS \mid dfrs = expandedDFRS(symDFRS) \bullet \\
\quad \forall s : dfrs.S \bullet (s, dfrs.I \cup dfrs.O, dfrs.T, symDFRS.functions) \notin is\_stable \Rightarrow \\
\quad \forall f1, f2 : symDFRS.F \bullet \forall e1 : f1 ; e2 : f2 \bullet \\
\qquad (s, e1.1, dfrs.I \cup dfrs.O, dfrs.T) \in static\_guards\_true \wedge \\
\qquad (s, e2.1, dfrs.I \cup dfrs.O, dfrs.T) \in static\_guards\_true \wedge \\
\qquad (s, e1.2, dfrs.T) \in timed\_guards\_true \wedge \\
\qquad (s, e2.2, dfrs.T) \in timed\_guards\_true \Rightarrow e1 = e2
\end{array}$$

According to the algorithms presented in Section 3.2, each requirement is mapped to an entry of a function. Therefore, if the requirements are consistent, for all states (s) of the e-DFRS ($dfrs$), if the guards of two entries ($e1, e2$) of two arbitrary functions ($f1, f2$) evaluate to true ($\{(..., e1.1, ...), (..., e2.1, ...)\} \subseteq static\_guards\_true$ and $\{(..., e1.2, ...), (..., e2.2, ...)\} \subseteq timed\_guards\_true$) in the same non-stable state, $(s, ...) \notin is\_stable$, these entries are the same ($e1 = e2$). In such a case, we say that the requirements are consistent. Otherwise, we would have two different system reactions for the same state. □

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

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

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

3.3.3.2 Complete requirements

The requirements are said to be complete if for every possible system input (after each delay transition), there is some system reaction (a function transition). This notion of completeness is formalised by Definition 3.3.2

Definition 3.3.2. *Complete requirements: let reqs be an arbitrary set of requirements, and symDFRS the corresponding s-DFRS obtained via Algorithm 6; the following predicate defines when these requirements are said to be complete:*

$$\begin{array}{l}
complete(reqs) \Leftrightarrow \exists dfrs : e\_DFRS \mid dfrs = expandedDFRS(symDFRS) \bullet \\
\quad \forall s1, s2 : dfrs.S \mid (\exists trns : dfrs.TR \mid trns.1 = s1 \wedge trns.3 = s2 \wedge trns.2 \in \operatorname{ran} del) \bullet \\
\qquad \exists s3 : dfrs.S ; trns2 : dfrs.TR \bullet trns2.1 = s2 \wedge trns2.3 = s3 \wedge trns.2 \in \operatorname{ran} fun
\end{array}$$

□

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

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

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

3.3.3.3 Reachable requirements

A requirement is reachable if there is a state of the e-DFRS where the guards of the function entry obtained from this requirement evaluate to true. If there is no such a state, we say that this requirement is not reachable, since it describes a system reaction that will never occur. Definition 3.3.3 defines formally the notion of reachable requirements.

Definition 3.3.3. *Reachable requirements: let reqs be an arbitrary set of requirements, and symDFRS the corresponding s-DFRS obtained via Algorithm 6; the following predicate defines when these requirements are said to be reachable:*

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

□

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

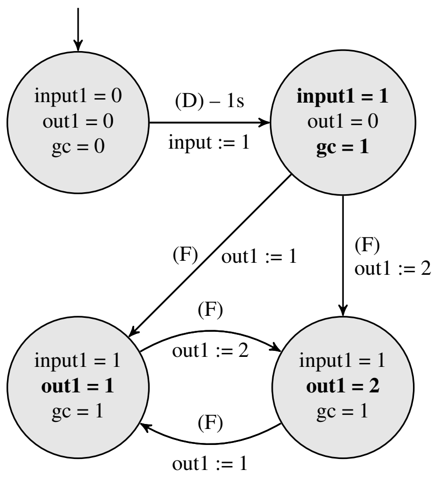
- *When input1 is true, and input1 is false, the system shall assign 1 to output1.*

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

3.3.3.4 Absence of time lock

Finally, the last property concerns the absence of time lock (see Definition 3.3.4). A time lock is characterised by a state from which it is not possible to perform delay transitions, immediately and even from states reachable by this state. If such a state exists, we have a time lock, since delay transitions cannot occur and, thus, time cannot elapse. Another way of

Figure 3.5: Example of time lock



[Source: author]

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

Definition 3.3.4. *No time lock: let reqs be an arbitrary set of requirements, and symDFRS the corresponding s-DFRS obtained via Algorithm 6; the following predicate defines when these requirements describe a system without time lock:*

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

□

To give an example of time lock, we consider the same one that was given to illustrate Definition 3.3.1. Figure 3.5 shows part of the corresponding e-DFRS. We note that when *input*1 becomes true (equal to 0), the system reaches a state from which is not possible to reach a stable state, since it can perform indefinitely function transitions. In such a situation, we say that there is a time lock.

If the specification is inconsistent or there is an unreachable requirement, we can easily identify the requirements involved as we keep traceability between the s-DFRS functions and the system requirements.

The properties defined in this section (Section 3.3.3) can be verified by exploring the e-DFRS state space. However, as an e-DFRS possibly comprises an infinite set of states, one possibility would be the specification of a bound for this check. Then, one can dynamically create an e-DFRS until this bound is reached, and, while it is created, check whether the desired properties are met (bounded model checking). Eligible criteria for this bound are the number of delay (function) transitions performed, and an upper bound for the system global clock, among others. Other possibilities would involve performing the analyses with the aid of interactive theorem provers or even by hand.

3.4 Theoretical validation

While an e-DFRS can be viewed as a semantics for the s-DFRS, from which it is obtained, in order to connect such a semantic representation to established ones in the literature, we show that an e-DFRS can be encoded as a TIOTS. This is an alternative timed model based on the widely used Input-Output Labelled Transition System (IOLTS) and ioco (TRETMANS, 1999). First, we define TIOTSs in Z (Section 3.4.1), and then we show how it can be obtained from e-DFRSs (Section 3.4.2) via a sound process (Section 3.4.3).

3.4.1 Formal model of TIOTS

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

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

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

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

TIOTS_LABELS

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

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

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

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

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

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

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

```
┌─ TIOTS_TRANSITION_RELATION ─────────────────────
│ T : TIOTS_TRANSREL
└─────────────────────────────────────────────────
```

```
┌─ TIOTS ─────────────────────────────────────────
│ TIOTS_LABELS
│ TIOTS_STATES
│ TIOTS_TRANSITION_RELATION
├──────────
│ ∀ entry : T • {entry.1, entry.3} ⊆ Q ∧ (entry.2, I, O, D) ∈ well_typed_tiots_transition
└─────────────────────────────────────────────────
```

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

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

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

3.4.2 From e-DFRSs to TIOTSs

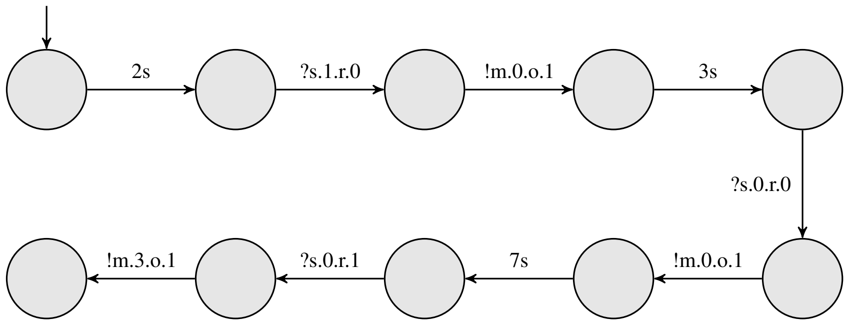
Before formalising the generation of a TIOTS from an e-DFRS, we explain the intuition behind the generation process. While a function transition is mapped to an output action, a delay transition is mapped to a delay followed by an input action. When a function transition leads to a non-stable event,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

Figure 3.6 shows the TIOTS obtained from the first five transitions presented in Figure 3.3. To differentiate input from output actions, we add “?” as a prefix to the former, and “!” to the latter. We note that the actions performed are strings that represent the value received for all system inputs or generated for all system outputs, even if the function transition does not range necessarily over the complete set of system outputs. We note that an output action is performed between the delay transitions, whose delays are 3s and 7s, to show that the system outputs remain unchanged. In this short example, we do not have τ events, as all states reached by function transitions are stable. However, considering the example shown in Figure 3.5, the corresponding TIOTS does not have any output action after the delay transition, but a loop of τ events due to the time lock.

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

The function *mapState* is a total injection from DFRS states to TIOTS ones. It is used to define the initial state (*tiots*.q_0) of the TIOTS; it is the result of this function when applied

Figure 3.6: The vending machine specification – TIOTS



[Source: author]

to the initial state of the e-DFRS. The states (*tiots*.*Q*) of a TIOTS are the ones related by its transition relation (*tiots*.*T*), which are characterised by *getStates*, besides its initial state.

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

To obtain the TIOTS transition relation, the function *mapTransitionRelation* considers two partitions of e-DFRS transitions: one with function transitions, $getTransitions(tr, \operatorname{ran} fun)$, another with delay transitions, $getTransitions(tr, \operatorname{ran} del)$. The function *getTransitions* filters function/delay transitions from a given transition relation (*tr*). The functions *mapFunTransitions* and *mapDelTransitions* consider these partitions and yield transition relations (*tr*1, and *tr*2), whose union is defined as the result of *mapTransitionRelations* ($mapTransitionRelation = tr1 \cup tr2$).

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

One important concern is related to the fresh states that are needed during this process. Therefore, we note that the third argument of *mapDelTransitions* is $\operatorname{ran} mapState$ (all TIOTS states that can be obtained from DFRS states), which is later used to identify fresh ones. For instance, one can see in Figure 3.6 that the first and third states are obtained from the first and second states in Figure 3.3, whereas the second state does not have any correspondence with a DFRS state.

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

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

The output action, $out(\ldots)$, is defined in terms of the current values of the output variables in the target state ($currentValues(\operatorname{dom} O \lhd s2)$). An example of output action is $!m.0.o.1$ (see Figure 3.6). When the function transition leads to a non-stable state, *mapFunTransitions* yields a transition relation, whose single element relates the source, $mapState(trans.1)$, and target, $mapState(trans.3)$, states with a τ event, $\{(mapState(trans.1), tau, mapState(trans.3))\}$.

The process of mapping delay transitions is more complicated due to three main reasons. First, as previously explained, we need to identify fresh states (that do not have correspondence to DFRS states); second, as also commented before, we need to define an output action between consecutive delay transitions; finally, the delay transitions that have the same amount of time elapsing are grouped into *non-time deterministic* partitions, since they have a particular treatment. To exemplify this last situation, we consider the states presented in Figure 3.4. The TIOTS transition relation obtained from this example is shown in Figure 3.7. One can see that first we have a delay of 1s leading to a state from which there are multiple possible input actions. The complete definition of how delay transitions are mapped into TIOTS transitions can be seen in Appendix B.

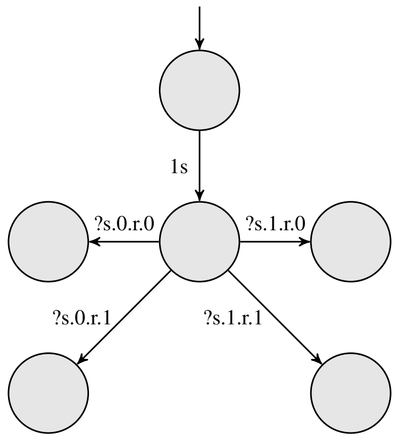
3.4.3 Soundness of mapping to TIOTS

Here we prove that the function *fromDFRStoTIOTS* is sound: for all e-DFRSs, all invariants of *TIOTS* hold in the obtained TIOTS (Theorem 3.4.1).

Theorem 3.4.1. *Soundness of fromDFRStoTIOTS*

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

Figure 3.7: The vending machine specification – TIOTS representation of delay transitions



[Source: author]

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

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

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

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

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

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

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

3.5 Concluding remarks

This chapter detailed the third phase of the NAT2TEST strategy, when a formal semantics is given for the system requirements via DFRS models. A DFRS models an embedded system whose inputs can be seen as data provided by sensors, whereas its outputs as data provided to actuators. A DFRS can also have internal timers, which might be used to trigger timed-based behaviour.

There are two representations of DFRSs: a symbolic (s-DFRS) and an expanded one (e-DFRS). The former, which is automatically derived from requirement frames, inherently avoids an explicit representation of possibly infinite sets of states and, thus, the state space explosion problem. The latter is built dynamically from its symbolic counterpart, possibly limited to some bound, and then used to bounded analyses such as requirements reachability, determinism, and completeness.

Here, we also proved by hand that an e-DFRS can be characterised as a TIOTS – a labelled transition system extended with time, which is widely used to characterise conformance relations for timed reactive systems. Being more abstract than a TIOTS, a DFRS comprises a more concise representation of timed requirements.

A classical notation for modelling timed reactive systems is timed automata. DFRS, however, is specifically designed to facilitate automatic generation of formal models from natural-language requirements. In particular, DFRS is tailored for embedded systems whose inputs and outputs are always available, as signals. The advantage of a DFRS model is the fact that, as opposed to classical timed reactive systems notations such as timed automata, it is a state-rich notation that embeds and enforces a number of properties of the models that are required of reactive embedded systems. For example, DFRS models enforce the principle of delayable transitions; use delay transitions to represent environment sitimuli and, thus, they cannot range over the output signals and timers of the system; include no self-transitions; ensure that there are no delay and function transitions emanating from any state. If we were to use general purpose notations such as timed automata to capture the natural-language requirements, the translation would be more complicated and costly.

4 A sound test strategy based on CSP

After providing a formal representation of system requirements via DFRS models, here we show how we can use the process algebra CSP as a target of such models and, thus, provide means for generating test cases (CARVALHO; SAMPAIO; MOTA, 2013b). It comprises the last two phases of the NAT2TEST$_{\text{CSP}}$ strategy: CSP generation, and generation of test cases.

It might be argued that test cases could be generated directly from DFRS models, and this is indeed the case. However, algorithms would need to be devised for that specific purpose. In particular, these algorithms would need to more elaborate than a traditional graph transversal, since e-DFRS models might comprise an infinite set of states, and it would be desirable to guide the graph transversal by test purposes (a list of observations required to be present in the test). In our CSP representation, we reuse successful and general-purpose algorithms for generating test cases.

The main challenge of encoding DFRS models as CSP processes is to achieve a CSP representation capable of dealing with discrete and continuous time, besides providing tool support for generating test cases from such a representation. We tackle this challenge by devising a symbolic time encoding, and by reusing successful and general-purpose techniques (refinement checking and SMT solving) for generating symbolic test cases. Furthermore, we prove that this testing strategy is sound with respect to a CSP timed input-output conformance relation we define, csptio.

In Section 4.1 we present the process algebra CSP, then we show how DFRSs are encoded using such notation (Section 4.2). Afterwards, we define CSP-TIO processes (Section 4.3), a central concept of our testing theory. Then, we define the csptio conformance relation (Section 4.4), which is the basis for our sound testing strategy (Section 4.6) considering test scenarios selected and generated as described in Section 4.5.

4.1 Communicating sequential processes

CSP is a formal language designed to describe behavioural aspects of systems. The fundamental element of a CSP specification is a process. CSP has two primitive processes:

one that represents successful termination (*SKIP*) and another that stands for an abnormal termination (*STOP*), also interpreted as a deadlock. In the simplest semantic model of CSP, the behaviour of a process is described by the set of sequences of events it can communicate to other processes. Events are atomic. To define a process as a sequence of events, we use the prefix process ($ev \rightarrow P$), where ev is an event and P a process. We can use the prefix operator to create an infinite (recursive) process such as $P = a \rightarrow b \rightarrow P$.

To define alternating behaviours, CSP offers (external, internal, and conditional) choice operators. An external choice ($\Box$) represents a deterministic choice between two processes, whereas the internal one ($\sqcap$) involves a non-deterministic choice. The conditional (*if*) choice operator is similar to the conditionals of standard programming languages. The *if* choice operator can also be represented as $P = Q \lhd b \rhd R$: if b is true, P behaves as Q, otherwise its behaviour is the one of R.

Two other relevant operators are the sequential and parallel composition operators. For example, the following sequential composition $P = P1\,;P2$ states that the behaviour of P is equivalent to the behaviour of $P1$, followed by the behaviour of $P2$, if and when *P1* terminates successfully.

Concerning the parallel composition, CSP allows a composition with ($\|$) or without ($|||$) synchronisation between the composed processes. CSP processes synchronise between themselves by means of events. In the composition $P \underset{X}{\|} Q$ the processes P and Q synchronise on the events in X, whereas $P\ {}_X\|_Y\ Q$ requires synchronisation on the events in $X \cap Y$.

A *channel* can be declared to denote a particular set of events. The term $c!e$, where c is a channel, denotes the event $c.e$ resulting from the evaluation of e, which is any CSP valid expression, whereas the term $c?v$ denotes any event $c.v$ where v is a value of the declared type of c. For instance, considering *channel c : {0,1,2}*, $c?v$ means *c.0*, *c.1* or *c.2*. It is also possible to interpret these symbols (*!* and *?*) as a communication sending or receiving a value through a channel, respectively.

Other two CSP operators used in this work are interrupt and hiding. The former ($\triangle$) is used to represent one process taking control of another process. For instance, $P = Q \triangle R$ means that P behaves as Q until the first external event of R is performed. The hiding operator ($\backslash$) is used to encapsulate events within a process and, thus, making them internal (represented as the special event τ). CSP also has a functional language for manipulating local data. Therefore, in this language, the state is represented by the processes parameters.

From a CSP specification written in its machine-readable version called CSP_M, the FDR tool can check desirable properties, such as: (1) deadlock-freedom, (2) divergence-freedom, (3) deterministic behaviour, and (4) refinement according to different semantic models (*traces*, *failures*, and *failures-divergences*). The first model considers the sequences of events that a process can perform. The second one augments this representation by also recording the events that a process cannot perform after a particular trace. The last one also consider divergent behaviour (when an infinite sequence of internal events can be performed – a livelock).

Our testing strategy is based on the traces semantic model. The rationale behind this assumption is that a black-box test case assesses only visible events of the SUT. Based on (ROSCOE, 2010), and presented as in (NOGUEIRA, 2012), Definition 4.1.1 characterises the traces for the CSP primitive processes and for the operators used in this work. A complete definition for all CSP operators can be found in (ROSCOE, 2010).

Definition 4.1.1. *Traces of CSP processes: let P and Q be CSP processes and Σ the set of all events these processes can perform. The traces of CSP processes are defined as follows.*

$$\mathscr{T}(SKIP) = \{\langle\rangle, \langle\checkmark\rangle\}$$
$$\mathscr{T}(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle^\frown s \mid s \in \mathscr{T}(P)\}$$
$$\mathscr{T}(P \,\Box\, Q) = \mathscr{T}(P) \cup \mathscr{T}(Q)$$
$$\mathscr{T}(P\,;Q) = (\mathscr{T}(P) \cap \Sigma^*) \cup \{s^\frown t \mid s^\frown\langle\checkmark\rangle \in \mathscr{T}(P) \wedge t \in \mathscr{T}(Q)\}$$
$$\mathscr{T}(P \lhd b \rhd Q) = \{s \mid s \in \mathscr{T}(P) \wedge eval(b)\} \cup \{t \mid t \in \mathscr{T}(Q) \wedge \neg\, eval(b)\}$$
$$\mathscr{T}(P \setminus X) = \{s \setminus X \mid s \in \mathscr{T}(P)\}$$
$$\mathscr{T}(P \triangle Q) = \mathscr{T}(P) \cup \{s^\frown t \mid s \in \mathscr{T}(P) \cap \Sigma^* \wedge t \in \mathscr{T}(Q)\}$$
$$\mathscr{T}(P \underset{X}{\parallel} Q) = \bigcup\{s \underset{X}{\parallel} t \mid s \in \mathscr{T}(P) \wedge t \in \mathscr{T}(Q)\}$$
$$\mathscr{T}(P \,{}_X\!\parallel_Y Q) = \bigcup\{s \,{}_X\!\parallel_Y t \mid s \in \mathscr{T}(P) \wedge t \in \mathscr{T}(Q)\}$$
$$\mathscr{T}(P \,|||\, Q) = \bigcup\{s \,|||\, t \mid s \in \mathscr{T}(P) \wedge t \in \mathscr{T}(Q)\}$$

Informally speaking, the traces of a process P, which is denoted as $\mathscr{T}(P)$, is the set of all possible sequences of events that the process P can perform. All processes can perform the empty trace (<>). The process $SKIP$ generates the special event $\checkmark$, which denotes successful termination, and then behaves as $STOP$, when no more events can be communicated. The traces of the prefix process ($a \rightarrow P$) includes those obtained by appending the event a to the traces of P. The traces of a sequential composition ($P\,;Q$) include the traces of P before termination, that is, without $\checkmark$, and the traces of P that denote successful termination (that is, the $\checkmark$ is the last event) followed by the traces of Q. It is worthy mentioning that the special event $\checkmark$ is removed from the traces of P as a consequence of the expression $\mathscr{T}(P) \cap \Sigma^*$, since the special event $\checkmark$ is not considered as part of the alphabet (Σ) of a process. The alphabet of a process denotes the events a process can perform.

The process "$P \lhd b \rhd Q$" represents a conditional choice operator, and the traces of this process is equal to traces of P if the condition evaluates to true, otherwise it is equal to the traces of Q. The process "$P \setminus X$" denotes hiding and, thus, the events of X are hidden within the traces of P. The process "$P \triangle Q$" denotes interruption, and the corresponding traces comprises the traces of P, as well as the traces of P, but removing the $\checkmark$ event if applicable, appended to the traces of Q.

The traces of a parallel composition is defined as the parallel composition of their respective traces. In this case, the events from the synchronization set X evolve together, whereas other events evolve independently. The traces of alphabetised parallelism $P \,{}_X\!\parallel_Y Q$ is defined as the alphabetised composition of their respective traces. In this case, the events from $X \cap Y$ evolve together, whereas P and Q evolve independently with respect to the elements of X and Y, respectively. Finally, the traces of the interleaving of two processes $P \mid\mid\mid Q$ is equal to the interleaving of the traces of P and Q.

4.2 Phase IV – encoding DFRS models as CSP processes

Now, we show how we can represent DFRS models as CSP processes. To accomplish this goal, we create processes to represent the behaviour defined by the function *genTransitions*, which was presented in Section 3.3.2. In this sense, we might say that our CSP representation takes into account an s-DFRS, along with the mechanism that generates its expanded version. A possible analogy that can be made is that a CSP specification represents an s-DFRS model, whereas the LTS defined by the operational semantics of CSP (see (ROSCOE, 2010) for more details about LTSs) corresponds to the respective e-DFRS model. Before presenting the algorithms that generate CSP specifications from DFRS models, we explain how CSP is used to encode DFRSs.

Section 4.2.1 presents an intuition of how CSP is used to represent DFRS models. Afterwards, the following sections detail the algorithms that generate this representation: how communication via shared memory (Section 4.2.2), function transitions (Section 4.2.3) and delay transitions (Section 4.2.4) are represented in CSP. Then, Section 4.2.5 shows how these algorithms are used together to generate the final CSP representation. Finally, Section 4.2.6 discusses our assumptions regarding this representation.

4.2.1 Overview of CSP representation of DFRS models

First, as a DFRS comprises input, output and timer variables, and CSP_M does not have an explicit representation of global variables (communication via shared memory), we create processes to encode such a notion. Among the different ways of representing state information (value of inputs, outputs and timers) in CSP, we adopt an alternative that creates an interleaving of processes, each one representing a single variable (memory cell). As discussed in (ROSCOE, 2010), this solution is appropriate for concurrent access. Moreover, FDR has compression algorithms suitable for minimising the state-space for such a representation. The general outline of the memory definition is:

$$
\begin{array}{l}
\textit{datatype TYPE} = \textit{type}_1 \mid \ldots \mid \textit{type}_n \\
\textit{datatype VAR} = \textit{input}_1 \mid \ldots \mid \textit{input}_i \mid \textit{output}_1 \mid \ldots \mid \textit{output}_j \mid \ldots \\
\textit{initialBinding} = \{\ (\textit{input}_1,\ \textit{value}_1),\ \ldots\ \} \\
\textit{channel get, set} : \textit{VAR.TYPE} \\
\textit{MCELL(var,val)} = \textit{get!var!val} \rightarrow \textit{MCELL(var,val)} \\
\qquad\qquad\qquad\qquad \Box\ \textit{set!var?val'} : \textit{range(tag(val))} \rightarrow \textit{MCELL(var,val')}
\end{array}
$$

MEMORY(binding) = ||| (var,val) : binding @ MCELL(var,val)
SYSTEM_MEMORY = MEMORY(initialBinding)

The datatype *VAR* represents the names of the s-DFRS variables, besides auxiliary variables. The datatype *TYPE* represents the possible values that the variables can assume (their types). Although DFRSs comprise only primitive types, the CSP specification defines a type for each variable. This is done to minimise state-explosion problems, since the memory process considers that each cell can assume any possible value of the variables' type.

Although it is possible to represent floating-point numbers in CSP, this is out of the scope of this work. Therefore, here we consider DFRS with only integer and boolean inputs and outputs. However, this restriction does not prevent us from modelling discrete and continuous time, since time is symbolically represented in CSP, as detailed later.

Considering the VM example, we have the variables and types shown in Code 4.1. Some variables (lines 14–15) are auxiliary, and explained later. The names *the_system_mode_values* and *the_coffee_machine_output_values* represent the possible values the system mode and the coffee machine output can assume, based on the enumerations identified for these variables (see Section 3.1). In CSP$_M$, -- defines a single-line comment.

Code 4.1: CSP – variables (vending machine)

```
 1 -- the_system_mode values = {0 = choice, 1 = idle,
 2 --     2 = preparing strong coffee, 3 = preparing weak coffee}
 3 the_system_mode_values = {0, 1, 2, 3}
 4
 5 -- the_coffee_machine_output values = {0 = strong, 1 = weak}
 6 the_coffee_machine_output_values = {0, 1}
 7
 8 datatype TYPE = I_the_system_mode.the_system_mode_values |
 9   I_the_coffee_machine_output.the_coffee_machine_output_values |
10   B.Bool
11
12 datatype VAR = the_coffee_request_button | the_coin_sensor |
13   the_system_mode | the_coffee_machine_output |
14   old_the_coffee_request_button | old_the_coin_sensor |
15   funTrans | the_request_timer | eta1 | eta2 | eta3 | eta4 | gc
```

[Source: author]

The memory is represented as an interleaving of memory cells (*MCELL*). Each cell represents one variable, which is initialised according to the set *binding*. The set *initialBinding* comprises pairs of variables and values, which are obtained from the initial state (s_0) of s-DFRSs.

Code 4.2 shows part of this set for the VM example. The system mode is initially 1, since this value represents the idle state within the enumeration associated with this variable. The process *SYSTEM_MEMORY* is the memory initialised considering this set.

Code 4.2: CSP – initial binding (vending machine)

```
initialBinding = {
  (the_coffee_request_button , B.false), (the_coin_sensor , B.false),
  (the_system_mode , I_the_system_mode.1),
  (the_coffee_machine_output , I_the_coffee_machine_output.0),
  ...
}
```

[Source: author]

The value stored in each cell can be read or updated by the channels *get* and *set*. Therefore, the *MCELL* process offers an external choice between these two possibilities (read/write). It is worth mentioning that it is only possible to write a value compatible with the type of the variable stored in the corresponding memory position. This restriction is imposed by the *tag* and *range* functions. These functions yield the type associated with a cell and its possible values, respectively. For the VM, the definition of these functions are shown in Code 4.3.

Code 4.3: CSP – tag and range functions (vending machine)

```
range(I_the_system_mode) =
  {I_the_system_mode.0, I_the_system_mode.1,
   I_the_system_mode.2, I_the_system_mode.3}

range(I_the_coffee_machine_output) =
  {I_the_coffee_machine_output.0, I_the_coffee_machine_output.1}

range(B) = {B.false , B.true}

tag(I_the_system_mode._) = I_the_system_mode
tag(I_the_coffee_machine_output._) = I_the_coffee_machine_output
tag(B._) = B
```

[Source: author]

Now, after explaining how communication via shared memory is represented in CSP, we proceed with the explanation of how DFRSs are encoded as CSP processes. Let *SPECIFICATION* be the CSP process that encodes the *genTransition* function, the process *SYSTEM* is defined as the parallel composition of this process with *SYSTEM_MEMORY*. This parallel composition ensures that the get and set events performed by the specification process reads and updates the memory process accordingly.

$$SYSTEM = SPECIFICATION \underset{\{|get,set|\}}{\parallel} SYSTEM\_MEMORY$$

As previously explained, an e-DFRS performs delay and function transitions. The former occurs when the system is in a stable state, whereas the latter when the state is not stable. The process *SPECIFICATION* captures this behaviour. The process *FUN* performs events that correspond to function transitions until a stable state is reached. When it happens, this process finishes successfully and, then, *SPECIFICATION* behaves as delay transitions. The event *stableState* is only a mark to indicate that a stable state was reached. Basically, time evolves (represented by the *DELAY* process, explained later) and new inputs are received (process *INPUTS*). Afterwards, it is checked whether a new stable state was reached; in CSP, we have a recursive definition for *SPECIFICATION*. The first events of this process ($set!timer_i!B.false$) are related to how time is symbolically encoded CSP, and are explained later.

$$\begin{aligned}
&SPECIFICATION = \\
&\quad set!timer_0!B.false \rightarrow ... \rightarrow set!timer_n!B.false \rightarrow \\
&\quad FUN \; ; \; stableState \rightarrow INPUTS \; ; \; DELAY \; ; \\
&\quad SPECIFICATION
\end{aligned}$$

Initially, the process *FUN* sets the auxiliary variable *funTrans* to false. This variable is used to check whether the system has successfully engaged on function transition. This analysis is performed inside the process *FUN_TRANS*. After behaving as *FUN_TRANS*, we read the current value of *funTrans*, which is stored in the local variable *engaged* (note: *B* is a type flag to denote that *engaged* keeps a boolean value). If events related to a function transition were performed, this local variable will be true. In such a situation, the process behaves as *FUN* recursively. Otherwise, *FUN* behaves as *OUTPUTS*.

$$\begin{aligned}
&FUN = set!funTrans!B.false \rightarrow FUN\_TRANS \; ; \; get!funTrans?B.engaged \rightarrow \\
&\quad if\ engaged\ then\ FUN\ else\ OUTPUTS
\end{aligned}$$

The process *OUTPUTS* reads from the memory the current value (v_i) of each system output ($output_i$), and performs a compound event combining all the values produced as output by the system at this moment. An analogy can be made between this event and the event created in TIOTSs to represent the assignments of a function transition (see Section 3.4.2). Afterwards, this process ends successfully, and, as explained before, events related to delay transitions occur.

$$\begin{aligned}
&OUTPUTS = get!output_1?v_1 \rightarrow ... \rightarrow get!output_j?v_j \rightarrow \\
&\quad output.output_1.v_1. \; ... \; .output_j.v_j \rightarrow SKIP
\end{aligned}$$

The process *FUN_TRANS* reads from the memory the current value of the system variables (inputs, outputs, and auxiliary variables), and, then, behaves as *SYSTEM_BEHAVIOUR*. That is, the process that performs events related to function transitions. We note that the values read from the memory are passed as arguments to this process.

$$\begin{aligned}
&FUN\_TRANS = \\
&\quad get!input_1?v_1 \rightarrow ... \rightarrow get!input_i?v_i \rightarrow
\end{aligned}$$

$$
\begin{aligned}
&\mathit{get!output}_1?v_{i+1} \rightarrow \ldots \rightarrow \mathit{get!output}_j?v_{i+j} \rightarrow\\
&\mathit{get!auxiliary}_1?v_{i+j+1} \rightarrow \ldots \rightarrow \mathit{get!auxiliary}_k?v_{i+j+k} \rightarrow\\
&\mathit{SYSTEM\_BEHAVIOUR}(v_1, \ldots\ v_i,\ v_{i+1}, \ldots\ v_{i+j},\ v_{i+j+1}, \ldots\ v_{i+j+k})
\end{aligned}
$$

The process *SYSTEM_BEHAVIOUR* is defined as follows.

$$
\begin{aligned}
&\mathit{SYSTEM\_BEHAVIOUR}(\ldots) =\\
&\quad (\mathit{static\_guard}_1 \text{ and } \mathit{eta}_1 \text{ and } \mathit{collateral\_effect}_1\ \&\\
&\qquad \mathit{set!funTrans!B.true} \rightarrow \mathit{assignments}_1 \rightarrow \mathit{REQ}_1 \rightarrow \mathit{SKIP})\\
&\quad \Box \ldots \Box\\
&\quad (\mathit{static\_guard}_m \text{ and } \mathit{eta}_m \text{ and } \mathit{collateral\_effect}_m\ \&\\
&\qquad \mathit{set!funTrans!B.true} \rightarrow \mathit{assignments}_m \rightarrow \mathit{REQ}_m \rightarrow \mathit{SKIP})\\
&\quad \Box\\
&\quad (\mathit{not}(\mathit{guard}_1 \text{ and } \ldots \text{ and } \mathit{guard}_m)\ \&\ \mathit{SKIP})
\end{aligned}
$$

The *SYSTEM_BEHAVIOUR* process has an external choice for each entry of each function of the s-DFRS (see Section 3.1.2.3). The static guard of the entry (*static_guard$_i$*) is translated to the CSP syntax, whereas a (boolean) flag is created for the timed guard (*eta$_i$*). Briefly, this flag is true when the corresponding timed guard should evaluate to true, and false otherwise. This flag is explained later, when presenting how time is symbolically encoded in CSP. Besides these two expressions, there is a third one (*collateral_effect$_i$*), since according to the definition of stable states (see the definition of *is_stable* in Section 3.3.2) a function transition is only performed when the static and timed guards evaluate to true, but the corresponding assignments of the transition have a collateral effect (change the value of at least one variable and, thus, lead to a new state).

To give a concrete example, we consider the requirement of the VM that states that the system goes to the *choice* mode, and resets the *request timer*, when a coin is inserted while in the state *idle*. As shown in Section 3.1.2.3, this requirement (REQ001) is formalised as follows.

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

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

In CSP, this entry is represented as follows. Note that [ ] is the CSP$_M$ version of $\Box$, and that *condition & P* is a shortcut to *if condition then P else STOP* (ROSCOE, 2010).

Code 4.4: CSP – SYSTEM_BEHAVIOUR fragment (vending machine)

```
SYSTEM_BEHAVIOUR ( . . . )  =
2   (not(v_old_the_coin_sensor == true) and
3    v_the_coin_sensor == true and v_the_system_mode == 1 and
4    (v_the_request_timer != true or v_the_system_mode != 0) &
5     set!funTrans!B.true -> reset.the_request_timer ->
6     set!the_request_timer!B.true ->
7     set!the_system_mode!I_the_system_mode.0 ->
8     REQ001 -> SKIP)
9   [] ...
```

[Source: author]

Lines 2–3 represent the static guard: the current value of the coin sensor is true, but its previous value was false (note that *v_old_...* is an auxiliary variable declared in the CSP to store the previous value of the coin sensor), and the system mode is 1. As this requirement does not have a timed guard, no flag is generated. The expression that captures the collateral effect can be seen in line 4. The effect of this requirement is assigning 0 to the system mode, besides resetting the request timer. Therefore, the corresponding function transition occurs if the current system mode is not 0 or the request timer was not already reset. Here, as time is symbolically represented, other auxiliary variable (*v_the_request_timer*) is created to store that this timer was already reset.

If the e-DFRS function transition occurs, in the CSP model we set *funTrans* to true to record that a function transition was performed (see the previous explanation of how we encode in CSP the idea of stable states), besides performing the transition collateral effect: setting the request timer flag to true, and changing the system mode to 0. The events *reset.the_request_timer* and *REQ*001 are marks used later. For instance, the event *REQ*001 allows us to generate test scenarios that necessarily cover the particular requirement *REQ*001 (an interesting coverage criteria).

Besides having external choices for every entry of each function of the s-DFRS, there is an additional external choice (the last external choice of the process *SYSTEM_BEHAVIOUR*). This choice states that, if all other guards evaluate to false, no function transition is performed. Therefore, the guard of this last choice is the negation of the conjunction of all other guards ($not(guard_1\ and\ ...\ and\ guard_m)$; $guard_i$ denotes ($static\_guard_i\ and\ eta_i\ and\ collateral\_effect_i$). If this condition is met, the variable *funTrans* is not set to true. In other words, we have a record that no function transition was performed.

To conclude our explanation of how we represent DFRSs in CSP, we need to show how delay transitions are represented. As explained in Section 3.3, a delay transition represents model stimuli from the environment (inputs) that happen immediately after a delay. The process *INPUTS*, mentioned before, represents these stimuli.

$INPUTS =$

$\quad c\_input_1?new\_v_1 \rightarrow get!input_1?current\_v_1 \rightarrow$

$\quad set!old\_input_1!current\_v_1 \rightarrow set!input_1!new\_v_1 \rightarrow$

$$\begin{aligned}
&\ldots\\
&c\_input_i?new\_v_i \rightarrow get!input_i?current\_v_i \rightarrow\\
&set!old\_input_i!current\_v_i \rightarrow set!input_i!new\_v_i \rightarrow\\
&input.input_1.new\_v_1.\ \ldots\ .input_i.new\_v_i \rightarrow SKIP
\end{aligned}$$

For every system input ($input_i$), first, we obtain a new value from its possible ones. This action is performed reading a new value (new_v_i) from a channel declared for each input (c_input_i), which ranges over the possible values of each system input. Afterwards, we read the current value of the input ($current\_v_i$), and store this value in the auxiliary variable created to keep the previous value of this input (old_input_i). Then, we update the input value considering the new obtained value ($set!input_i!new\_v_i$). Finally, similarly to the process *OUTPUTS*, we perform a compound event communicating the values read as input by the system at this moment. An analogy can be made between this event and the event created in TIOTSs to represent the assignments of a delay transition (see Section 3.4.2).

A concrete example of this process can be seen in Code 4.5 for the VM example. Lines 2–5 obtains a new value for the coffee request button, copies the current value to the corresponding old variable, and then updates the current value of this variable. Lines 6–9 are similar, but regarding the coin sensor. Finally, lines 10–11 show the compound event mentioned in the last paragraph.

Code 4.5: CSP – process INPUTS (vending machine)

```
1  INPUTS =
2    c_the_coffee_request_button?newV_the_coffee_request_button ->
3    get!the_coffee_request_button?B.v_the_coffee_request_button ->
4    set!old_the_coffee_request_button!B.v_the_coffee_request_button ->
5    set!the_coffee_request_button!B.newV_the_coffee_request_button ->
6    c_the_coin_sensor?newV_the_coin_sensor ->
7    get!the_coin_sensor?B.v_the_coin_sensor ->
8    set!old_the_coin_sensor!B.v_the_coin_sensor ->
9    set!the_coin_sensor!B.newV_the_coin_sensor ->
10   input.the_coffee_request_button.B.newV_the_coffee_request_button
11        .the_coin_sensor.B.newV_the_coin_sensor -> SKIP
```

[Source: author]

As already mentioned, we support both discrete and continuous time by capturing time aspects in a CSP process symbolically. Concrete values are given later by an SMT solver, if one desires to generate concrete test cases (see Section 5.2.5). To analyse the system behaviour according to a discrete time model, we just need to configure the solver to work with integer numbers. However, to generate concrete test cases within a continuous-time domain, the solver needs to consider real numbers. The process *DELAY* captures this symbolic representation.

$$\begin{aligned}
&DELAY =\\
&\quad set!eta_1!B.false \rightarrow \ldots \rightarrow set!eta_m!B.false \rightarrow
\end{aligned}$$

$$
\begin{array}{l}
\mathit{get!input_1?v_1} \rightarrow \ldots \rightarrow \mathit{get!input_i?v_i} \rightarrow \\
\mathit{get!output_1?v_{i+1}} \rightarrow \ldots \rightarrow \mathit{get!output_j?v_{i+j}} \rightarrow \\
\mathit{get!auxiliary_1?v_{i+j+1}} \rightarrow \ldots \rightarrow \mathit{get!auxiliary_k?v_{i+j+k}} \rightarrow \\
(\\
\quad (\mathit{static\_guard_1}\ \&\ \mathit{set!eta_1B.true} \rightarrow \mathit{SKIP}) \\
\quad \Box \ldots \Box \\
\quad (\mathit{static\_guard_m}\ \&\ \mathit{set!eta_mB.true} \rightarrow \mathit{SKIP}) \\
\quad \Box \\
\quad (\mathit{not(static\_guard_1\ and \ldots and\ static\_guard_m)}\ \&\ \mathit{SKIP}) \\
) \\
;\ \mathit{get!eta_1?v_1} \rightarrow \ldots \rightarrow \mathit{get!eta_m?v_m} \rightarrow \\
\mathit{delayChannel.eta_1.v_1.\ \ldots\ .eta_m.eta_m} \rightarrow \mathit{SKIP}
\end{array}
$$

For each timed guard we introduce an auxiliary variable named eta_i. Initially, this process sets false to all eta_i variables and reads from the memory the value of system variables (inputs, outputs and other auxiliary variables, such as variables keeping old values). Afterwards, this process analyses if the system is in a state where a static guard ($static\_guard_i$) is true and there is a corresponding timed guard. If this is the case, it means that the amount of elapsed time influences if this transition is taken or not. Due to the underlying delayable assumption (see Section 3.3.2), we say that some amount of time has elapsed that makes this guard true. Then we set to true the corresponding eta_i variable. Therefore, when the process *SYSTEM_BEHAVIOUR* evaluates this condition, as the flag will be true, the corresponding statements will be performed.

The concrete amount of elapsed time is determined later by the solver, when generating concrete test cases (Section 5.2.5). However, this amount respects the temporal constraints associated with eta_i (Section 4.4 details how this is achieved by the solver). Therefore, we keep a traceability between the *eta* variables and timed guards. For instance, Code 4.6 shows this mapping for the VM example (embedded as comments). We note that *gc* stands for the system global clock, and that the timed guards are shown in prefix notation (a standard format for SMT solvers).

Code 4.6: Traceability between eta variables and timed guards (vending machine)

```
-- eta1 |-> (> (- gc the_request_timer) 30.0)
-- eta2 |-> (AND (<= (- gc the_request_timer) 50.0)
                 (>= (- gc the_request_timer) 30.0))
-- eta3 |-> (<= (- gc the_request_timer) 30.0)
-- eta4 |-> (AND (<= (- gc the_request_timer) 30.0)
                 (>= (- gc the_request_timer) 10.0))
```

[Source: author]

If the system is in a state where the amount of elapsed time has no influence (the negation of the conjunction of all static guards), there is no relevant time constraint, and no *eta* variable is set to true. This behaviour is captured by the last external choice.

Finally, after reading the updated value of the *eta* variables, similarly to *INPUTS* and *OUTPUTS*, the *DELAY* process performs a compound event considering the value of these variables. We leave the explanation of the need of this compound event to when we explain our CSP timed input-output conformance relation (Section 4.4).

Code 4.7 shows a fragment of the *DELAY* process for the VM example.

Code 4.7: CSP – fragment of *DELAY* (vending machine)

```
 1 DELAY =
 2   set!eta4!B.false -> set!eta3!B.false ->
 3   set!eta2!B.false -> set!eta1!B.false ->
 4   ... ->
 5   (
 6     ...
 7     []
 8     ((v_the_system_mode == 2) & set!eta2!B.true -> SKIP)
 9     []
10     ...
11     []
12     ((v_the_system_mode == 3) & set!eta4!B.true -> SKIP)
13     []
14     (not(...) & SKIP)
15   ) ; ...
16   delay.eta1.v_eta1.eta2.v_eta2.eta3.v_eta3.eta4.v_eta4 ->
17   SKIP
```

[Source: author]

The external choices highlighted (lines 8, 12) correspond to the situations when the system produces strong (the system mode is 2) and weak (the system state is 3) coffee, respectively. In the former situation, the *eta2* variable is set to true (see in Code 4.6 that it is mapped to the time required for producing strong coffee – from 30.0 to 50.0 time units), whereas in the latter situation, the *eta4* variable is set to true (see in Code 4.6 that it is mapped to the time required for producing weak coffee – from 10.0 to 30.0 time units). Appendix C shows the complete CSP specification for the VM example. In the following sections, we present the algorithms that generate a CSP specification from DFRS models. Afterwards, we discuss assumptions of this representation.

4.2.2 Creating memory representation

Algorithm 7 generates two CSP processes (*MCELL* and *MEMORY* – lines 5–10) to simulate the memory where the value of the system inputs and outputs are stored. Before generating these processes, some auxiliary code is also generated (lines 1–3). This code is also part of the CSP memory definition (lines 1–4). In the end, a CSP process is created considering an instance of the memory based on the initial binding (line 5–10).

Algorithm 7: *generateMEMORY* – generates process SYSTEM_MEMORY

```
input  : $dfrs$
output : $memoryCSP, etaBinding, varList$
1  $valuesCSP = generateValues(dfrs)$;
2  $typesCSP = generateTypes(dfrs)$;
3  $variablesCSP, bindingCSP, etaBinding, varList = generateVarBinding(dfrs)$;
4  $memoryCSP = valuesCSP + typesCSP + variablesCSP + bindingCSP$;
5  $memoryCSP = memoryCSP +$
6      “channel get, set : VAR.TYPE
7      MCELL(var,val) = get!var!val -> MCELL(var,val)
8                       [] set!var?val’ : range(tag(val)) -> MCELL(var,val’)
9      MEMORY(binding) = ||| (var,val) : binding @ MCELL(var,val)
10     SYSTEM_MEMORY = MEMORY(initialBinding)”;
```

[Source: author]

Algorithm 8 generates a set of possible values concerning the system inputs and outputs that are used later to define types. Besides that, it also generates the code for the following functions in CSP: *range* and *tag*. The former yields a set of possible values with a tag preceding each value, and the latter yields a tag for each type (boolean or integer types that consider the previous identified set of possible values instead of the full range of integer numbers – it is necessary to minimise state explosion problems). We note here that, although the formal definition of DFRSs does not have this information of possible values of variables, the NAT2TEST tool (see Chapter 5) keeps this information, besides allowing the user to edit it (adding new possible values).

Line 1 (Algorithm 8) initialises the return variable. A different set of possible values is generated for each input and output that are not boolean (lines 2–5). As boolean is a pre-defined type in CSP, we do not need to create a set to enumerate its possible values. Currently, we do not support float numbers in CSP (lines 17–18). Therefore, we just define the functions *range* and *tag* concerning boolean values (lines 19–20).

If there is an input or output that is an integer (line 5), we create a set that comprises the list of possible values of this variable (lines 6–13). The name of this set is the variable name appended to *I_*, which denotes it is an integer (line 6). Then, we define the functions *range* and *tag* concerning this set (lines 14–16). Regarding the VM, the output produced by the execution of Algorithm 8 is the first six lines shown in Code 4.1, besides the definitions presented in Code 4.3.

Algorithm 9 defines a new datatype *TYPE* that comprises integer types considering the previously identified set of possible values instead of the full range of integer numbers (lines 6–10); as well as a boolean type (lines 11–12). It is worth emphasising that each type is identified by its respective tag: *B* for booleans or *I_*†, where † represents the name of one specific variable. For the VM, the output of this algorithm are the lines 8–10 of Code 4.1.

Algorithm 10 generates the list of variables that are going to be stored in the memory (system inputs and outputs, besides auxiliary variables), an initial binding to these variables, and

Algorithm 8: *generateValues* – generates a list of possible values

```
input  : $dfrs$
output : $valuesCSP$
1  $valuesCSP = new\ String()$;
2  for $var \in dfrs.I, dfrs.O$ do
3      if $var.type = boolean$ then
4          $continue$;
5      else if $input.type = integer$ then
6          $tagName =$"I_" $+\ var.name$;
7          $listOfValues = null$;
8          $rangeOfValues = null$;
9          for $value \in input.possibleValuesList$ do
10             if $listOfValues = null$ then $listOfValues = value$;
11             else $listOfValues = listOfValues +$ ", " $+\ value$;
12             if $rangeOfValues = null$ then $rangeOfValues = tagName +$ "." $+\ value$;
13             else $rangeOfValues = rangeOfValues +$ ", " $+\ tagName +$ "." $+\ value$;
14         $valuesCSP = valuesCSP + var.name +$ "_values = {" $+\ listOfValues +$ "}";
15         $valuesCSP = valuesCSP +$ "range(" $+\ tagName +$ ") = {" $+\ rangeOfValues +$ "}";
16         $valuesCSP = valuesCSP +$ "tag(" $+\ tagName +$ "._) = " $+\ tagName$;
17     else
18         $throw\ Exception($"float numbers are not yet supported in the CSP level"$)$;
19 $valuesCSP = valuesCSP +$ "range(B) = {B.false, B.true}";
20 $valuesCSP = valuesCSP +$ "tag(B._) = B";
```

[Source: author]

Algorithm 9: *generateTypes* – generates CSP types

```
input  : $dfrs$
output : $typesCSP$
1  $typesCSP = null$;
2  for $var \in dfrs.I, dfrs.O$ do
3      if $var.type = boolean$ then
4          $continue$;
5      else
6          $tagName =$"I_" $+\ var.name$;
7          if $typesCSP = null$ then
8              $typesCSP =$"datatype TYPE = " $+\ tagName +$ "." $+\ var.name +$ "_values";
9          else
10             $typesCSP = typesCSP +$ " | " $+\ tagName +$ "." $+\ var.name +$ "_values";
11 if $typesCSP = null$ then $typesCSP =$"datatype TYPE = B.Bool" ;
12 else $typesCSP = typesCSP +$ " | B.Bool" ;
```

[Source: author]

a mapping between the DFRS timed guards and the *eta* variables, which are created to represent these guards symbolically. This mapping will be used later. A list of the variables contained in the memory is also a return of this algorithm. This list is used to help the implementation of other algorithms, as explained later.

Lines 1–3 adds to the list of variables some auxiliary variables. Lines 4–5 adds to this list the system inputs, outputs, and timers. Lines 6–17 iterate over the list of functions (line 7) and the entries *(static guard* $\times$ *timed guard* $\rightarrow$ *list of assignments)* of each function (line 8) to perform two tasks.

The first task (lines 9–13) concerns adding to the list of variables a new *eta* variable for each timed guard. It also adds to the *etaBinding* a mapping between the created *eta* variabled and the corresponding guard. The second task (lines 14–17) is related to use of *prev* in the guards. When it happens, we create an *old* variable for each variable associated with a *prev*.

Lines (18–34) iterate over the list of variables identified, and defines an initial binding for them: *0* for integers and *false* for boolean variables. These values are preceded by their corresponding tag. In the end (lines 35–36), the CSP code related to the variables and the initial binding is created. For the VM, this is code shown in Code 4.1 (lines 12–15) and Code 4.2. Now, after creating the memory representation in CSP, we proceed with the explanation of how function and delay transitions of DFRSs are encoded in CSP.

4.2.3 Encoding function transitions

As explained in the beginning of Section 4.2, function transitions are represented by the CSP process *FUN*. It performs events that correspond to function transitions until a stable state is reached. Algorithm 11 yields the body of the *FUN* process, as well as list of local variables passed as arguments to *SYSTEM_BEHAVIOUR*. This is used to help the generation of the process *SYSTEM_BEHAVIOUR*.

Lines 1–3 generate the code of process *FUN*, besides initialising the variable *localVarList*. In what follows, lines 4–14 create the code of the process *FUN_TRANS*. First, the value of the variables are read from the memory (lines 5–9) and stored in local variables (preceded by *v_*). The value of the variables *gc* and *funTrans* are not relevant here and, thus, are not considered. Then, the process *FUN_TRANS* behaves as *SYSTEM_BEHAVIOUR* passing the created local variables as arguments.

Algorithm 12 generates the process *SYSTEM_BEHAVIOUR*. Lines 1–2 create auxiliary variables for this algorithm. Lines 3–7 create the signature of this process considering the list of local variables that are passed as argument to it.

Then, for each function (line 8), the algorithm generates an external choice (lines 9–15) for each entry *(staticGuard* $\times$ *timedGuard* $\rightarrow$ *actionList)*. This choice is guarded by the following condition (guard): the static guard (line 14), the *eta* variable related to the timed guard (lines 16–19) are both true (line 20), besides being true that the assignments contained in the

Algorithm 10: *generateVarBinding* – generates CSP variables and initial binding

```
input  : $dfrs$
output : $variablesCSP, bindingCSP, etaBinding, varList$
$varListiableList = new\ List()$;
$varList.add($“gc”$)$;
$varList.add($“funTrans”$)$;
for $var \in dfrs.I, dfrs.O, dfrs.T$ do
    if $varList.find(var.name) = null$ then $varList.add(var.name)$;
$timedGuardCounter = 1$;
for $f \in dfrs.F$ do
    for $(staticGuard, timedGuard, actionList) \in f$ do
        if $timedGuard \neq null$ then
            $etaName =$“eta” $+\ timedGuardCounter$;
            $varList.add(etaName)$;
            $etaBinding.put(timedGuard, etaName)$;
            $timedGuardCounter = timedGuardCounter + 1$;
        $list = \{varName \mid staticGuard.contains($“prev(” $+\ varName +$ “)”$)\}$;
        for $varName \in list$ do
            if $varList.find(varName) = null$ then
                $varList.add($“old_” $+\ varName)$;
$variables = new\ String()$;
$binding =$“(funTrans, B.false)”;
for $varName \in varList$ do
    if $varList.getIndex(varName) \neq 0$ then $variables = variables +$ “ | ” ;
    $variables = variables + varName$;
    $binding = binding +$ “, ”;
    $auxName = varName$;
    if $auxName.startsWith($“old_”$)$ then $auxName = auxName.remove($“old_”$)$ ;
    $initialValue = null$;
    if $auxName.startsWith($“eta”$) \lor dfrs.T.find(auxName) != null$ then
        $initialValue =$“B.false”;
    else
        $var = dfrs.I.find(auxName)$;
        if $var = null$ then $var = dfrs.O.find(auxName)$;
        if $var.type = integer$ then $initialValue =$“I_” $+\ input.name +$ “.0” ;
        else $initialValue =$“B.false” ;
    $binding = binding +$ “(” $+\ varName +$ “, ” $+\ initialValue +$ “)”;
$variablesCSP =$“datatype VAR = ” $+\ + variables$;
$bindingCSP =$“initialBinding = {” $+\ binding +$ “}”;
```

[Source: author]

Algorithm 11: *generateFUN* – generates processes FUN and FUN_TRANS

```
input  : $dfrs, varList$
output : $funCSP, localVarList$
1  $funCSP =$ “set!funTrans!B.false -> FUN_TRANS ; get!funTrans?B.engaged”;
2  $funCSP = funCSP +$ “ -> if engaged then FUN else OUTPUTS”;
3  $localVarList = new\ List()$;
4  $funCSP = funCSP +$ “FUN_TRANS =”;
5  for $var \in varList \wedge !var.name.equal(“gc”) \wedge !var.name.equal(“funTrans”)$ do
6      $funCSP = funCSP +$ “get! $+ var.name$;
7      $funCSP = funCSP +$ “?” $findTag(var, dfrs) +$ “v_” $+ var.name$;
8      $funCSP = funCSP +$ “ -> ”;
9      $localVarList.add($“v_” $+ var.name)$;
10 $funCSP = funCSP +$ “SYSTEM_BEHAVIOUR(”;
11 for $var \in localVarList$ do
12     if $localVarListist.indexOf(var)! = 0$ then $funCSP = funCSP +$ “, ”;
13     $funCSP = funCSP + var$;
14 $funCSP = funCSP +$ “)”;
```

[Source: author]

action list have some collateral effect (line 15). If this condition is fulfilled, the CSP auxiliary variable is set to true (line 23), the actions in *actionList* (lines 24–36) are performed, and then the CSP process performs an event to indicate which requirement is originally associated to this behaviour (lines 37–39). This event is used later to select test scenarios (see Section 4.5). Line 41 generates the CSP code that declare these events.

It is worth noting three details. When the assignment in the action list comprises a variable that has an old counterpart, before updating its value, we copy to its old version (*old_*) its current value (lines 32–34). Second, we group in *combinedNegation* the conjunction of the negation of each CSP guard. This is done to create a last external choice, when no associated actions are performed (line 40). Third, concerning actions related to timers, that is reset operations, we generate a communication over the channel reset (lines 26–28), since time is just symbolically modelled in CSP. Line 42 generates the CSP code that declares these reset events. Concerning the VM example, Code 4.4 shows a fragment of the CSP process generated by this algorithm.

Algorithm 12 relies upon some auxiliary functions: *findTag*, *changeFormat*, *genCollateralGuard*, *genReqEvents*, *getResetEvents*. The definition of these functions are not presented here as they are straightforward: *findTag* yields the tag (*I_* or *B_*) associated to a given variable, *changeFormat* changes the format of conditional expression from the DFRS notation to CSP, *genCollateralGuard* generates a conditional expression that ensures that the list of actions has a collateral effect (i.e., the value of at least one variable is going to be changed), *genReqEvents* and *genResetEvents* generate the CSP code that declares the requirement and reset-related events, respectively.

To finish our encoding of s-DFRSs as CSP, Algorithm 13 generates the code of the pro-

Algorithm 12: *generateBEHAVIOUR* – generates process SYSTEM_BEHAVIOUR

```
    input  : dfrs, localVarList, etaBinding
    output : behaviourCSP
 1  reqList, resetList = new List();
 2  behaviourCSP = new String();
 3  behaviourCSP = "SYSTEM_BEHAVIOUR(";
 4  for var ∈ localVarList do
 5      if localVarList.indexOf(var) != 0 then behaviourCSP = behaviourCSP + ", ";
 6      behaviourCSP = behaviourCSP + var;
 7  behaviourCSP = behaviourCSP + ") =";
 8  for f ∈ dfrs.F do
 9      firstCondition = true;
10      combinedNegation = null;
11      for (staticGuard, timedGuard, actionList) ∈ f do
12          if firstCondition then firstCondition = false;
13          else behaviourCSP = behaviourCSP + "[]";
14          guard = changeFormat(staticGuard);
15          collateral = genCollateralGuard(actionList);
16          if timedGuard ≠ null then
17              etaName = etaBinding.find(timedGuard);
18              etaName = "v_" + etaName;
19              guard = guard + " and " + etaName;
20          guard = guard + " and " + collateral;
21          if combinedNegation = null then combinedNegation = "not(" + guard + ")" ;
22          else combinedNegation = combinedNegation + "and not(" + guard + ")" ;
23          behaviourCSP = behaviourCSP + "(" + guard + " & set!funTrans!B.true -> ";
24          for action ∈ actionList do
25              varName = action.before(":=");
26              if dfrs.T.find(varName) ≠ null then
27                  behaviourCSP = behaviourCSP + "reset." + varName + " -> ";
28                  resetList.add(varName);
29              else
30                  value = action.after(":=");
31                  tagName = findTagName(varName, dfrs);
32                  if localVarList.indexOf("v_old_" + varName) != 0 then
33                      behaviourCSP = behaviourCSP + "set!old_" + varName + "!"
34                          + tagName + ".v_" + varName + " -> ";
35                  behaviourCSP = behaviourCSP + "set!"
36                      + varName + "!" + tagName + "." + value + " -> ";
37          reqID = f.getRequirementID((staticGuard, timedGuard, actionList));
38          reqList.add(reqID);
39          behaviourCSP = behaviourCSP + reqID + " -> SKIP)";
40      behaviourCSP = behaviourCSP + "[] (" + combinedNegation + " & SKIP)";
41  behaviourCSP = behaviourCSP + genReqEvents(reqList);
42  behaviourCSP = behaviourCSP + genResetEvents(resetList);
```

[Source: author]

cess *OUTPUTS*, which performs an output event when a stable state is reached. After declaring the channel *output* (lines 1–4), the value of the output variables are read from the memory and kept on local variables (lines 6–7). Then, this process performs the output event (lines 8–10).

Algorithm 13: *generateOUTPUTS* – generates process OUTPUTS

```
input  : $dfrs$
output : $outputsCSP$
1  $outputsCSP$ =“channel output : ”;
2  for $var \in dfrs.O$ do
3      if $dfrs.O.indexOf(var)! = 0$ then $outputsCSP = outputsCSP +$ “.”;
4      $outputsCSP = outputsCSP +$ “VAR.TYPE”;
5  $outputsCSP = outputsCSP +$ “OUTPUTS =”;
6  for $var \in dfrs.O$ do
7      $outputsCSP = outputsCSP +$ “get!” $+ var.name +$ “?v_” $+ var.name +$ “ ->″;
8  $outputsCSP = outputsCSP +$ “output”;
9  for $var \in dfrs.O$ do $outputsCSP = outputsCSP +$ “.” $+ var.name +$ “v_” $+ var.name$ ;
10 $outputsCSP = outputsCSP +$ “ -> SKIP”;
```

[Source: author]

4.2.4 Encoding delay transitions

As explained in the beginning of Section 4.2, delay transitions are represented by the CSP processes *INPUTS* and *DELAY*. To generate these two processes we rely on two algorithms (*generateINPUTS* and *generateDELAY*) whose body is similar to the ones described in Algorithm 13 and in Algorithm 12. Due to the similarity, we omit them here, but provide only an explanation of the differences

The differences with respect to the *INPUTS* are described in what follows. Instead of declaring the channel *output*, it creates the channel *input*, considering the number of system inputs. This channel is used to communicate the new values received as input. Therefore, before performing the communications over the channel *input*, we generate the code that simulates the receiving of new input values: an additional channel is created for each system input, considering its list of possible values, and a value is read from this channel. Before saving the new read value into the memory, we copy the current value of the system inputs to its old counterparts, if they exist. Code 4.5 shows the code generated by *generateINPUTS* for the VM example.

The generation of *DELAY* starts by generating the code that sets all *eta* variables to false. Then, as in *SYSTEM_BEAHAVIOUR*, we have external choices created after the entries of the DFRS functions. The difference here is that the guard of each external choice considers only the static guard. Moreover, the body of each choice is not derived from the list of actions, but it sets the *eta* variable mapped to timed guard associated with the considered static guard. Finally, after reading the updated values of the *eta* variables, it communicates over the channel *delay* these values. Therefore, it also declares this channel *delay*. The output of *generateDELAY* with respect to the VM example is partially shown in Code 4.7.

4.2.5 Creating a CSP specification

To finish the generation of the CSP representation of s-DFRSs, Algorithm 14 generates the definition of *SPECIFICATION*: a process that sets to false the auxiliary variables created after the DFRS timers (lines 1–3), and, then, behaves as *FUN*, followed by *DELAY* and *INPUTS* when a stable state is reached. Afterwards, it behaves recursively as *SPECIFICATION* (lines 4–5). Finally, this algorithm generates the code of the process *SYSTEM* (lines 6–7), which composes *SPECIFICATION* in synchronous parallelism with the *SYSTEM_MEMORY*.

Algorithm 14: *generateSPEC* – generates process SPECIFICATION

```
input  : $dfrs$
output : $specificationCSP$
1 $specificationCSP$ =“SPECIFICATION = ”;
2 for $var \in dfrs.T$ do
3     $specificationCSP = specificationCSP +$ “set!” $+ var.name +$ “B.false ->”;
4 $specificationCSP = specificationCSP +$ “ FUN ; stableState ->” $+$
5     “INPUTS ; DELAY ; SPECIFICATION”;
6 $specificationCSP = specificationCSP +$ “SYSTEM = SPECIFICATION”
7     “[| {| get, set |} |] SYSTEM_MEMORY”;
```

[Source: author]

Algorithm 15: *generateCSP* – creates a CSP specification

```
input  : $dfrs$
output : $csp$
1 $memoryCSP, etaBinding, varList = generateMEMORY(dfrs)$;
2 $funCSP, localVarList = generateSystemBehaviours(dfrs, varList)$;
3 $behaviourCSP = generateBEHAVIOUR(dfrs, localVarList, etaBinding)$;
4 $outputsCSP = generateOUTPUTS(dfrs)$;
5 $inputsCSP = generateINPUTS(dfrs, varList)$;
6 $delayCSP = generateDELAY(dfrs, etaBinding)$;
7 $specificationCSP = generateSPEC(dfrs)$;
8 $csp = memoryCSP + funCSP + behaviourCSP + outputsCSP +$
9     $inputsCSP + delayCSP + specificationCSP$;
```

[Source: author]

Using the previously defined functions, Algorithm 15 creates a CSP specification to simulate the corresponding DFRS.

4.2.6 Assumptions of CSP representation

To represent DFRSs as CSP processes, some premises shall hold with respect to the DFRS models. They are necessary due to our symbolic representation of time. Now, we present and discuss them.

In non-stable states, considering the entries of the s-DFRS functions whose static guard is true, only one distinct timed guard can be true (Definition 4.2.1). To understand this restriction, remember that we use *eta* variables to represent that a particular timed guard was satisfied by a delay, and that only one *eta* variable can be set to true per execution of the CSP process *DELAY*. Therefore, if a delay could satisfy more than one distinct timed guard, our CSP process would not model properly the DFRS, since only one *eta* variable would be set to true.

Definition 4.2.1. *No more than one timed guard evaluates to true: let sdfrs be an s-DFRS, and edfrs the e-DFRS obtained from sdfrs by the application of expandedDFRS, the following property shall hold when using CSP to represent DFRS models:*

$$
\begin{array}{l}
\forall s : edfrs.S \mid (s, edfrs.I \cup edfrs.O, edfrs.T, sdfrs.F) \notin is\_stable \bullet \\
\quad \#\{f : sdfrs.F;\, sG, tG : EXP;\, asgmts : ASGMTS \mid (sG, tG, asgmts) \in f \land \\
\qquad (s, sG, edfrs.I \cup edfrs.O, edfrs.T) \in static\_guards\_true \land \\
\qquad (s, tG, edfrs.T) \in timed\_guards\_true \bullet tG\} = 1
\end{array}
$$

A second premise is that the DFRS comprises only satisfiable timed guards (Definition 4.2.2). Being more precise, for all stable states where a given static guard is true, but its timed guard is false, there shall exist some delay that satisfies this timed guard. If this property does not hold, in the CSP, we would set true to an *eta* variable, but in practice the constraint associated with this variable is not satisfiable. Therefore, our CSP representation would not be an appropriate representation of the DFRS.

Definition 4.2.2. *Satisfiable timed guards: let sdfrs be an s-DFRS, and edfrs the e-DFRS obtained from sdfrs by the application of expandedDFRS, the following property shall hold when using CSP to represent DFRS models:*

$$
\begin{array}{l}
\forall s : edfrs.S;\, f : sdfrs.F;\, sG, tG : EXP;\, asgmts : ASGMTS \mid (sG, tG, asgmts) \in f \land \\
\quad (s, edfrs.I \cup edfrs.O, edfrs.T, sdfrs.F) \in is\_stable \land \\
\quad (s, sG, edfrs.I \cup edfrs.O, edfrs.T) \in static\_guards\_true \land \\
\quad (s, tG, edfrs.T) \notin timed\_guards\_true \bullet \exists d : DELAY \bullet \\
\qquad d \in enablingDelays(s, edfrs.I \cup edfrs.O, edfrs.T, (sG, tG, asgmts))
\end{array}
$$

Other restriction (Definition 4.2.3) is related to the dependence of inputs with respect to satisfiable timed guards. Let s be a stable state, $(sG, tG, asgmts)$ an entry of a function f, such that sG is true, tG is false, and there is a maximum delay (upper bound) that enables tG from s, there must not exist a set of input assignments ($asgmt2$) that disables sG. If this property does not hold, our CSP representation would not be appropriate as it would allow the following situation: the system is in a state such that sG is true, the CSP specification behaves as *INPUTS*, sG is disabled, when performing *DELAY*, since sG is false now, the corresponding *eta* is not set to true and, thus, the CSP specification has modelled a situation where a set of inputs might happen unconstrained with respect to time, which is not the case (remember the upper bound).

Definition 4.2.3. *No dependence of inputs for satisfiable timed guards: let sdfrs be an s-DFRS, and edfrs the e-DFRS obtained from sdfrs by the application of expandedDFRS, the following property shall hold when using CSP to represent DFRS models:*

$$
\begin{array}{l}
\forall s : edfrs.S ; f : sdfrs.F ; sG, tG : EXP ; asgmts : ASGMTS \mid (sG, tG, asgmts) \in f \land \\
\quad (s, edfrs.I \cup edfrs.O, edfrs.T, sdfrs.F) \in is\_stable \land \\
\quad (s, sG, edfrs.I \cup edfrs.O, edfrs.T) \in static\_guards\_true \land \\
\quad (s, tG, edfrs.T) \notin timed\_guards\_true \land \\
\quad maxDelays(s, edfrs.I \cup edfrs.O, edfrs.T, (sG, tG, asgmts)) \neq \emptyset \land \\
\quad (\exists d : DELAY \bullet d \in enablingDelays(s, edfrs.I \cup edfrs.O, edfrs.T, (sG, tG, asgmts))) \bullet \\
\qquad \neg\, (\exists asgmts2 : ASGMTS ; s2 : STATE \bullet \mathrm{dom}\, asgmts2 = \mathrm{dom}\, edfrs.I \land \\
\qquad\quad (\forall asgmt : asgmts2 \bullet asgmt.2 \in values(edfrs.I(asgmt.1))) \land \\
\qquad\quad s2 = nextState(s, edfrs.T, asgmts2) \land \\
\qquad\quad (s2, sG, edfrs.I \cup edfrs.O, edfrs.T) \notin static\_guards\_true)
\end{array}
$$

Finally, another restriction is the one presented in Definition 4.2.4. Briefly, from a non-stable state (s), it is not possible to reach other non-stable state ($s2$) performing only function transitions such that a static guard that was false in s becomes true in $s2$, considering that in s the timed guard associated to this static guard was already true.

Definition 4.2.4. *No interdependence of function transitions and timed guards: let sdfrs be an s-DFRS, and edfrs the e-DFRS obtained from sdfrs by the application of expandedDFRS, the following property shall hold when using CSP to represent DFRS models:*

$$
\begin{array}{l}
\forall s : edfrs.S ; f : edfrs.F \mid (s, edfrs.I \cup edfrs.O, edfrs.T, sdfrs.F) \notin is\_stable \bullet \\
\quad \neg\, (\exists s2 : edfrs.S ; sG, tG : EXP ; asgmts : ASGMTS \bullet (sG, tG, asgmts) \in f \land \\
\qquad (s, sG, edfrs.I \cup edfrs.O, edfrs.T) \notin static\_guards\_true \land \\
\qquad (s, tG, edfrs.T) \in timed\_guards\_true \land \\
\qquad (s2, sG, edfrs.I \cup edfrs.O, edfrs.T) \in static\_guards\_true \land \\
\qquad s2 \in funReachable(s) \land s2 \neq s)
\end{array}
$$

Our CSP representation is also not capable of modelling such a behaviour, since in s, as sG is false, the *eta* variable associated with tG would not be set to true, and when sG becomes true after a sequence of function transitions, the corresponding *asgmts* would not be performed since the *eta* variable was not set to true previously.

These properties, similarly to the ones presented in Section 3.3.3, can be dynamically verified during the creation of an e-DFRS via bounded model checking: while the model is created, we check whether the desired properties are met. Eligible criteria for this bound are the number of delay (function) transitions performed, and an upper bound for the system global clock. As discussed in Section 4.4.2, bounds such as these ones are commonly used when analysing properties of timed reactive systems.

4.3 CSP-TIO processes

Hereafter we develop out testing theory based on CSP. This theory is an extension of the one presented in (NOGUEIRA; SAMPAIO; MOTA, 2014) considering particularities of our models (data-flow reactive systems), and particularly handling time. The work in (NOGUEIRA; SAMPAIO; MOTA, 2014) defines a CSP I/O process as a triple (P, α_I, α_O) composed by a CSP process (P), and its input (α_I) and its output (α_O) events, such that $\alpha_I \cap \alpha_O = \emptyset$. Here, we extend this notion to define CSP timed input-output processes as shown below.

Definition 4.3.1. *CSP-TIO processes: a CSP-TIO process is a tuple $(P, \alpha_I, \alpha_O, \alpha_T, \alpha_R, \alpha_A)$, where P is a CSP process, α_I and α_O represent input and output events, respectively, α_T and α_R represent time evolving and timer reset events, and α_A comprises auxiliary events used during test-scenario generation and test selection. These sets are disjoint ($disjoint\langle \alpha_I, \alpha_O, \alpha_T, \alpha_R, \alpha_A \rangle$) and, together, they define the alphabet of the CSP-TIO process ($\alpha = \bigcup\{\alpha_I, \alpha_O, \alpha_T, \alpha_R, \alpha_A\}$).*

A CSP-TIO process also has a particular property: its traces present an alternation of input, time and output events. To define this property, consider the function *iotAlt* (Definition 4.3.2)

Definition 4.3.2. *Function iotAlt: let TIO be a CSP-TIO process, t a trace, pos $\in \{0, 1, 2\}$, head and tail functions that yield the head and tail of non-empty sequences. The function iotAlt is defined as follows.*

```
iotAlt(TIO, t, pos) =
    if t = $\langle\rangle$ then true
    else if (pos = 0 $\wedge$ head(t) $\in \alpha_{TIO_O}$) $\vee$
            (pos = 1 $\wedge$ head(t) $\in \alpha_{TIO_I}$) $\vee$
            (pos = 2 $\wedge$ head(t) $\in \alpha_{TIO_T}$) then
              iotAlt(TIO, tail(t), (pos+1)%3)
    else false
```

The rationale for requiring this property (formally defined in Definition 4.3.3) is the reasonable expectation that a data-flow reactive system, as a class of embedded systems whose inputs and outputs are always available as signals, exhibit an alternating sequence of: receiving new inputs (input events), after some delay (time events), and then producing new values for its output signals (output events). Initially, we have the initial values communicated by its output signals considering the initial values of the input signals. We note that in Definition 4.3.3, the property holds if the trace t belongs to $\mathscr{T}(TIO_P \setminus (\alpha_{TIO_R} \cup \alpha_{TIO_A}))$, that is, if we ignore (hide) the reset and auxiliary events of the CSP-TIO process.

Definition 4.3.3. *Alternation of input, time and output events: let TIO be a CSP-TIO process, and iotAlt as presented in Definition 4.3.2; then* $\forall t : \mathscr{T}(TIO_P \setminus (\alpha_{TIO_R} \cup \alpha_{TIO_A})) \bullet iotAlt(TIO, t, 0)$.

Furthermore, we also assume that a CSP-TIO process is a non-terminating process, and it is not used to represent systems with time lock (a state from which a time event can no longer happen). These assumptions are reasonable, since we consider in this work embedded systems whose inputs and outputs are always available, as signals. These assumptions in conjunction with Definition 4.3.3 implies that a CSP-TIO process is always able to produce some output after each input.

To give a concrete example, consider the CSP-TIO process for the VM (*VM_CSP-TIO*) shown in Code 4.8. First, we hide from *SYSTEM* its internal events (lines 1–2). As internal events we consider the communications over the channels get and set (used to synchronise with the *MEMORY* process and, thus, simulate communication via shared memory); and channels that comprise the set of possible values of input variables.

Code 4.8: CSP – CSP-TIO process (vending machine)

```
 1 S_internal = {|get, set,
 2   c_the_coffee_request_button, c_the_coin_sensor|}
 3
 4 S = SYSTEM \ S_internal
 5
 6 S_inputs = {|input|}
 7 S_outputs = {|output|}
 8 S_time = {|delay|}
 9 S_reset = {reset.the_request_timer}
10 S_auxiliary = {REQ003, REQ005, REQ001, REQ002, REQ004, stableState}
11
12 S_alphabet = union(S_inputs, union(S_outputs,
13   union(S_time, union(S_reset, S_auxiliary))))
14
15 VM_CSP–TIO = (S, S_inputs, S_outputs, S_time, S_reset, S_auxiliary)
```

[Source: author]

The inputs, outputs and time events are defined as the events communicated over the channels *input*, *output*, and *delay*, respectively (lines 6–8). In the VM, the event used to denote reset of timers is *reset.the_request_timer* and, thus, *S_reset* is the singleton set composed by this event (line 9). The first five events of *S_auxiliary* are used to select test scenarios guided by a requirement-coverage criteria (see Section 4.5). The last event (*stableState*) is used during generation of test scenarios, as explained later.

The property defined in Definition 4.3.3 holds for *VM_CSP-TIO* as a consequence of the definition of the process *SPECIFICATION*. First, it behaves as *FUN*, which eventually behaves as *OUTPUTS* and, thus, produces an output event, then behaves as *INPUTS* (generating an input event), and finally as *DELAY* (generating a time event). Afterwards, *SPECIFICATION* behaves recursively and, thus, alternating again within this sequence of output, input and time events.

4.4 A CSP timed input-output conformance relation

As we use the process algebra CSP, a natural choice for a conformance relation would be one of the traditional CSP refinement notions (traces, failures or failures-divergences). However, we consider a different approach influenced by particular characteristics of the kind of systems we are dealing with.

First, the inputs and outputs of a data-flow reactive system are always available, as signals. Second, the systems considered have a clear separation between inputs and outputs. Finally, as defined by Definition 4.3.3, there is a particular alternation of input, time and output events. As the relation *ioco* (TRETMANS, 1999) is a traditional relation that also segregates input and output events, besides having also influenced the definition of timed relations, we consider it as a basis for our work. In Section 4.4.1, we present and discuss our CSP timed input-output conformance relation: csptio. Then, in Section 4.4.2, we show how conformance verification with respect to csptio is mechanised.

4.4.1 Definition of csptio conformance

The relation csptio can be seen as a timed extension of *cspio* (NOGUEIRA; SAMPAIO; MOTA, 2014), which is inspired by *ioco* itself and is formalised in terms of CSP constructs and traces refinement. In contrast to *cspio*, however, we can reason about *time* (discrete and continuous). Our conformance relation assumes that the specification $S = (S_P, \alpha_{S_I}, \alpha_{S_O}, \alpha_{S_T}, \alpha_{S_R}, \alpha_{S_A})$ is a CSP-TIO process (see Definition 4.3.1), and that the SUT $I = (I_P, \alpha_{I_I}, \alpha_{I_O}, \alpha_{I_T}, \alpha_{I_R}, \alpha_{I_A})$ can also be modelled in such a formalism (*testability hypothesis*).

Differently from S, implementations (I) do not have reset and auxiliary events: $\alpha_{I_R} = \emptyset$ and $\alpha_{I_A} = \emptyset$. These are reasonable assumptions, since an implementation, as a black-box, only exposes the values communicated over its output signals. Therefore, the alphabet of an implementation comprises only its inputs, outputs and time elapsing events ($\alpha_I = \bigcup\{\alpha_{I_I}, \alpha_{I_O}, \alpha_{I_T}\}$), which is a direct consequence of being a CSP-TIO process and $\alpha_{I_R} = \emptyset$, $\alpha_{I_A} = \emptyset$.

The alphabet of I is assumed to be compatible with that of S (Definition 4.4.1). Although our compatibility definition allows reasoning about partial specifications (implementations might consider more input and output events than specifications), we assume that both consider the same input and output signals. What is allowed is considering new values for these signals.

Definition 4.4.1. *Compatibility of alphabets: let S and I be CSP-TIO processes, their alphabets are compatible if, and only if, $\alpha_{S_I} \subseteq \alpha_{I_I} \wedge \alpha_{S_O} \subseteq \alpha_{I_O}$.*

To give a concrete example, we consider a specification with one input signal ($in1$), whose values range over the set $\{0,1\}$. The set α_{S_I} is $\{input.in1.0, input.in1.1\}$. Suppose one implementation ($imp1$) that has two input signals ($in1, in2$), both ranging over the set $\{0,1\}$. In such a situation, α_{I_I} is:

$$\{input.in1.0.in2.0, input.in1.1.in2.0, input.in1.0.in2.1, input.in1.1.in2.1\}$$

and, thus, its alphabet is not compatible with the one of the considered specification. However, suppose another implementation (*imp*2) with the same input signal (*in*1), but more possible values ($\{0,1,2\}$). In such a case, its alphabet is compatible with that of the specification. Moreover, the specification can be seen as a partial one, since it does not state how the system should react over the input *input.in*1.2.

The time events of specifications and implementations are not related, since we assume that time is symbolically modelled in specifications (as shown in Section 4.2), whereas it is a concrete concept on implementations. In other words, we assume that, with the aid of a projection function $proj : \alpha_{I_T} \rightarrow \mathbb{N}$, we can extract from the time events of implementations (α_{I_T}) the concrete amount of time elapsed. For instance, let *elapse*.10 be an event in α_{I_T}, $proj(elapsed.10) = 10$. Here, we consider for simplicity $\mathbb{N}$, since our CSP representation does not consider floating-point numbers. Therefore, $\{ev : \alpha_{I_T} \bullet proj(ev)\} \subseteq \mathbb{N}$. Finally, the reset and auxiliary event of specifications do not have any relation with implementation events, since $\alpha_{I_R} = \emptyset \wedge \alpha_{I_A} = \emptyset$.

Besides the relation between alphabets, another important aspect when reasoning about conformance is quiescence. This is defined as the situation when the system is unable to perform an output event now or in the future, without performing first an input event. If we recall the property of alternating input, time and output events that holds for CSP-TIO processes (Definition 4.3.3), we easily note that after performing an output event, a CSP-TIO process is always quiescent, since it is not possible for it to perform an output without performing first an input event.

In some conformance relations (e.g., ioco), it is necessary to annotate the model with information regarding quiescence. It is important to reason whether the implementation is less quiescent (an expected property), and not more quiescent (an undesired property). In csptio, despite having quiescence, it is not necessary to annotate this information, since both specifications and implementations are always quiescent at the same states (after performing output events) due to Definition 4.3.3. If this is not true, then the property presented in Definition 4.3.3 does not hold, and we would not be considering CSP-TIO processes – a fundamental expectation, as explained in the beginning of this section.

Both specifications and implementations are also expected to be input-complete (also known as input-enabled) with respect to their input alphabets (α_{S_I} and α_{I_I}, respectively) at certain states. Regarding specifications, this is ensured by the way we generate CSP specifications from s-DFRSs. In Section 4.2, when explaining the process *INPUTS*, we have discussed that the new value for the input signals can be any value of its types (c_input_i ranges over the possible values of each system input). When creating CSP-TIO models for implementations, we also expect that the input-enabled property holds. However, it is worth noting that as a consequence of Definition 4.3.3 again, a CSP-TIO process only expects input events at certain moments (just after performing output events). Therefore, input-completeness only applies to these states.

This conformance relation is defined in terms of the traces semantics. In order to define it, we need two auxiliary functions: *out* and *elapse*. Intuitively, the function *out* receives a CSP-TIO process, besides a trace, and yields the output events that can be performed after this trace (see Definition 4.4.2). It is important to note that, since *out* is not concerned with time-related information, events that do not belong to $\alpha_{TIO_I} \cup \alpha_{TIO_O}$ are not considered. Therefore, the trace t is restricted ($\upharpoonright$) to the events of $\alpha_{TIO_I} \cup \alpha_{TIO_O}$, and all other events ($\alpha_{TIO} \setminus (\alpha_{TIO_I} \cup \alpha_{TIO_O})$) are hidden ($\setminus$) in TIO_P.

Definition 4.4.2. *Function out: let TIO be a CSP-TIO process, and t a trace, the function out is defined as follows:*

$$\begin{aligned} out(TIO,t) = \{ev : \alpha_{TIO_O} \mid \\ (t \upharpoonright (\alpha_{TIO_I} \cup \alpha_{TIO_O})) \frown \langle ev \rangle \in \mathscr{T}(TIO_P \setminus (\alpha_{TIO} \setminus (\alpha_{TIO_I} \cup \alpha_{TIO_O})))\}\end{aligned}$$

The function *elapse* yields the amount of time that might elapse after performing a particular trace. However, since specifications and implementations deal with time differently, we have two versions of this function: $elapse_{Con}$, used when considering concrete time representation (implementations), and $elapse_{Sym}$, used when considering symbolic time representation (specifications).

The function $elapse_{Sym}$ yields the amount of time that might elapse from a symbolic time event. As explained in Section 4.3, the time events of a specification are the communications over the channel *delay*. In Section 4.2, we have shown that this channel is defined considering *eta* variables: an eta_i variable is a (boolean) flag created after one timed guard. We recall that we keep traceability between the *eta* variables and their corresponding timed guards (see Code 4.6). When a true value is associated with one of these *eta* variables, we know that at this moment the time elapsed respects the timed guard associated with this variable. Note that no more than one true value will be communicated over the channel *delay*, since the *eta* variables are reset in the beginning of *DELAYS*, and each choice sets only one of these variables to true.

Consider that $map : \alpha_{S_T} \rightarrow String$ is a mapping function that yields the timed guard associated to the given time event via the inspection of which *eta* variable is set to true. If no variable is set to true, it means that the amount of time that might elapse is unconstrained.

To find out the concrete values that satisfy time constraints, we rely on a constraint solver. In this work, we use the SMT solver Z3. In summary, the function $elapse_{Sym}$ yields a set of concrete values that represent the time that might elapse after a certain trace t (see Definition 4.4.3). In this definition, *isSAT* denotes the algorithm that uses the solver to find the concrete values for time elapsing (see Algorithm 16).

Definition 4.4.3. *Function $elapse_{Sym}$: let TIO be a CSP-TIO process of a specification, t a* *trace, and map a function that yields from symbolic time events strings that represent the time constraints associated with the event (map* : $\alpha_{S_T} \rightarrow$ *String), then function* $elapse_{Sym}$ *is defined as follows:*

$$\begin{aligned}&elapse_{Sym}(TIO, t, map) = \{ev : \alpha_{TIO_T}; delay : \mathbb{N} \mid \\ &\quad (t \upharpoonright \alpha_{TIO_T}) \frown \langle ev \rangle \in \mathscr{T}(TIO_P \setminus (\alpha_{TIO} \setminus \alpha_{TIO_T})) \wedge \\ &\quad isSAT(\alpha_{TIO_T}, \alpha_{TIO_R}, (t \upharpoonright \alpha_{TIO_T} \cup \alpha_{TIO_R}) \frown \langle ev \rangle, map, delay) \bullet delay\}\end{aligned}$$

Analogously to *out* (Definition 4.4.2), the function $elapse_{Sym}$ restricts the trace to symbolic time events (α_{TIO_T}), and hides from TIO_P all other events, since it is only concerned with time events. It is important to note that, with respect to *isSAT*, we consider the traces restricted to time (α_{TIO_T}), but also to reset (α_{TIO_R}) events. This is necessary, since, as explained later, the reset events are relevant to assessing the satisfiability of constraints.

The definition of *isSAT* is given in Algorithm 16. Given the time (α_{TIO_T}) and reset (α_{TIOT_R}) events, the trace performed, along with the time event $((t \upharpoonright \alpha_{TIO_T} \cup \alpha_{TIO_R}) \frown \langle ev \rangle)$, the *map* function, and a possible delay (*delay*), the function *isSAT* verifies whether this delay satisfies the time constraint associated with *ev*. If so, *delay* belongs to the output of $elapse_{Sym}$. To perform this analysis, in line 1, *isSAT* calls *buildSMTProblem* to derive an SMT problem, then, it adds a new constraint to this problem (line 2) before checking its satisfiability (line 3 – *checkSAT* is the function provided by the SMT solver to check whether there is a solution for the given SMT problem). Before explaining the goal of this new constraint, we explain the function *buildSMTProblem*.

Algorithm 16: Definition of isSAT

```
input  : $\alpha_T, \alpha_R, t, map, delay$
output : $satisfiable$
1 $constraints, indexMap = buildSMTProblem(\alpha_T, \alpha_R, t, map)$;
2 $constraints.push($“(assert (= _d”$+indexMap.get($“_d”$)$ +“ ”$+delay+$“))”$)$;
3 $satisfiable = checkSAT(constraints)$;
```

[Source: author]

Algorithm 17 describes how the function *buildSMTProblem* derives an SMT problem from a given trace. Intuitively, we create variables to represent the global clock and timers at different instants (gc_i and $timer_i$, respectively), and, for each time and reset events, constraints are added to the SMT problem to constrain how time evolves (these variables are updated). For instance, when we find a (symbolic) time event that does not restrict how the time evolves, we create the constraints below (note that they are in prefix notation):

(declare-fun gc_i *() Int)*
(declare-fun $\_d_j$ *() Int)*
(assert (> $\_d_j$ *0))*
(assert (= gc_i *(+* gc_{i-1} $\_d_j$*)))*

If there is no constraint on how the time evolves, the global clock (gc_i) is constrained to be equal to its previous value (gc_{i-1}) added by a delay ($\_d_j$) that needs to be greater than 0. However, when the amount of time elapsed needs to satisfy some time constraint (the one mapped to the *eta* variable whose value is true), we create an additional constraint. For instance, suppose that the following constraint needs to be satisfied *(> (- gc t) 30)*, the additional constraint is:

> *(assert (> (- gc_i t_k) 30))*

As one can notice, we only need to append to the variables concerned (*gc* and *t*) their current indexes. When a reset event is found with respect to a timer *t*, we create the constraints:

> *(declare-fun t_k () Int)*
> *(assert (= t_k gc_i))*

That is, at this moment, the value of the timer (t_k) becomes equal to the current value of the global clock (gc_i). Time (α_{TIO_T}) and reset (α_{TIO_R}) events are the only ones in the given trace as we restrict traces to elements of these two sets, since they are the only events with relevant information about time evolving.

Algorithm 17 systematises the generation of constraints from traces. First, it initialises auxiliary variables (lines 1–2) to store the constraints (*constraints*) and the current index of the global clock, timers, and delays (*indexMap*). These indexes are initially 1 (lines 3–4, 9).

Afterwards, constraints are generated to declare variables for the SMT problem at their initial stage: the global clock (gc_1), lines 5–7; and timers (t_1), lines 8–12. Initially, the value of these variables is 0 (lines 6 and 11). After declaring these variables, their indexes are incremented (lines 7 and 12).

Then, the algorithm loops over the events of the trace *t* (lines 13–14). If the event is a time one ($h \in \alpha_T$ – line 15), we generate constraints as previously explained (lines 16–21). In what follows, after updating the indexes of the SMT variables (lines 22–23), we check whether the amount of time elapsed needs to be constrained (line 24). As explained earlier, it happens when the time event has a value *true* associated with some *eta* variable. If it is the case, we retrieve the constraint associated with this time event with the aid of the function *map* (line 25). Afterwards (lines 26–27), this constraint is modified to append the current indexes of the SMT variables to the name of these variables. Finally, an additional constraint is created and pushed into the stack of constraints (lines 28–29).

For reset events ($h \in \alpha_R$ – line 30), we generate the constraints that ensure that the current value of the timer mentioned by this event (the function *getTimerName* yields the name of the timer being mentioned – line 31) is equal to the current value of the global clock (lines 32–34). To exemplify the usage of *getTimerName* within the VM example, when it is applied to the event *reset.the_request_timer*, it yields *the_request_timer*.

To illustrate Algorithm 17, we consider part of the trace (up to the last delay event) shown in Section 4.5 with respect to the VM example. We reproduce below this trace, restricted

Algorithm 17: Definition of buildSMTProblem

```
input  : $\alpha_T, \alpha_R, t, map$
output : $constraints, indexMap$
$constraints = new\ ConstraintsStack < String > ()$;
$indexMap = new\ HashMap < String, Integer > ()$;
$indexMap.put($“_d”$), 1)$;
$indexMap.put($“gc”$, 1)$;
$constraints.push($“(declare-fun gc”$+indexMap.get($“gc”$)+$“ () Int)”$)$;
$constraints.push($“(assert (= gc”$+indexMap.get($“gc”$)+$“ 0))”$)$;
$indexMap.put($“gc”$, indexMap.get($“gc”$) + 1)$;
for $timer \in dfrs.T$ do
    $indexMap.put(timer, 1)$;
    $constraints.push($“(declare-fun ”$+timer+$“”$+indexMap.get(timer)+$“ () Int)”$)$;
    $constraints.push($“(assert (= ”$+timer+$“”$+indexMap.get(timer)+$“ 0))”$)$;
    $indexMap.put(timer, indexMap.get(timer) + 1)$;
while $!empty(t)$ do
    $h = head(t)$;
    if $h \in \alpha_T$ then
        $constraints.push($“(declare-fun _d”$+indexMap.get($“_d”$)+$“ () Int)”$)$;
        $constraints.push($“(assert (> _d”$+indexMap.get($“_d”$)+$“ 0))”$)$;
        $constraints.push($“(declare-fun gc”$+indexMap.get($“gc”$)+$“ () Int)”$)$;
        $constraints.push($“(assert (= gc”$+indexMap.get($“gc”$) + 1+$
                    “ (+ gc”$+indexMap.get($“gc”$)+$
                    “ _d”$+indexMap.get($“_d”$))+$“)))”$)$;
        $indexMap.put($“gc”$, indexMap.get($“gc”$) + 1)$;
        $indexMap.put($“_d”$, indexMap.get($“_d”$) + 1)$;
        if $h.indexOf($“true”$)! = -1$ then
            $constraint = map(h)$;
            for $v \in indexMap \wedge constraint.indexOf(v)! = -1$ do
                $constraint = constraint.replace(v, v + indexMap.get(v))$;
            $constraint =$“(assert ”$+constraint+$ “)”;
            $constraints.push(constraint)$;
    else if $h \in \alpha_R$ then
        $var = getTimerName(h)$;
        $constraints.push($“(declare-fun ”$+var+$“”$+indexMap.get(var)+$“ () Int)”$)$;
        $constraints.push($“(assert (= ”$+var+$“”$+indexMap.get(var)+$
                    “ gc”$+indexMap.get($“gc”$)+$“))”$)$;
        $indexMap.put(var, indexMap.get(var) + 1)$;
    $t = tail(t)$;
```

[Source: author]

to time and reset events, since, as explained, these are the events relevant to the time-related analysis.

This trace depicts the situation when the coin sensor becomes true after some delay (first event – unconstrained), the system resets the request timer (second event), this sensor becomes false again after some other delay (third event – unconstrained), and then the coffee request button is pressed after a delay that satisfies the constraint mapped to *eta1* (fourth event – constrained by *eta1*).

```
<
    delay.eta1.B.false.eta2.B.false.eta3.B.false.eta4.B.false,
    reset.the_request_timer,
    delay.eta1.B.false.eta2.B.false.eta3.B.false.eta4.B.false,
    delay.eta1.B.true.eta2.B.false.eta3.B.false.eta4.B.false
>
```

Given this trace, the function *buildSMTProblem* yields the following SMT problem. Note that the constraints are grouped for legibility purposes into five blocks (separated by blank lines).

```
(declare-fun gc1 () Int)
(declare-fun the_request_timer1 () Int)
(assert (= gc1 0))
(assert (= the_request_timer1 0))

(declare-fun gc2 () Int)
(declare-fun _d1 () Int)
(assert (> _d1 0))
(assert (= gc2 (+ gc1 _d1)))

(declare-fun the_request_timer2 () Int)
(assert (= the_request_timer2 gc2))

(declare-fun gc3 () Int)
(declare-fun _d2 () Int)
(assert (> _d2 0))
(assert (= gc3 (+ gc2 _d2)))

(declare-fun gc4 () Int)
(declare-fun _d3 () Int)
(assert (> _d3 0))
(assert (= gc4 (+ gc3 _d3)))
(assert (> (- gc4 the_request_timer3) 30))
```

The first block is concerned with the declaration of the variables of the SMT problem in their initial stage. Then, we have one block for each event of the trace. The second block declares the constraint to advance time with no constraints (first event of the trace). The third block represents the reset of the request timer (second event of the trace). The fourth block is another time elapse with no constraint (third event of the trace). The fifth block advances the time such that the system goes to the producing-strong-coffee mode (fourth event of the trace). To reach this state, the coffee request needs to be performed 30 time units after the coin has been inserted. Provided with this satisfiability problem, an SMT solver such as Z3 searches for solutions (also known as models): values for all variables declared such that all constraints are satisfied.

Now, we return to the explanation of the function *isSAT*. The goal of this function is to check whether a given delay (*delay*) is possible with respect to the last event of the given trace (t). To make this analysis, Algorithm 16 adds an extra constraint to the SMT problem (line 2):

(assert (= _d_k delay))

This constraint requires the last performed delay (_d_k) to be equal to the given *delay*. If the solver finds a solution for this modified SMT problem, then *isSAT* yields satisfiable. Regarding the example shown above, this additional constraint is *(assert (= _d3 delay))*. One final important remark about our explanation of building SMT problems: we consider discrete time (the SMT variables are declared as integers), but to reason about continuous time it would suffice to declare these variables as float.

Now, we proceed our explanation of the calculation of the time that can elapse, but with respect to implementations, which deal with concrete time delays. Before presenting $elapse_{Con}$, we define two auxiliary functions: *subTimedTrace* and *elementAt*.

The function *subTimedTrace* (Definition 4.4.4) yields a subtrace from a given one (t), starting from its first element up to the *i-th* element that belongs to α_T. Intuitively, it yields a subtrace that contains $i+1$ elements from α_T. Considering this, the function is defined recursively over t: if the head is not an element of α_T, it does not decrement i, since the subtrace must have $i+1$ elements of α_T. If the head is an element of α_T, and if it is the *i-th* element, the function stops considering this last element, otherwise, it continues until this situation is reached.

Definition 4.4.4. *Function subTimedTrace: let t be a trace, α_T a set of events, i a natural number ($i \in \mathbb{N}$), and assume that $\#(t \upharpoonright \alpha_T) > i$, the function subTimedTrace is defined as follows:*

```
subTimedTrace(t, $\alpha_T$, i) =
    if head(t) $\notin \alpha_T$ then head(t) $\frown$ subTimedTrace(tail(t), $\alpha_T$, i)
    else
        if i = 0 then head(t) else head(t) $\frown$ subTimedTrace(tail(t), $\alpha_T$, i-1)
```

The function elementAt (Definition 4.4.5) yields the *i-th* element of a trace (t). Therefore, it recursively iterates over the given trace until the *i-th* element is reached.

Definition 4.4.5. *Function elementAt: let t be a trace, i a natural number ($i \in \mathbb{N}$), and assume that $\#t > i$, the function $subTimedTrace$ is defined as follows:*

elementAt(t, i) = if i = 0 then head(t) else elementAt(tail(t), i-1)

Considering these two functions, given a trace t of the specification, $elapse_{Con}$ (see Definition 4.4.6) yields the amount of time that might elapse after performing this trace. Since $elapse_{Con}$ deals with concrete time elapses, whereas $elapse_{Sym}$ considers symbolic representations, we need to find the traces of the implementation ($t1$) that correspond to the given trace t of the specification. These traces perform the same sequence of input and output events of t ($t1 \upharpoonright (\alpha_{I_I} \cup \alpha_{I_O}) = t \upharpoonright (\alpha_{S_I} \cup \alpha_{S_O})$), and the same number of time events ($\#(t1 \upharpoonright \alpha_{I_T}) = \#(t \upharpoonright \alpha_{S_T})$). Moreover, each *i-th* concrete time delay of $t1$ ($proj(elementAt(t1 \upharpoonright \alpha_{I_T}))$ is a valid delay considering the corresponding symbolic time event of the specification.

To verify this property we rely on $isSAT$ (see Definition 16), and the previously defined functions $subTimedTrace$ and $elementAt$. Then, for all time events of the implementation ($ev : \alpha_{I_T}$) that can happen after the trace $t1$ (restricted to time events – α_{I_T}), the function $elapse_{Con}$ yields the concrete time delay associated with these time events ($proj(ev)$).

Definition 4.4.6. *Function $elapse_{Con}$: let I and S be CSP-TIO processes of an implementation and a specification, respectively, t a trace from S, $proj : \alpha_{I_T} \rightarrow \mathbb{N}$ a function that projects from time events the concrete amount of time elapsed, and $isSAT$, $subTimedTrace$, map and $elementAt$ the functions defined previously, then function $elapse_{Con}$ is defined as follows:*

$$\begin{aligned}
&elapse_{Con}(I,S,t,proj,map) = \{t1 : \mathcal{T}(I_P)\,; ev : \alpha_{I_T} \mid \\
&\quad t1 \upharpoonright (\alpha_{I_I} \cup \alpha_{I_O}) = t \upharpoonright (\alpha_{S_I} \cup \alpha_{S_O}) \land \#(t1 \upharpoonright \alpha_{I_T}) = \#(t \upharpoonright \alpha_{S_T}) \land \\
&\quad (\forall i : \mathbb{N} \mid i \geq 0 \land i < \#(t1 \upharpoonright \alpha_{I_T}) \bullet isSAT(\alpha_{S_T}, \alpha_{S_R}, \\
&\qquad subTimedTrace(t \upharpoonright (\alpha_{S_T} \cup \alpha_{S_R}), \alpha_{S_T}, i), map, proj(elementAt(t1 \upharpoonright \alpha_{I_T}), i))) \land \\
&\quad (t1 \upharpoonright \alpha_{I_T}) \frown \langle ev \rangle \in \mathcal{T}(I_P \setminus (\alpha_I \setminus \alpha_{I_T})) \bullet proj(ev)\}
\end{aligned}$$

Now, based on the previously defined functions (out, $elapse_{Sym}$, and $elapse_{Con}$) we define csptio: a CSP timed input-output conformance relation (see Definition 4.4.7).

Definition 4.4.7. *CSP timed input-output conformance relation (csptio): let S and I be CSP-TIO processes, $proj : \alpha_{I_T} \rightarrow \mathbb{N}$ a mapping function from time events of I to natural numbers, and $map : \alpha_{S_T} \rightarrow String$ a mapping function from time events of S to strings (constraints), I* ***csptio*** *S if, and only if, the following property holds:*

$$\forall t : \mathcal{T}(S_P) \bullet out(I,t) \subseteq out(S,t) \land elapse_{Con}(I,S,t,proj,map) \subseteq elapse_{Sym}(S,t,map)$$

The intuition behind this definition is that, after a trace t of the specification ($t \in \mathscr{T}(S_P)$), the implementation shall output a subset of the expected outputs allowed by the specification ($out(I,t) \subseteq out(S,t)$). Furthermore, the implementation cannot have an amount of elapsed time not allowed in the specification ($elapse_{Con}(I,S,t,proj,map) \subseteq elapse_{Sym}(S,t,map)$).

To illustrate csptio, suppose an implementation ($I1$) that always produces weak coffee as output, independently of the time elapsed between inserting the coin and the coffee request. The relation *I1 csptio VM_CSP-TIO* does not hold since there is a trace from the specification where producing strong coffee is the only expected option and, thus, it is the single element in *out* of *VM_CSP-TIO*, but, for the same trace, *out* of $I1$ is equal to producing weak coffee. Therefore, as *out* of I is not a subset of *out* of *VM_CSP-TIO* for such a trace, $\neg$ *(I cpstio VM_CSP-TIO).*

Concerning time-based behaviour, another implementation ($I2$) that dispenses weak coffee within $[9,12]$ time units after the coffee request is also not in conformance with *VM_CSP-TIO*, since 9 belongs to $elapse_{Con}$ of $I2$, but not to $elapse_{Sym}$ of *VM_CSP-TIO*, for a trace representing the production of weak coffee.

Differently, a third implementation ($I3$) that always dispenses strong coffee within 33 time units after the request is in conformance with *VM_CSP-TIO*, since $elapse_{Con}(I3,...) = \{33\}$ for the traces that dispense strong coffee, and it is true that $\{33\}$ belongs to $elapse_{Sym}$ of *VM_CSP-TIO*, since, according to the specification, a strong coffee is dispensed within $[30,50]$ time units after the request.

4.4.2 Verifying csptio conformance

We have mechanised the verification that an implementation conforms to its specification, based on csptio, in terms of a high-level strategy by reusing successful techniques and tools: refinement checking (FDR) and SMT solving (Z3). Furthermore, our mechanisation is sound with respect to the csptio definition (Theorem 4.4.1). The theorem is itself an automated means for checking conformance with respect to csptio.

Here, we provide a proof sketch. Let us analyse separately the conjunction terms. The proof of the first term is similar to the proof of Theorem 3 presented in (NOGUEIRA; SAMPAIO; MOTA, 2014), but lifted to CSP-TIO processes. Here, as the CSP processes have more information than input and output actions, we hide all events of the alphabets of the specification (α_S) and of the implementation (α_I) that are not inputs or outputs ($S_P \setminus (\alpha_S \setminus (\alpha_{S_I} \cup \alpha_{S_O}))$ and $I_P \setminus (\alpha_I \setminus (\alpha_{I_I} \cup \alpha_{I_O}))$, respectively).

The first term is a refinement expression. If an input event occurs in the implementation, but not in the specification, on the right-hand side of the refinement, the parallel composition does not progress through this event (this event is refused). As the refinement considers the traces model, refused events are not taken into account. Therefore, new input events in the implementation are allowed. The goal of the process $ANY(\alpha_{I_O}, STOP)$ is to avoid that the right-hand side refuses output events that the implementation can perform, but the specification cannot. Therefore, if after a common trace the implementation performs output events not expected in the specification, these events appear in the traces of the implementation and the refinement expression is false.

Theorem 4.4.1. *Mechanisation of csptio: let S and I be CSP-TIO processes; I **csptio** S holds if, and only if, the following predicate holds:*

$$\begin{array}{l}
(S_P \setminus (\alpha_S \setminus (\alpha_{S_I} \cup \alpha_{S_O})) \sqsubseteq_{T} \\
\quad (S_P \setminus (\alpha_S \setminus (\alpha_{S_I} \cup \alpha_{S_O})) \triangle ANY(\alpha_{I_O}, STOP)) \underset{\alpha_{I_I} \cup \alpha_{I_O}}{\parallel} I_P \setminus (\alpha_I \setminus (\alpha_{I_I} \cup \alpha_{I_O}))) \wedge \\
(\forall t : \mathscr{T}(S) ; d : \mathbb{N} \bullet d \in elapse_{Con}(I, S, t, proj, map) \Rightarrow \\
\quad \exists ev : \alpha_{S_T} \bullet (t \upharpoonright \alpha_{S_T}) \frown \langle ev \rangle \in \mathscr{T}(S_P \setminus (\alpha_S \setminus \alpha_{S_T})) \wedge \\
\qquad isSAT(\alpha_{S_T}, \alpha_{S_R}, (t \upharpoonright \alpha_{S_T} \cup \alpha_{S_R}) \frown \langle ev \rangle, map, d))
\end{array}$$

The second term of Theorem 4.4.1 is just another way, more appropriate for a constraint solver, of expressing set inclusion. Briefly, we check that every concrete delay of the implementation (after a certain trace of the specification) is also feasible on the specification after the same trace. If this holds, it means that the set of possible delays of the implementation is a subset of the delays of the specification. The restrictions ($\upharpoonright$) over traces, and the hiding ($\setminus$) performed is to focus our analyses on time-related events (see explanation of Definition 4.4.3).

When implementing the mechanisation described in Theorem 4.4.1, two important assumptions are made with respect to infinity. As a consequence of the type of systems considered in this work (embedded systems whose inputs and outputs are always available, besides the fact that these systems should be free of time-lock – time can always advance) we have two sources of infinity: $\mathscr{T}(S)$ and $elapse_{Con}$ (thus, I_P too).

Our CSP representation of s-DFRSs (specifications) is a non-terminating process: it does not have deadlocks introduced by *STOP*, it does not have deadlocks due to synchronisation, and *SPECIFICATION* is a never ending process. This affirmation can be easily checked using FDR via a deadlock-freedom analysis. Therefore, although the operational model (LTS) of S might be finite, we have an infinite number of traces, since a new event can always be performed.

Therefore, since we have an infinite set of traces, implementing the verification of the second term would lead to a non-terminating program. As we discuss in Section 6.2, this is a common practical problem of reasoning about non-terminating timed systems. Some timed conformance relations adopt some timeout criterion, such as the number of events performed or a time upper bound, whereas other relations take into account techniques for symbolic analysis of time to avoid the explicit definition of a stopping criterion. In this work, we follow the first approach: we limit the time verification to traces of length k, an arbitrarily defined value.

Due to the expected absence of time-lock, and the fact that time is concretely represented in implementations, typically $elapse_{Con}$ might also yield an infinite set. Therefore, as also usually adopted when testing timed-systems, we also limit delays up to an arbitrarily defined timeout (b upper bound for delays). Considering this, the computation of $elapse_{Con}$ is performed by enumerating, with the aid of the techniques discussed in Section 4.5, test-scenario selection and generation, the traces $t1$ that satisfy the conditions stated by Definition 4.4.6. Therefore, in practice, we are concerned with $I\ csptio_{k,b}\ S$: I is in conformance with S considering traces with no more than k events, and delays no greater than b.

4.5 Test-scenario generation and selection

Test scenarios are central elements used to construct test cases, since they describe particular execution flows of the specification we are interested in testing. First, we describe how test scenarios can be systematically generated via refinement checking; then, how test purposes can be used to select test scenarios of interest. Except by the definition of the process $ACCEPT'$, which fits to our purposes, we reuse here the test-scenario generation and test-scenario selection techniques proposed in (NOGUEIRA; SAMPAIO; MOTA, 2014).

To generate test scenarios from a process P, we create a modified version of P (P') that has mark events ($MARK = \{accept.n\}$ for $n \in \mathbb{N}$), which do not belong to the alphabet of P ($MARK \cap \alpha_P = \emptyset$), in the execution flows of P we are interested in testing. To enumerate these flows we rely on refinement checking. In the CSP traces model, $P \sqsubseteq_T P'$ is defined as $\mathscr{T}(P') \subseteq \mathscr{T}(P)$: the traces of P' are required to be a subset of the traces of P. As P' has events (the mark ones) that does not belong to P, clearly this refinement expression does not hold. Therefore, we can use FDR to obtain a counterexample for this refinement. To provide a concrete example, consider the requirement REQ003 of the VM:

- When the system mode is choice, and the coin sensor is false, and the coin sensor was false, and the coffee request button changes to pressed, and the request timer is greater than 30.0, the coffee machine system shall: reset the request timer, assign preparing strong coffee to the system mode.

Let *VM_CSP-TIO* be the CSP-TIO process defined in Code 4.8, and S' a modified version of S such that, as soon as a stable state (performing the event *stableState*) is reached after an execution flow related to $REQ003$, S' performs the mark event $accept.1$ (later, in this section, when defining test purposes, we show how S' is created from its counterpart S). If we use FDR to assess whether $S \sqsubseteq_T S'$ holds, we get a negative answer along with a counterexample (trace $trace1$).

This trace describes the following scenario: initially, the system mode is idle (1) and the machine output is strong coffee (0 – default value). After some delay (not restricted to any constraint, since all *eta* variables are false), the coin sensor becomes true. Although the order of events are input events followed by time ones, the interpretation is that the input was received after the delay represented by the time events. Now, the system resets the request timer and goes to the choice mode (0). This reaction is the one described by the requirement $REQ003$.

Afterwards, the coin sensor becomes false again. The output signals remain the same. When the coffee request button is pressed after a delay that satisfies the constraint mapped to $eta1$ (the request was made at least 30.0 time units after inserting the coin), the system resets again the request timer, besides going to the preparing strong coffee mode (2). This behaviour is the one described by the requirement REQ003. Therefore, just after reaching a stable state, the mark event is performed. As this event does not belong to the alphabet of the unmodified version of process S, this trace does not belong to $\mathscr{T}(S)$, but belongs to $\mathscr{T}(S')$ and, thus, is a counterexample of the refinement expression $S \sqsubseteq_{\mathrm{T}} S'$.

```
trace1 = <
    output.the_system_mode.I_the_system_mode.1.
        the_coffee_machine_output.I_the_coffee_machine_output.0,
    stableState,
    input.the_coffee_request_button.B.false.the_coin_sensor.B.true,
    delay.eta1.B.false.eta2.B.false.eta3.B.false.eta4.B.false,
    reset.the_request_timer, REQ001,
    output.the_system_mode.I_the_system_mode.0.
        the_coffee_machine_output.I_the_coffee_machine_output.0,
    stableState,
    input.the_coffee_request_button.B.false.the_coin_sensor.B.false,
    delay.eta1.B.false.eta2.B.false.eta3.B.false.eta4.B.false,
    output.the_system_mode.I_the_system_mode.0.
        the_coffee_machine_output.I_the_coffee_machine_output.0,
    stableState,
    input.the_coffee_request_button.B.true.the_coin_sensor.B.false,
    delay.eta1.B.true.eta2.B.false.eta3.B.false.eta4.B.false,
    reset.the_request_timer, REQ003,
    output.the_system_mode.I_the_system_mode.2.
        the_coffee_machine_output.I_the_coffee_machine_output.0,
    stableState, accept.1
>
```

The FDR refinement checking algorithm, when looking for counterexamples, yields the shortest one first. Although it is possible to ask FDR to yield more than one counterexample, this does not fit into our purposes since it might find different execution paths that lead to the same trace. Therefore, for our purposes, these different execution paths would comprise the same test scenario.

Nevertheless, we can force FDR to identify new and different counterexamples from the shortest one to the largest one. To accomplish this goal, first, we create a process from the obtained counterexample. This is performed by the auxiliary process *CE*. It recursively iterates over the elements of the trace performing events until it reaches the trace end, when it deadlocks. The process *STOP* is used instead of *SKIP*, since the former does not insert any new event in the traces model.

$$CE(trace) = \mathit{if}\ trace = \langle\rangle\ \mathit{then}\ STOP\ \mathit{else}\ head(trace) \rightarrow CE(tail(trace))$$

To get the next counterexample (test scenario), we check a slightly different refinement expression. Let t be the first yielded counterexample, the second one is obtained verifying whether $S \,\square\, CE(t) \sqsubseteq_{\mathrm{T}} S'$ holds. When we compose the process yielded by *CE* in external choice with S, we add to the traces of the left-hand process the trace t, which ends with the mark event ($accept.1$). Therefore, as this counterexample can no longer be considered a counterexample of the refinement expression, this forces FDR to find a different trace of S that leads to the event $accept.1$. This idea can be repeated n times to generate n counterexamples. When the refinement holds, it means that all counterexamples have been found (all traces leading to the mark event).

$$S \,\square\, CE(t_1) \,\square\, ... \,\square\, CE(t_n) \sqsubseteq_{\mathrm{T}} S'$$

The strategy presented so far can, thus, be used to generate as many as desired test scenarios. Now, we need to understand how to select test scenarios based on the properties we want to observe and test. In other words, how we add mark events to the desired places to create the modified process S' from its counterpart S. To accomplish this goal, we rely on test purposes, which are also CSP processes.

A test purpose describes characteristics that are required to be present in the obtained test scenarios. Regarding CSP specifications, a test purpose specifies the traces that need to be present in the generated test scenarios. Here, we consider Definition 11 of the work (NOGUEIRA; SAMPAIO; MOTA, 2014) (reproduced below as Definition 4.5.1).

Definition 4.5.1. *CSP test purpose: let TP and S be CSP processes, m an event from MARK, and* $X \subseteq (\alpha_S)^*$ *a subset of the traces constructed from* α_S*, where* α_S *denotes the alphabet of S, the process TP is a test purpose for S if it is deterministic and* $\forall\, t : \mathscr{T}(TP) \bullet (t \in X) \vee (t \notin X \wedge t = t' \frown \langle m \rangle \wedge t' \in X)$.

Intuitively, a test purpose *TP* for a CSP process *S* comprises traces (test scenarios of interest) appended with a mark event. To ease the task of writing test purposes in CSP based on Definition 4.5.1, previous work define a set of primitive processes that combined enables the creation of more complicated test purposes (NOGUEIRA; SAMPAIO; MOTA, 2014). Here, we reuse some of these definitions, but we also define one tailored for our purposes (*ACCEPT*').

The process *ACCEPT* is the most basic primitive process for defining test purposes. It receives an integer (*id*), performs the event *accept.id*, and then it deadlocks. Basically, this process is used to create mark events ($accept.id \in MARK$).

channel accept : Int
$ACCEPT(id) = accept.id \rightarrow STOP$

The process *ANY* has two parameters. The first one is a set of events (*events*). The second one is another CSP process (*NEXT*). After performing any event that belongs to *events*, it behaves as the process *NEXT*.

$ANY(events, NEXT) = \Box\ ev : events\ @\ ev \rightarrow NEXT$

To explain the process *UNTIL*, we need to understand the auxiliary process *RUN* first: it continuously offers events that belongs to *events*. Then, the process *UNTIL* offers indefinitely the events that belongs to *events*1 but not to *events*2 ($diff(events1, events2)$). When some event from *events*2 is performed this behaviour is interrupted, and *UNTIL* behaves as *NEXT*.

$RUN(events) = \Box\ ev: events\ @\ ev \rightarrow RUN(events)$
$UNTIL(events1, events2, NEXT) = RUN(diff(events1,events2)) \triangle ANY(events2, NEXT)$

These processes are defined in (NOGUEIRA; SAMPAIO; MOTA, 2014). Here, we also propose the auxiliary process *ACCEPT'* tailored for CSP-TIO processes. It ensures that a mark event is only inserted after reaching a stable state. Otherwise, the test scenario would contain the information of the stimuli provided for the system (inputs), but not necessarily the system reaction (outputs).

$ACCEPT'(alphabet, id) = UNTIL(alphabet, \{stableState\}, ACCEPT(id))$

This process (*ACCEPT'*) is defined in terms of the *UNTIL* process. It continuously offers events of the alphabet of the CSP-TIO process until a stable state is reached, when it behaves as $ACCEPT(id)$ (inserts a mark). Finally, to modify an original process (*P*) to insert a mark, we compose the process with a test purpose (*TP*) using the CSP interface parallel operator ($P \underset{\alpha_P}{\|} TP$). In CSP$_M$ the auxiliary process *PP* creates this parallelism ($[|\ \alpha\ |]$ is the CSP$_M$ syntax for $\underset{\alpha}{\|}$).

$PP(alphabet, P1, P2) = P1\ [|\ alphabet\ |]\ P2$

Using these processes, we can easily elaborate a test purpose to select test scenarios that cover certain requirements. As shown in Section 4.2, the process *SYSTEM_BEHAVIOUR* has events named after the requirements identifiers (e.g., *REQ*003) and, thus, it is possible to trace to the system requirements the location where the system is exhibiting the behaviour of a certain requirement. For example, in Code 4.9 we can see the test purpose that selects test scenarios that cover the requirement REQ003 (line 1), besides how the process S', previously considered when explaining the generation of test scenarios, is defined (line 2).

Code 4.9: CSP – test purpose (vending machine)

```
TP_REQ003 = UNTIL(S_alphabet, {REQ003}, ACCEPT'(S_alphabet, 1))
S' = PP(S_alphabet, S, TP_REQ003)
```

[Source: author]

Although we have presented how to select and generate test scenarios to cover particular requirements, it is a weak coverage criterion. For instance, as one can observe analysing the previously shown counterexample (*trace*1), besides covering requirement REQ003 it also covers requirement REQ001. A possible more interesting criterion would be to consider the structure of the operational model (LTS) obtained from a CSP process. In that way, we could take into account structural criteria such as node and transition coverage. However, this is outside the current scope of our research.

4.6 Phase V – sound test-case generation

In Section 4.5, we have explained how test scenarios can be selected and incrementally generated from CSP-TIO specifications. Now, we explain how we build symbolic test cases from test scenarios. These test cases are called symbolic, since they are built from test scenarios that deal with a symbolic time representation. We show that the way we build these test cases is sound with respect to csptio (Definition 4.4.7). In other words, if the test execution fails, it implies that the implementation does not conform to the specification.

A symbolic test case is also a CSP-TIO process (Definition 4.6.1). It interacts with the implementation CSP-TIO process to indicate whether it is a valid implementation with respect to a specification according to csptio.

Definition 4.6.1. *Test case CSP-TIO process: let I and S be CSP-TIO processes of an implementation and a specification, respectively; a test case generated from S to test I is a CSP-TIO process $TC = (TC_P, \alpha_{TC_I}, \alpha_{TC_O}, \alpha_{TC_T}, \alpha_{TC_R}, \alpha_{TC_A})$, such that $\alpha_{TC_I} \subseteq \alpha_{I_O}$, $\alpha_{TC_O} \subseteq \alpha_{I_I} \cup VERD$, $\alpha_{TC_T} \subseteq \alpha_{I_T}$, $\alpha_{TC_R} = \alpha_{TC_A} = \emptyset$, where $VERD = \{pass, fail, inconclusive\}$.*

The outputs of a test case are the inputs provided for an implementation, besides the test case verdict ($VERD = \{pass, fail, inconclusive\}$), whereas the inputs of a test case are the outputs generated by the implementation. Therefore, we have an alphabet inversion. Regarding time events, a test case share the same time events of the implementation, since it has to assess whether the performed delays are expected with respect to the specification. As implementations do not have reset and auxiliary events, as previously explained, these sets are also empty in test cases. These restrictions over the constituent elements of the test-case alphabet are also formalised in Definition 4.6.1.

The execution ($EXEC(I, TC)$) of a test case (TC) against an implementation (I) is formalised as the parallel synchronisation of the test case (TC_P) and the implementation (I_P) CSP processes.

$$EXEC(I, TC) = I_P \underset{\alpha_{I_I} \cup \alpha_{I_O} \cup \alpha_{I_T}}{\|} TC_P$$

The execution of a test case might lead to three distinct verdicts: pass, fail or inconclusive ($VERD = \{pass, fail, inc\}$). It is said to pass when the implementation behaviour is coherent with the specification. It is said to fail when the behaviour is not coherent. An inconclusive verdict is reached when the implementation exhibits a behaviour that is correct with respect to the specification, but is not the one being assessed by the test case being currently executed. In our testing theory, we define CSP processes for these three verdicts.

channel pass, fail, inc
$PASS = pass \rightarrow STOP$
$FAIL = fail \rightarrow STOP$
$INC = inc \rightarrow STOP$

To verify the presence of a verdict $v \in VERD$ in the execution of a test case, we can verify the following refinement expression. After hiding all events of the test case (note that $\alpha_{TC_I} \subseteq \alpha_{I_O}$, $\alpha_{TC_O} \subseteq \alpha_{I_I}$, $\alpha_{TC_T} \subseteq \alpha_{I_T}$), but the events related to verdicts ($v \in VERD$), we check whether $v \rightarrow STOP$ is a trace of the execution. If it is, then this verdict v is reached by the execution of the given test case.

$$VER(v, I, TC) \equiv EXEC(I,TC) \setminus (\alpha_{I_I} \cup \alpha_{I_O} \cup \alpha_{I_T}) \sqsubseteq_T v \rightarrow STOP$$

Now, we explain how we create a test case from a given test scenario. The CSP process *T_TC_BUILDER* (timed test-case builder) yields the process component of a test case (TC_P). Its parameters are: S, I, *atrace*, *timeBinding*, and *timeEval*. The first two parameters are the CSP-TIO processes of the specification and the implementation.

The third parameter (*atrace*) is an annotated trace created from a test scenario. First, auxiliary events must be removed from test scenarios, since they are only used within the process of selecting and generating test scenarios: let *ts* be a test scenario, the corresponding annotated trace is built from $ts \upharpoonright (\alpha_S \setminus \alpha_{S_R})$. For each event of the restricted test scenario, two additional pieces of information are recorded. Therefore, each element of *atrace* is a tuple $(ev_i, outs_i, delays_i)$, where ev_i stands for the event, $outs_i$ is the set of output events the specification performs after the trace $\langle ev_1, ..., ev_{i-1} \rangle$, including ev_i if it is an output of the specification ($ev_i \in \alpha_{S_O}$). Similarly, $delays_i$ keeps the time events (α_{S_T}) of the specification after the trace $\langle ev_1, ..., ev_{i-1} \rangle$. However, differently from $outs_i$, even if ev_i is a time event ($ev_i \in \alpha_{S_T}$), it does not belong to $delays_i$. To construct annotated traces from test scenarios we rely on Algorithm 1 presented in (NOGUEIRA; SAMPAIO; MOTA, 2014), lifted to record time events as well.

The last two parameters of *T_TC_BUILDER* are related to testing the time-related behaviour. The variable *timeBinding* is a set of pairs (n, v), where n is the global clock or the timers defined in the specification, and v their current value. For instance, in the VM, the initial configuration of *timeBinding* is $\{(gc, 0), (the\_request\_timer, 0)\}$. The last parameter (*timeEval*) is similar to the function $map : \alpha_{S_T} \rightarrow String$ used for *elapse_Sym*: it maps each symbolic time event to a function that, provided a time binding, assesses whether a time constraint is satisfied. In what follows, we present a fragment of the *timeEval* definition for the VM example.

```
eval_default(binding) = true
eval_eta1(binding) =
    if getValue(binding, gc) - getValue(binding, the_request_timer) > 30
    then true else false
...
timeEval = {
    (delay.eta1.B.false.eta2.B.false.eta3.B.false.eta4.B.false, eval_default),
    (delay.eta1.B.true.eta2.B.false.eta3.B.false.eta4.B.false, eval_eta1),
    ...
    (delay.eta1.B.false.eta2.B.false.eta3.B.false.eta4.B.true, eval_eta4)
}
```

The function *eval_default*, which always yields true, is mapped to the time event that denotes time evolution with no constraints. In other words, in such a situation, any delay satisfies the constraints, since there are none. Differently, the function *eval_eta*1, considering the provided *binding*, evaluates whether the difference between the value of the global clock and the request timer is greater than 30. Note that the function *eval_eta*1 is mapped to the time event where *eta1* is true, and that the time constraint associated to this *eta* variable is exactly the one just described (see Code 4.6). The auxiliary function *getValue*, considering a given *binding*, yields the value of the provided variable. The set passed as argument to pick is always a singleton, since the name of the variables and time events are unique.

```
pick({element}) = element
getValue(binding, name) = pick({value | (n, value) <- binding, n == name})
```

Within the definition of *T_TC_BUILDER*, some other auxiliary functions are needed. Given two mappings such as the ones previously described (*timeEval* and *timeBinding*) and a symbolic time event, the function *eval* analyses whether the time constraint mapped to the time event evaluates to true in the provided time binding.

```
eval(mapping, binding, event) = getValue(mapping, event)(binding)
```

Given a concrete time delay, the function *evolveTime* yields a new time binding by advancing the global clock by this delay. The function *resetTimer* also yields a new time binding, but by resetting the value of a given timer. We recall that the reset operation is defined as assigning to the timer the current value of the global clock.

```
evolveTime(binding, delay) =
    union(
        {(var, value) | (var, value) <- binding, var != gc},
        {(var, value + delay) | (var, value) <- binding, var == gc}
    )

resetTimer(binding, timer) =
    union(
        {(var, value) | (var, value) <- binding, var != timer},
        {(var, getValue(binding, gc)) | (var, value) <- binding, var == timer}
    )
```

Based on the definitions presented so far, we are now able to define how to build sound test cases via the process *T_TC_BUILDER* (Definition 4.6.2). This CSP process iterates recursively over each tuple $(ev, outs, delays)$ of the given annotated trace, and it yields a pass verdict when the last element of the a trace is reached.

If *ev* is a reset event, after resetting the timer mentioned by *ev* (*resetTimer*(...) – note that we use here the function *getTimerName* explained in Algorithm 17), the test case behaves recursively as *T_TC_BUILDER* considering the tail of the annotated trace.

If *ev* is not a time-related event ($ev \notin \alpha_{S_R}$ and $ev \notin \alpha_{S_T}$), it is an input or an output of the specification (an output or an input of the test case, respectively, due to the alphabet inversion explained before). If it is a specification output ($ev : \alpha_{S_O}$), there are three possible scenarios:

- the output is the expected one, and the process behaves as *T_TC_BUILDER* considering the annotated trace tail (first external choice);
- the output is not the one expected by the current test case, but it is a possible output according to the specification and, thus, an inconclusive verdict is reached ($ANY(outs \setminus \{ev\}, INC)$);
- the output is not the one expected by the current test case, neither one admissible by the specification and, thus, a fail verdict is reached ($ANY(\alpha_{I_O} \setminus outs, FAIL)$).

If *ev* is a specification input (a test case output), it is communicated to the implementation and the test case continues considering the annotated trace tail (first external choice). Remember that the implementation alphabet is compatible with the specification alphabet, besides being input-enabled.

Definition 4.6.2. *Definition of T_TC_BUILDER: let I and S be CSP-TIO processes of an implementation and a specification, respectively, ev an event, outs* $\subseteq \alpha_{S_O}$ *and delays* $\subseteq \alpha_{S_T}$ *sets of events, timeBinding a set of pairs* (n, v)*, where n is the name of the system global clock or of a system timer, and v the value associated with this name, and timeEval a function from* α_{S_T} *to* *time evaluation functions, as described earlier. Then, T_TC_BUILDER is defined as follows:*

```
$T\_TC\_BUILDER(S, I, \langle\rangle, timeBinding, timeEval) = PASS$
$T\_TC\_BUILDER(S, I, \langle(ev, outs, delays)\rangle \frown tail, timeBinding, timeEval) =$
    $if\ ev \in \alpha_{S_R}\ then$
        $T\_TC\_BUILDER(S, I, tail,$
            $resetTimer(timeBinding, getTimerName(ev)), timeEval)$
    $else$
        $ev \notin \alpha_{S_T}\ \&\ ev \rightarrow T\_TC\_BUILDER(S, I, tail, timeBinding, timeEval)$
        $\Box$
        $ev \in \alpha_{S_O}\ \&\ ANY(outs \setminus \{ev\}, INC)\ \Box\ ANY(\alpha_{I_O} \setminus outs, FAIL)$
        $\Box$
        $\Box\ ev_{time} \in \alpha_{I_T} \bullet ev_{time} \rightarrow$
            $if\ eval(timeEval, evolveTime(timeBinding, prj(ev_{time})), ev)\ then$
                $T\_TC\_BUILDER(S, I, tail,$
                    $evolveTime(timeBinding, prj(ev_{time})), timeEval)$
            $else$
                $if\ true \in \{d : delays \bullet$
                    $eval(timeEval, evolveTime(timeBinding, prj(ev_{time})), d)\}$
                $then\ INC\ else\ FAIL$
```

In the case *ev* is not a specification input, output or reset event, it is a time event. As the implementation represents time concretely, the test case needs to be ready to engage on any communication over the possible implementation time delays ($\Box\ ev_{time} \in \alpha_{I_T} \bullet ev_{time}$). This is the underlying reason for stating that the time events of a test case is equal to the time events of the implementation being tested ($\alpha_{TC_T} = \alpha_{I_T}$ – see Definition 4.6.1).

After synchronising on this event (ev_{time}), the test case performs the following analyses, which are analogous to the ones performed with respect to specification output events.

- the implementation delay (ev_{time}) is coherent with the current time event (*eval*(...) evaluates to true), and, thus, after evolving the time by the implementation delay (*evolveTime*(...)), the test case behaves as *T_TC_BUILDER* considering the annotated trace tail;
- the implementation delay (ev_{time}) is not coherent with the current time event of the test case (*ev*), but it is a possible delay according to the specification (*true* ∈ {...}), and, thus, an inconclusive verdict is reached (*INC*);
- the implementation delay is not coherent with the current time event of the test case (*ev*), besides being not a possible one according to the specification, and, thus, a fail verdict is reached (*FAIL*).

To illustrate the usage of *T_TC_BUILDER*, consider the following situations:

- An implementation that always dispenses weak coffee for any given delay between inserting the coin and pressing the coffee request button, and an annotated trace built from a test scenario where a strong coffee is dispensed. In such a situation, when testing the dispense of strong coffee (i.e., expecting the implementation to perform the event *ev*), if the implementation is ready to perform the same event *ev*, the test case passes. However, it is not going to be the case, since the implementation always dispenses weak coffee. An inconclusive verdict is also not applicable, since in such a situation the only admissible output according to the specification is strong coffee and, thus, $outs \setminus \{ev\} = \emptyset$. Therefore, when the implementation performs the event representing the dispense of weak coffee, as this event is different from *ev*, the test case fails since this event of the implementation belongs to $\alpha_{I_O} \setminus outs$.

- An implementation that has a faulty behaviour such that, after inserting the coin, if the user does not press the coffee request button within 10s, the machine assumes the button was pressed anyway; and an annotated trace built from a test scenario where a strong coffee is dispensed. In such a situation, an inconclusive verdict is reached, since the implementation, after receiving the coin, always performs events denoting time elapsing lower than or equal 10s, which are admissible events by the specification, but are not the events this particular test case wants to observe (time elapses greater than or equal to 30s, such that strong coffee is dispensed).

- An implementation that always dispenses weak coffee within 15s after receiving the user request, and an annotated trace built from a test scenario were a where a weak coffee is dispensed. In such a situation, the execution of the test case reaches a pass verdict, since the implementation time elapse (15s) is admissible with respect to the implementation, which says that weak coffee is dispensed within 10s to 30s after the request.

- An implementation that always dispenses weak coffee within 31s after receiving the user request, and an annotated trace built from a test scenario were a where a weak coffee is dispensed. In such a situation, the execution of the test case reaches a fail verdict, since the implementation time elapse (31s) is not admissible with respect to the implementation, which says that weak coffee is dispensed within 10s to 30s after the request.

Now, we define what is a sound test case: if a test execution leads to a fail verdict, it follows that the implementation does not conform to the specification according to csptio (Definition 4.6.3).

Definition 4.6.3. *Sound test case: let I and S be CSP-TIO processes of an implementation and a specification, respectively, TC a CSP-TIO test case, then TC is a sound test case if $VER(fail, I, TC) \Rightarrow \neg\ (I\ csptio\ S)$.*

Theorem 4.6.1 states that *T_TC_BUILDER* is sound. In other words, a CSP-TIO process yielded by *T_TC_BUILDER* is sound; it does not deliver a false fail verdict.

Theorem 4.6.1. *Soundness of T_TC_BUILDER: let I and S be CSP-TIO processes of an implementation and a specification, respectively, atrace an annotated trace obtained from S with no auxiliary events, timeBinding and timeEval elements as explained before, then $TC = T\_TC\_BUILDER(S, I, atrace, timeBinding, timeEval)$ is a sound test case, such that $\alpha_{TC_I} = \alpha_{I_O}$, $\alpha_{TC_O} = \alpha_{S_I}$, and $\alpha_{TC_T} = \alpha_{I_T}$.*

We provide here a proof sketch of Theorem 4.6.1. According to Definition 4.6.2, a fail verdict occurs in one of two situations: after a given trace (t), the implementation output is not a valid one with respect to the specification or the implementation concrete time delay is not a valid delay considering the specification symbolic time event. If the first situation is true, we have that $\neg\ (out(I,t) \subseteq out(S,T))$ and, thus, $\neg$ *(I csptio S)*. The second situation is the one that $\neg\ (elapse_{Con}(I, S, t, proj, map) \subseteq elapse_{Sym}(S, t, map))$ and, thus, $\neg$ *(I csptio S)* as well.

4.7 Concluding remarks

Initially, after a brief overview of the process algebra CSP, this chapter has explained how DFRSs can be encoded as CSP processes. Then, we have formally defined a CSP-TIO process: a tuple composed by the CSP process obtained from a DFRS, along with its distinct input, output, time, reset and auxiliary alphabets.

Afterwards, considering CSP-TIO processes, we have proposed a CSP timed input-output conformance relation (csptio), whose sound mechanisation is supported by a refinement checker (FDR) and a constraint solver (Z3). Then, we have explained how test scenarios can be systematically selected and generated from CSP-TIO processes. These test scenarios are used to generate symbolic test cases within our formal testing theory, which is based on csptio. Furthermore, we have also proved that our process of generating symbolic test cases is sound with respect to csptio.

5 Tool support and empirical evaluation

The NAT2TEST strategy, which provides means for generating test cases from natural-language requirements, is automated by the NAT2TEST tool (CARVALHO et al., 2015). In this chapter, firstly, we describe the examples considered to evaluate the NAT2TEST strategy with the aid of its tool support (Section 5.1). Afterwards, we present the constituent components of the NAT2TEST tool (Section 5.2):

- CNL-Parser (Section 5.2.1): verifies whether the system requirements are written in accordance with the SysReq-CNL grammar.
- RF-Generator (Section 5.2.2): identifies requirement frames from the syntax trees yielded by the CNL-Parser.
- DFRS-Generator (Section 5.2.3): derives an s-DFRS from requirements frames, besides guiding the creation of the corresponding e-DFRS.
- CSP_M-Generator (Section 5.2.4): encodes DFRSs as CSP_M processes.
- TC-Generator (Section 5.2.5): generates test cases from the CSP representation.
- SCR-Generator (Section 5.2.6): generates test cases using an alternative representation (SCR) with the aid of the commercial tool T-VEC[1].
- IMR-Generator (Section 5.2.7): generates test cases using an alternative representation (IMR) with the aid of the commercial tool RT-Tester[2].

The VM example is used to exemplify the tool features. Finally, we present and discuss the empirical analyses performed to evaluate the NAT2TEST tool (Section 5.3). We analyse two aspects: (i) performance (Section 5.3.1), and (ii) the ability to detect defects by means of mutation analysis (Section 5.3.2).

[1]http://www.t-vec.com/solutions/products.php
[2]http://www.verified.de/en/products/rt-tester

Ideally, we would like to have evaluated the NAT2TEST strategy considering real (embedded) implementations. However, as two of the considered examples are from the literature, and regarding the other two we did not have access to their implementations due to security policies, we systematically created reference Java programs following the natural-language requirements, and used the generation of mutants as means for systematic introduction of errors. In such a analysis, we consider as a baseline the generation and execution of random tests using Randoop (PACHECO et al., 2007).

Moreover, to provide an empirical argument to whether the DFRS models are expressive enough to represent the behaviour of a timed reactive system as defined using natural language, we assess whether test cases, either independently written by domain specialists from industry or generated by a commercial tool (RT-Tester) from the same set of requirements, are compatible with the corresponding DFRS models (Section 5.3.3).

5.1 Considered examples of critical systems

We evaluate the NAT2TEST tool considering examples from the literature, but also from the aerospace and the automotive industry: (i) a vending machine (VM – toy example); (ii) a simplified control system for safety injection in a nuclear power plant (NPP – publicly available), (iii) the priority command function (PC – provided by Embraer[3], our industrial partner); and (iv) part of the turn indicator system of Mercedes vehicles (TIS – publicly available). In what follows, we describe these examples.

5.1.1 Vending machine

As a toy example, we consider a vending machine (VM) that is an adaptation of the coffee machine presented in (LARSEN; MIKUCIONIS; NIELSEN, 2004). Initially, the VM is in an *idle* state. When it receives a coin, it goes to the *choice* state. After inserting a coin, when the coffee option is selected, the system goes to the *weak* or *strong* coffee state. If coffee is selected within 30 seconds after inserting the coin, the system goes to the *weak coffee* state. Otherwise, it goes to the *strong coffee* state. Therefore, if the user selects the coffee option too quickly (less than 30s), a weak coffee is dispensed instead of a strong one.

The time required to produce a weak coffee is different from that of a strong coffee. The machine outputs a weak coffee between 10 to 30 seconds after a user request, whereas 30 to 50 seconds are necessary for a strong coffee. After producing a weak or strong coffee, the system goes back to the idle state.

This system has two input signals (one that detects when a coin is inserted, another to detect when the request button is pressed) and two output signals (the current system mode, and

[3]*Empresa Brasileira de Aeronáutica*
http://www.embraer.com/en-us/pages/home.aspx

which type of coffee should be produced).

5.1.2 Nuclear power plant

We consider a simplified version of a control system for safety injection in a nuclear power plant (NPP) as described in (LEONARD; HEITMEYER, 2003). If the water pressure is too low (less than 900 units), the system injects coolant into the reactor, otherwise there is no need to inject coolant. Two switches also comprise the described system: one that overrides safety injection, and another that resets the system after blockage.

The work reported in (LEONARD; HEITMEYER, 2003) presents a formal definition of this system using the SCR notation. Considering this formal specification, we wrote requirements in natural language, which are considered here as input to our test generation strategy.

This system has three input signals (the actual water pressure, a switch to block the injection of coolant, and a switch that reset the system after blockage) and three output signals (the safety injection mode, the current blockage mode, and the pressure mode). Differently from the VM, this example does not have time-dependent behaviour.

5.1.3 Priority command

The priority command function (PC) decides whether the pilot or copilot will have priority in controlling the airplane side sticks. The system monitors whether the pilot and copilot side sticks are in the neutral position, and whether the side stick priority button has been pressed. Taking into account this information, a control logic is applied to decide who is going to have priority. This example was provided by Embraer, our industrial partner.

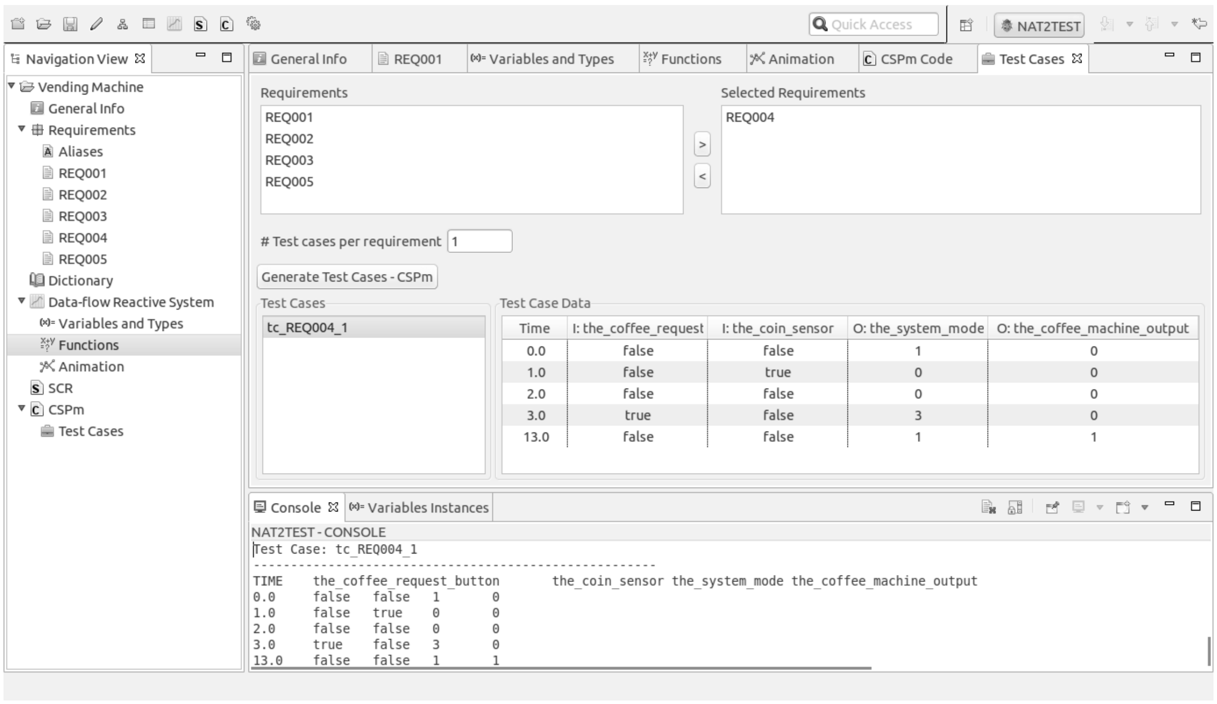
This system has four input signals (the stick position and the status of the priority button for both the pilot and the co-pilot) and one output signal (a priority command). Although the order the events (stick movement and priority request) occur is relevant, similarly to the NPP example, the actual time the events occur is not relevant to determine the system behaviour.

5.1.4 Turn indicator system

We also considered a simplification of the turn indicator system specification that is currently used by Daimler for automatically deriving test cases, concrete test data and test procedures. In 2011 Daimler allowed the publication of this specification to serve as a “real-world” benchmark supporting research of MBT techniques.

Our simplification results in a size reduction of the original model presented in (PELESKA et al., 2011), but serves well as a proof of concept, because it still represents a safety-critical system portion with real-time and concurrent aspects. The system has three inputs: (1) the turn indicator lever, which may be in the idle, left or right position; (2) the emergency flashing button; and (3) the battery voltage. The system outputs are the car flashing lights.

Figure 5.1: The NAT2TEST tool



Requirements	Selected Requirements
REQ001	REQ004
REQ002	
REQ003	
REQ005	

Test cases per requirement: 1

Test Cases: tc_REQ004_1

Test Case Data

Time	I: the_coffee_request	I: the_coin_sensor	O: the_system_mode	O: the_coffee_machine_output
0.0	false	false	1	0
1.0	false	true	0	0
2.0	false	false	0	0
3.0	true	false	3	0
13.0	false	false	1	1

```
NAT2TEST - CONSOLE
Test Case: tc_REQ004_1
------------------------------------------------------
TIME    the_coffee_request_button       the_coin_sensor the_system_mode the_coffee_machine_output
0.0     false   false   1       0
1.0     false   true    0       0
2.0     false   false   0       0
3.0     true    false   3       0
13.0    false   false   1       1
```

[Source: author]

The simplified TIS comprises two parallel components: the *flashing mode* component, which is responsible for controlling the system flashing state (only left or right lights flashing, left and right lights flashing, left or right tip flashing, and no lights flashing), and the *lights controller* component, which is responsible for turning on and off the flashing lights respecting the flashing periods (320 milliseconds on and 240 milliseconds off).

5.2 NAT2TEST tool

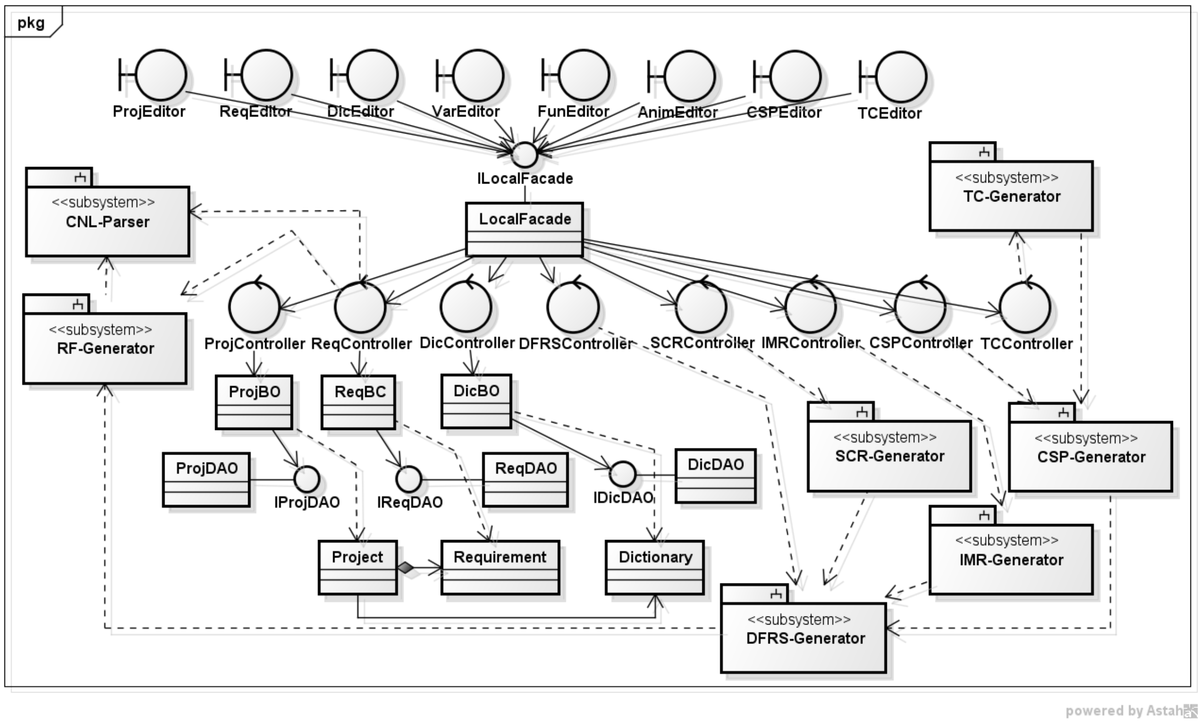
The NAT2TEST tool is written in Java (it is multi-platform), and its GUI is built using the Eclipse RCP[4] framework, which provides means to create client-side applications quickly using a collection of plug-ins. Figure 5.1 shows the tool interface.

Each phase of the NAT2TEST strategy, presented in Section 1.4, is realised by a different component. Figure 5.2 shows an analysis diagram of the tool architecture, which follows a traditional layered structure – presentation, business, and data layers. The first layer comprises screens (e.g., ProjEditor, ReqEditor, etc.) that interact with the business layer via the LocalFacade. Despite that, its porting to a web-based environment would be straightforward. It would suffice to create a web-based GUI that would interact with the LocalFacade via some networking protocol.

The business layer has a set of controllers that are responsible for interacting with the

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

Figure 5.2: The NAT2TEST tool architecture



[Source: author]

components that realise each phase of the NAT2TEST strategy. Besides that, it also persists data (i.e., projects, requirements and dictionaries) via Business Objects (BOs) and Data Access Objects (DAOs). We do not persist other elements (e.g., the s-DFRS model), as they can be automatically derived from the requirements within seconds. In the following sections we describe each component in terms of implementation details and functionalities provided.

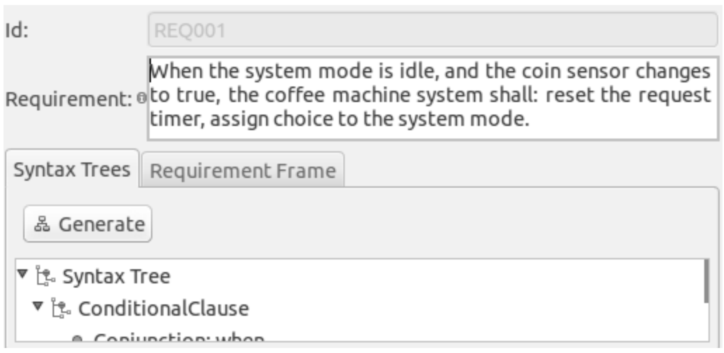
5.2.1 CNL-Parser – Phase I

The CNL-Parser is responsible for analyzing the system requirements according to the SysReq-CNL grammar, yielding the corresponding syntax trees. However, before the syntactic analysis takes place, it is necessary to classify each word in the input requirement into its corresponding lexical class. Thus, the syntactic analysis phase also includes the morphological analysis level, performing a morphosyntactic analysis of the input requirements. Note that, in linguistics, the morphosyntactic analysis defines the POS categories of words using criteria from both morphology and syntax fields.

In NLP, the algorithm that performs this analysis is known as a POS-Tagger (ALLEN, 1995). Unlike programming languages, in natural language the same lexeme may bear more than one classification (lexical ambiguity). For example, *off* can be an adverb, a preposition or an adjective.

In this work, we implemented a customized POS-Tagger that searches all possible clas-

Figure 5.3: NAT2TEST tool – parsing requirements



Id: REQ001

Requirement: When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode.

[Source: author]

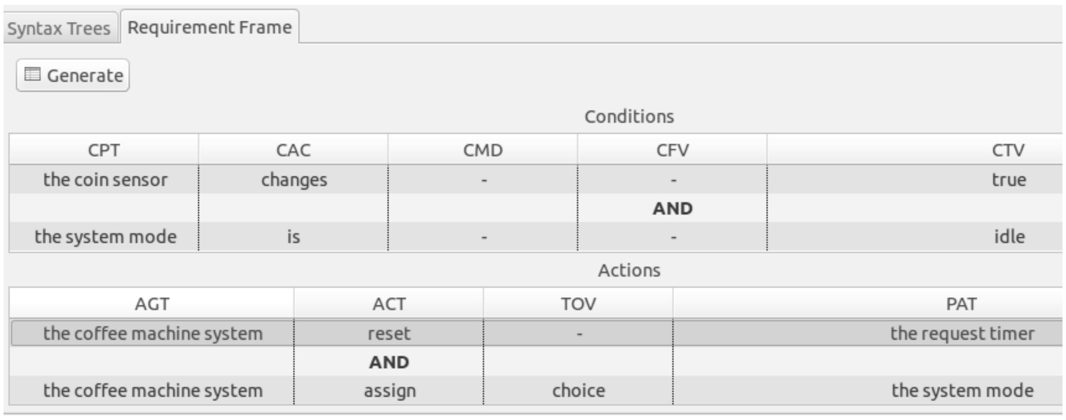
sifications of each lexeme. Let a clause be composed of "*word*$_1$ ... *word*$_k$ ... *word*$_n$"; if the word that is being currently analyzed is the k-th word, our POS-Tagger verifies for each group of words, starting from the k-th word ("word$_k$", "word$_k$ word$_{k+1}$", ..., "word$_k$ ... word$_n$"), whether it is a lexeme in the SysReq-CNL lexicon. For instance, consider the following fragment: "... *according to* ...". Let the current word under analysis be "according". First, the POS-Tagger will try to classify just "according", but it also tries "according to", and so on. In this case, the valid classification will be obtained for the group "according to", which is a preposition.

The lexemes and their categories are given as input to the CNL-Parser. In this work, we implemented a version of the Generalized LR (GLR) parsing algorithm (TOMITA, 1986). It is important to emphasize that this algorithm is more than a simple parser, but actually a parser generator. In other words, one can update the SysReq-CNL and no extra change is required in the code, since the algorithm implemented will generate a suitable parser automatically. Moreover, as the SysReq-CNL is written in EBNF and the GLR parsing algorithm expects as input a grammar in Backus-Naur Form (BNF), we have also implemented an automatic translator between these two notations.

The GLR algorithm generalizes the traditional Look-Ahead LR (LALR) parser (see (AHO et al., 2006)) algorithm to handle non-deterministic and ambiguous grammar. This kind of algorithm is required since the natural support for lexical ambiguity can enable non-deterministic and ambiguous scenarios. This way, the parser may generate more than one syntax tree per input requirement. In this case, the NAT2TEST strategy terminates, and the requirement analyst shall manually remove the ambiguity, and thus define which syntax tree shall be considered as the correct one.

When writing requirements, the NAT2TEST tool also provides other functionalities, such as editing the domain-specific dictionary, besides defining and referencing aliases to promote text reuse, as previously mentioned. Furthermore, the tool is also capable of assisting the user while writing the requirements by informing the next expected grammatical classes. Figure 5.3 shows the NAT2TEST tool graphical interface for editing requirements; in particular, considering the requirement REQ001 of the VM example.

Figure 5.4: NAT2TEST tool – inferring thematic roles


Syntax Trees | Requirement Frame

Generate

Conditions

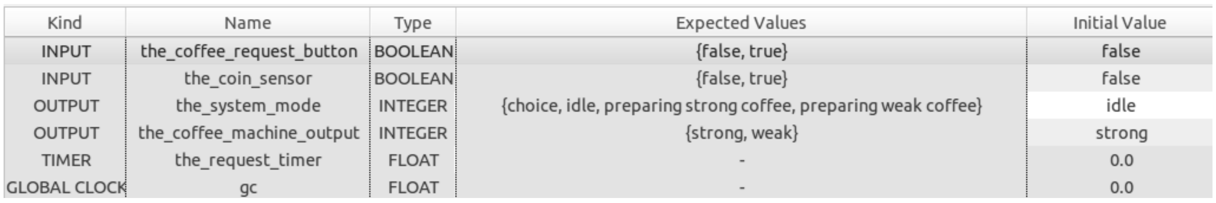
CPT	CAC	CMD	CFV	CTV
the coin sensor	changes	-	-	true
			AND	
the system mode	is	-	-	idle

Actions

AGT	ACT	TOV	PAT
the coffee machine system	reset	-	the request timer
	AND		
the coffee machine system	assign	choice	the system mode

[Source: author]

Figure 5.5: NAT2TEST tool – editing initial value of DFRS variables


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

[Source: author]

5.2.2 RF-Generator – Phase II

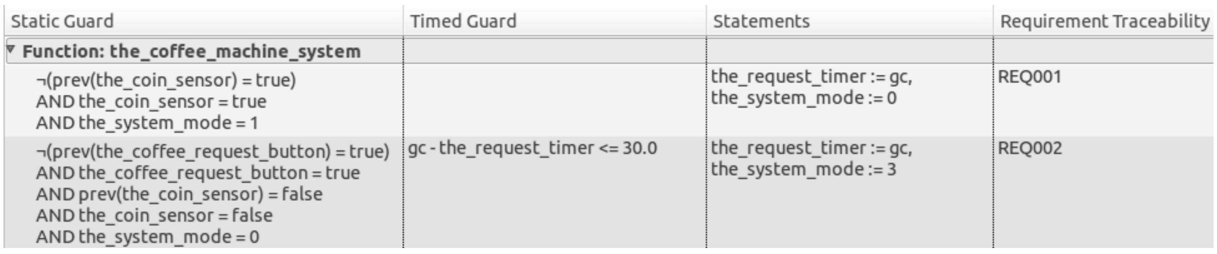
The RF-Generator component is implemented using the visitor design pattern to analyse the syntax trees, considering the inference rules defined in Section 2.2. These rules are used to associate each word, or group of words, identified in the syntax tree with the corresponding thematic roles. Besides that, this component also verifies whether the contextual rules also presented in Section 2.2 are not violated. If they are, the user is alerted and asked to perform the appropriate editing. Figure 5.4 shows the requirement frame derived from the requirement REQ001 of the VM example.

5.2.3 DFRS-Generator – Phase III

The DFRS-Generator component is the one that implements the algorithms presented in Section 3.2. Figure 5.5 shows the inferred variables, along with their types for the VM. The tool also allows the user to edit the initial values and, thus, the initial state.

In Figure 5.6 one can see part of the function obtained from the VM requirements. It is important to note that the tool keeps traceability information between the requirements and the function entries. We also note that there is some syntactic sugar to prevent a verbose representation. For instance, *previous* is reduced to *prev*, and *current* is simply hidden. Moreover,

Figure 5.6: NAT2TEST tool – viewing DFRS' functions and traceability information



Static Guard	Timed Guard	Statements	Requirement Traceability
Function: the_coffee_machine_system			
¬(prev(the_coin_sensor) = true) AND the_coin_sensor = true AND the_system_mode = 1		the_request_timer := gc, the_system_mode := 0	REQ001
¬(prev(the_coffee_request_button) = true) AND the_coffee_request_button = true AND prev(the_coin_sensor) = false AND the_coin_sensor = false AND the_system_mode = 0	gc - the_request_timer <= 30.0	the_request_timer := gc, the_system_mode := 3	REQ002

[Source: author]

the format of the timed guard, as well as the assignment of timers, show explicitly how timers are dealt with by our strategy: the reset of a timer is encoded as assigning the value of *gc* to the timer, and comparisons concerning the timer mean comparing the difference between the current value of *gc* and the timer. More details are provided in Section 3.3.2, where we explain how to obtain an e-DFRS from a symbolic one.

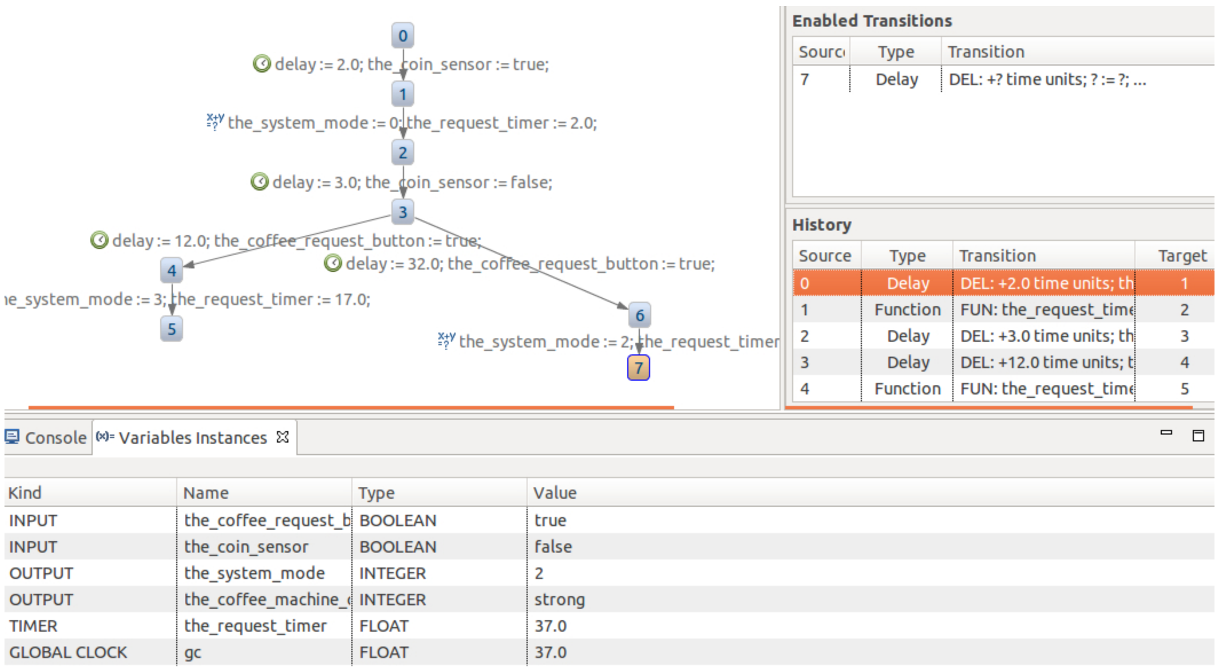
The tool also supports validation of the requirements by animating the s-DFRS; in other words, by manually exploring the state space of the corresponding e-DFRS. It is accomplished by an implementation of the function *genTransitions* (see Section 3.3.2), allowing us to create and explore the states of an e-DFRS dynamically (see Figure 5.7). When the *animator* screen is opened, it automatically creates the e-DFRS initial state (state 0): the initial state of the corresponding s-DFRS, which is obtained from the system requirements.

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

Figure 5.7 illustrates part of the e-DFRS for the example shown in Figure 3.3, but it also describes the transition that leads to the production of strong coffee. We note that from the state 3, if the coffee request button is pressed 12 seconds after inserting the coin, the system goes to the weak mode ($the\_system\_mode := 3$), which is represented by the state 5. Differently, if the request is made 32 seconds after inserting the coin, the system goes to the strong mode ($the\_system\_mode := 2$), which is represented by the state 7.

With the aid of this tool support, we assess whether test cases, either independently written by domain specialists from industry or generated by a commercial tool from the same set of requirements, are compatible with the corresponding DFRS models. We detail this analysis in Section 5.3.3.

Figure 5.7: NAT2TEST tool – dynamic creation of e-DFRSs


0
delay := 2.0; the_coin_sensor := true;
1
the_system_mode := 0; the_request_timer := 2.0;
2
delay := 3.0; the_coin_sensor := false;
3
delay := 12.0; the_coffee_request_button := true;
delay := 32.0; the_coffee_request_button := true;
4
ie_system_mode := 3; the_request_timer := 17.0;
5
6
the_system_mode := 2; the_request_timer
7
Console
Variables Instances

Enabled Transitions

Sourc	Type	Transition
7	Delay	DEL: +? time units; ? := ?; ...

History

Source	Type	Transition	Target
0	Delay	DEL: +2.0 time units; th	1
1	Function	FUN: the_request_time	2
2	Delay	DEL: +3.0 time units; th	3
3	Delay	DEL: +12.0 time units; t	4
4	Function	FUN: the_request_time	5

Kind	Name	Type	Value
INPUT	the_coffee_request_b	BOOLEAN	true
INPUT	the_coin_sensor	BOOLEAN	false
OUTPUT	the_system_mode	INTEGER	2
OUTPUT	the_coffee_machine_	INTEGER	strong
TIMER	the_request_timer	FLOAT	37.0
GLOBAL CLOCK	gc	FLOAT	37.0

[Source: author]

5.2.4 CSP$_M$-Generator – Phase IV

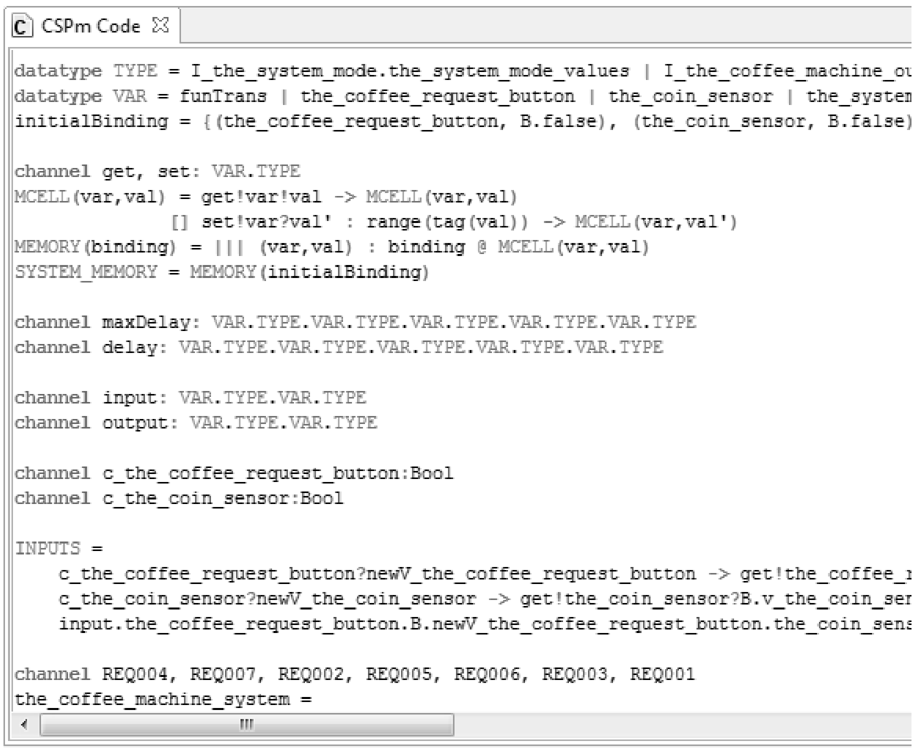
The CSP$_M$-Generator component encodes DFRSs as CSP processes. More specifically, it describes in CSP$_M$ how the expanded DFRS is obtained from the symbolic one. As explained in details in Section 4.2, first, processes are created to represent a shared (global) memory, which comprises the current values of the DFRS inputs and outputs. Time is modelled symbolically to prevent state explosion when compiling the CSP specification and generating the corresponding LTS. When some behaviour depends on the amount of time elapsed, we just assume that the delay occurred satisfies the temporal constraints, and we perform a specific event to represent this assumption. Later, we use Z3 to find concrete values for delays that satisfy these constraints (see Section 5.2.5).

Then, the tool creates a CSP process from the functions of the symbolic DFRS. Here, we also keep traceability with the original requirements by means of events named after the requirements identifier. When these events occur, it implicitly states that the system is presenting the behaviour described by the corresponding requirement. Finally, some auxiliary processes are generated to represent the occurrence of function and delay transitions. Figure 5.8 shows a fragment of the CSP$_M$ code generated for the VM example.

5.2.5 TC-Generator – Phase V

This component generates concrete test cases. The testing theory presented and developed in Chapter 4 has a premise that the implementation model is or can be modelled as a

Figure 5.8: NAT2TEST tool – viewing the generated CSP



```
datatype TYPE = I_the_system_mode.the_system_mode_values | I_the_coffee_machine_ou
datatype VAR = funTrans | the_coffee_request_button | the_coin_sensor | the_system
initialBinding = {(the_coffee_request_button, B.false), (the_coin_sensor, B.false)

channel get, set: VAR.TYPE
MCELL(var,val) = get!var!val -> MCELL(var,val)
               [] set!var?val' : range(tag(val)) -> MCELL(var,val')
MEMORY(binding) = ||| (var,val) : binding @ MCELL(var,val)
SYSTEM_MEMORY = MEMORY(initialBinding)

channel maxDelay: VAR.TYPE.VAR.TYPE.VAR.TYPE.VAR.TYPE.VAR.TYPE
channel delay: VAR.TYPE.VAR.TYPE.VAR.TYPE.VAR.TYPE.VAR.TYPE

channel input: VAR.TYPE.VAR.TYPE
channel output: VAR.TYPE.VAR.TYPE

channel c_the_coffee_request_button:Bool
channel c_the_coin_sensor:Bool

INPUTS =
    c_the_coffee_request_button?newV_the_coffee_request_button -> get!the_coffee_r
    c_the_coin_sensor?newV_the_coin_sensor -> get!the_coin_sensor?B.v_the_coin_ser
    input.the_coffee_request_button.B.newV_the_coffee_request_button.the_coin_sens

channel REQ004, REQ007, REQ002, REQ005, REQ006, REQ003, REQ001
the_coffee_machine_system =
```

[Source: author]

CSP-TIO process. If one desires to test an implementation that is not formally modelled as a CSP-TIO process, the NAT2TEST strategy can also be used, although we cannot guarantee the soundness of the testing process, since soundness is proved with respect to test cases as CSP-TIO processes.

To analyse the behaviour of an implementation that is not a CSP-TIO process, we derive concrete test cases, which can be seen as purely test data, from test scenarios. No test oracle is automatically associated with the generated concrete test cases, since, in our formal testing theory, the oracle is embedded in the process *T_TC_BUILDER* and, thus, test scenarios do not have this information.

Therefore, to use concrete test cases to analyse the behaviour of an implementation, it is necessary to first create test drivers, to send the test data to the implementation and to read the produced outputs from them, besides writing a test oracle. Assuming that, concrete test cases can be provided as input for simulating models (e.g., Simulink models) or testing code (e.g., C code). In this work, we performed this task when evaluating the NAT2TEST strategy, as described in Section 5.3.2.

The generation of concrete test cases is done in two steps: (1) the enumeration of test scenarios via the FDR model checker (as explained in Section 4.5), and (2) the instantiation of time-related events via Z3 (considering Algorithm 17 – Section 4.4.1). The enumeration of test cases is performed with the aid of a TCL[5] script.

Due to the potential large (possibly infinite) number of test cases, it is important to consider some coverage criteria (e.g., maximum number of test cases, coverage of nodes/transitions of the LTS, requirement coverage, among others) to guide the test-generation process. Here, we consider requirement coverage: one can select which requirements should be covered by the generated test cases. To meet this criteria the tool uses the idea explained in Section 4.5. For instance, considering the following requirement (REQ004) of the VM example:

- *When the system mode is preparing weak coffee, and the request timer is greater than or equal to 10.0, and the request timer is lower than or equal to 30.0, the coffee machine system shall: assign idle to the system mode, assign weak to the coffee machine output.*

FDR yields the following trace (sequence of events performed). Note that the symbol "..." is used to abstract some events or parts of a compound one.

$$\begin{aligned}
< & \ output.the\_system\_mode.I.1.the\_coffee\_machine\_output.I.0, ..., \\
& input.the\_coffee\_request\_button.B.false.the\_coin\_sensor.B.true, ..., \\
& delay...B.false..., reset\_the\_request\_timer, ..., \\
& output.the\_system\_mode.I.0.the\_coffee\_machine\_output.I.0, ..., \\
& input.the\_coffee\_request\_button.B.false.the\_coin\_sensor.B.false, ..., \\
& delay...B.false..., ..., \\
& output.the\_system\_mode.I.0.the\_coffee\_machine\_output.I.0, ..., \\
& input.the\_coffee\_request\_button.B.true.the\_coin\_sensor.B.false, ..., \\
& delay...eta2.B.true..., reset\_the\_request\_timer, ..., \\
& output.the\_system\_mode.I.3.the\_coffee\_machine\_output.I.0, ..., \\
& input.the\_coffee\_request\_button.B.false.the\_coin\_sensor.B.false, ..., \\
& delay...eta3.B.true, ..., \\
& output.the\_system\_mode.I.1.the\_coffee\_machine\_output.I.1, \\
& accept >
\end{aligned}$$

Basically, we can split the events into three distinct groups: input, output, and time-related events (delays and resets). From the first two, the tool infers the stimuli provided to the system, as well as the expected response. For example, the first input event represents the situation when the coffee request button is not pressed (*false*), and the coin sensor is true. Similarly, the first output event tells us that the system should react by going to the choice state (represented by 0), and the machine output is still equal to its initial value (assuming 1).

One concrete test case is obtained with the aid of Z3. From the reset and delay events we automatically generate a satisfiability problem. More specifically, there is a mapping from

[5] http://www.tcl.tk/

each *eta* variable that appears in the trace to a time constraint that needs to be fulfilled. Z3 is then used to find solutions (delays) that satisfy these constraints. More details are available in Section 4.4.1.

Figure 5.1, previously presented in Section 5.2, shows the screen where the user can select which requirements the test cases are going to cover, as well as inspect the generated test cases, which are presented in a tabular form. The concrete test case depicted in Figure 5.1, and reproduced in Table 5.1, is one possible concrete test case obtained from the previous trace. In this test case, a weak coffee is produced, since the user requested the coffee 2 seconds after inserting the coin. The coffee was delivered 10 seconds after the request. We note that in this test case the initial value of the machine output signal is *strong*.

Table 5.1: Example of test case for REQ004 (vending machine)

TIME	request	coin	mode	output
0.0	false	false	idle	strong
1.0	false	true	choice	strong
2.0	false	false	choice	strong
3.0	true	false	preparing weak coffee	strong
13.0	false	false	idle	weak

[Source: author]

In Table 5.1, *request*, *coin*, *mode*, and *output* stands for the system inputs (namely, *the_coffee_request_button*, *the_coin_sensor*) and the system outputs (namely, *the_system_mode*, *the_coffee_machine_output*), respectively. For legibility purposes, in the two rightmost columns, the numbers were replaced by their respective values according to the enumerations of each signal (see Figure 5.5): $0 \mapsto \textit{choice}, 1 \mapsto \textit{idle}, 3 \mapsto \textit{preparing weak coffee}$ for the system mode, and $0 \mapsto \textit{strong}, 1 \mapsto \textit{weak}$ for the coffee machine output.

5.2.6 SCR-Generator – Alternative I

To compare our proposal of using refinement checking to generate test cases with strategies employed by the industry, we consider two popular tools for that goal: T-VEC and RT-Tester. The first generates test cases from SCR specifications (as presented in the current section), whereas the second one is based on an intermediate representation called IMR (Section 5.2.7).

To allow a fairer comparison with our approach, we create an interface for using these tools within the context of natural-language requirements. Although it is possible and actually more promising to generate these notations from DFRS models, we derive SCR and IMR specifications directly from requirement frames, since the DFRS models were devised later, and it is not within the scope of this work to show how DFRSs models could be used instead.

Currently, SCR is being applied in several different control system industries (HEIT-

MEYER; BHARADWAJ, 2000), like Grumman, Bell Laboratories and Lockheed. Our work uses the T-VEC (BLACKBURN; BUSSER; FONTAINE, 1997) tool for automating the generation of tests from SCR specifications. It is a non-linear constraint solving theorem prover that helps ensuring requirements satisfiability, besides supporting testing. Moreover, T-VEC also allows the generation of test drivers in different programming languages, which assess the SUT with respect to the generated tests.

SCR employs monitored and controlled variables to define system requirements. It also has mode classes, used to model system states, and terms, internal variables declared for reuse purposes. SCR also allows the definition of a set of assumptions to impose constraints on the variables. It is also possible to define assertions describing properties, such as security and safety.

Functions define the behaviour of a system by specifying how changes of monitored variables, and even controlled ones, affect each controlled variable. Functions describe conditions (predicates) that, when valid, change a specific variable in a particular way. SCR allows two types of predicates: condition predicate, which is defined considering a single system state; and event predicate, which takes into account the changes that happen between two states.

An SCR event predicate has the form: $\dagger(c)$ *WHEN d*, where $\dagger$ stands for *@T, @F, @C*. A dashed ($'$) variable is the variable value in the new state, and an undashed variable stand for its value in the previous state, the meaning of these event predicates are: *@T(c) WHEN d* $\equiv \neg c \wedge c' \wedge d$, *@F(c) WHEN d* $\equiv c \wedge \neg c' \wedge d$, and *@C(c) WHEN d* $\equiv c \neq c' \wedge d$. If *WHEN d* is omitted, the meaning is defined considering only the previous and current value of c. To clarify the meaning of these event predicates, consider the following examples, and remember that a dashed variable stands for the value of this variable in the current state, whereas an undashed one refers to the value of this variable on the previous system state.

- @T(x < 3) WHEN y = 4 $\equiv$ NOT(x < 3) AND (x' < 3) AND (y = 4)
- @F(x < 30) $\equiv$ (x < 30) AND NOT(x' < 30)
- @C(x < 30) $\equiv$ (x < 30) $\neq$ (x' < 30)

From the first SCR proposal, some extensions were introduced. For instance, the support for the *WHERE* and *WHILE* semantics, in addition to *WHEN*: *WHERE* means that d is true only in the next state and *WHILE* means that d is true in the previous and next states (ATLEE; GANNON, 1993). The SCR we are generating follows the grammar presented in (LEONARD; HEITMEYER, 2003), but it also accepts this enhancement.

The SCR generation starts by determining the variables and their types. Variables are extracted from the PAT and CPT roles. Variables may be classified as controlled or as monitored. A variable v that is used in conditions is classified as monitored as it does not change. Its name starts with the prefix "m__" (e.g., *CPT*{the coin sensor} becomes the "m__the_coin_sensor" variable). Otherwise, if v appears as a patient, its value can change. Therefore it is classified as controlled, and receives a name starting with the prefix “c__”. Variables that appear both in conditions and in actions give rise to a monitored and a controlled variable. The former represents the system input, whereas the latter the system reaction.

To determine the variable types, we examine the contents of the thematic roles that are related to that specific variable occurrence. For example, if v appears as a CPT, we examine the contents of CFV and CTV. If v appears as a patient, we examine the values of TOV. This inference process is analagous to the one used when inferring variables of a DFRS model (see Section 3.2).

The next step consists of identifying SCR functions to describe how the controlled variables evolve according to modifications of other variables. A function is created for each controlled variable (*var*). It comprises one or more predicates obtained from the system requirements whose patient is the controlled variable (*var*).

As previously explained, there are two types of predicates in SCR: condition and event predicates. Predicates of the first type is generated when the requirement considers only the current value of variables (*var* is false). However, when the requirement mentions the current but also the previous value, event predicates are considered (*var* changes from false to true). For instance, the requirement REQ001 of the VM (reproduced below):

- REQ001 - *When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode.* [VM]

is translated to the following event predicate related to the variable *c__the_system_mode*:

(@T(m__the_coin_sensor = true)) AND (m__the_system_mode’ = 0) -> 1

Consider that, regarding the system mode, the value 0 stands for the idle state, and 1 for the choice state. The left-hand expression of -> is the guard, whereas the right-hand expression (*1*) is the expected system reaction when the guard is fulfilled (assigning 1 to the variable c__the_system_mode). We note that the previous predicate does not deal with the temporal aspect of the requirement (resting the request timer). We explain how time is modelled in SCR later.

When only condition or event predicates are associated with a controlled variable, the process of generating functions is straightforward. However, when a mixed scenario is the case, a special handling is required, since this is not allowed in SCR. In such a situation, two auxiliary variables are created: “c_event__” to group the event predicates, and “c_cond__” to group the condition predicates. Then, the controlled variable function is defined based on these two auxiliary variables. This way, the controlled variable changes triggered by any change of one of these variables using the $@C$ operator (see in Figure 5.9 the definition of the variable *c__the_system_mode*).

In some extensions of SCR there is a natural handling for time, namely via the operator *DUR*. However, the SCR version accepted by T-VEC does not implement this operator. Therefore, we need to create a special handling for time in our codification. Briefly, we create a new auxiliary monitored variable to denote the time, which is used when timed-based behaviour is necessary. In (CARVALHO et al., 2014c) we detail how SCR specifications are created from requirement frames, providing details for the aspects briefly mentioned here. The idea informally presented here is implemented in the SCR-Generator component. Figure 5.9 shows the SCR specification automatically generated for the VM example.

First, monitored and controlled variables are declared, besides auxiliary ones (*m__TIME* to represent the time, *c_cond__the_syste_mode* and *c_event__the_system_mode* to group the condition and event predicates related to the system mode). We note that outputs that are also inputs (the system mode) also give rise to monitored variables.

Afterwards, we show the definition of some functions. As previously explained, there are in SCR two types of predicates: condition (enclosed by *if-fi* blocks) and event (enclosed by *ev-ve* blocks) ones. The expression before the symbol -> represents the guard, whereas the one after is the expected system reaction (the value that is assigned to the controlled variable when the corresponding guard evaluates to true).

To generate test cases from the SCR specification, we use the T-VEC tool. T-VEC uses a tabular representation of SCR (BLACKBURN; BUSSER; FONTAINE, 1997), and thus the SCR specification is created using the T-VEC Tabular Modeler. Then, we use one component of T-VEC, the Vector Generation System (VGS), to automatically generate test cases from the SCR specification. When generating test cases, we can use the various configuration possibilities provided by the VGS component, such as generation guided by coverage criteria.

5.2.7 IMR-Generator – Alternative II

Here, system behaviour is internally modelled by state machines, captured in the IMR notation. The system model is arranged in hierarchical components $c \in C$, so that a partial function $p_C : C \nrightarrow C$ mapping each component but the root c_r to its parent is defined (PELESKA et al., 2011). Each component may declare variables, and hierarchic scope rules are applied in name resolution. Interfaces between Test Environment (TE) and SUT, as well as global model variables, are declared at the level of c_r. All variables are typed. When parsing the model the scope rules are applied to all expressions and unique variable symbol names are used from then on. Therefore we can assume that all variable names are unique and taken from a symbol set V with pairwise disjoint subsets $I, O, T \subset V$ denoting TE $\rightarrow$ SUT inputs, SUT $\rightarrow$ TE outputs and timers, respectively.

Each leaf component is associated with a state machine $s \in SM$, where SM denotes the set of all state machines which are part of the model. State machines are composed of *locations*

Figure 5.9: The vending machine specification – SCR

```
spec the_coffee_machine_system
 monitored variables
  m__the_system_mode : INTEGER;
  m__the_request_timer : FLOAT, initially 0;
  m__the_coin_sensor : BOOLEAN;
  m__the_coffee_request_button : BOOLEAN;
  m__TIME : FLOAT, initially 0;
 controlled variables
  c__the_coffee_machine_output : INTEGER;
  c__the_system_mode : INTEGER;
  c__the_request_timer : FLOAT;
  c_cond__the_system_mode : INTEGER;
  c_event__the_system_mode : INTEGER;
 function definitions
  var c__the_coffee_machine_output :=
   if
   [] ((m__TIME - m__the_request_timer) <= 30.0) AND
       ((m__TIME - m__the_request_timer) >= 10.0) AND (m__the_system_mode = 2) -> 0
   [] ((m__TIME - m__the_request_timer) <= 50.0) AND
       ((m__TIME - m__the_request_timer) >= 30.0) AND (m__the_system_mode = 3) -> 1
   fi
  var c__the_system_mode :=
   ev
   [] @C(c_cond__the_system_mode) -> c_cond__the_system_mode'
   [] @C(c_event__the_system_mode) -> c_event__the_system_mode'
   ve
  var c_cond__the_system_mode :=
   if
   [] ((m__TIME - m__the_request_timer) <= 30.0) AND
       ((m__TIME - m__the_request_timer) >= 10.0) AND (m__the_system_mode = 2) -> 0
   [] ((m__TIME - m__the_request_timer) <= 50.0) AND
       ((m__TIME - m__the_request_timer) >= 30.0) AND (m__the_system_mode = 3) -> 0
   fi
  var c_event__the_system_mode :=
   ev
   [] (@T(m__the_coin_sensor = true)) AND (m__the_system_mode' = 0) -> 1
   [] (@T(m__the_coffee_request_button = true)) AND
       ((m__TIME' - m__the_request_timer') <= 30.0) AND
       (m__the_coin_sensor = false) AND (m__the_coin_sensor' = false) AND
       (m__the_system_mode' = 1) -> 2
   [] (@T(m__the_coffee_request_button = true)) AND
       ((m__TIME' - m__the_request_timer') > 30.0) AND
       (m__the_coin_sensor = false) AND (m__the_coin_sensor' = false) AND
       (m__the_system_mode' = 1) -> 3
   ve
```

[Source: author]

(also called *control states*) $\ell \in L(s)$ and *transitions*

$$\tau = (\ell, g, \alpha, \ell') \in \Sigma(s) \subseteq L(s) \times G \times A \times L(s)$$

connecting source and target locations ℓ and ℓ', respectively. Transition component $g \in \mathrm{Bexpr}(V)$ denotes the guard condition of τ, which is a Boolean expression over symbols from V. For timer symbols $t \in T$ occurring in g we only allow boolean conditions $\mathrm{elapsed}(t, c)$ with constants c. Intuitively speaking, $\mathrm{elapsed}(t, c)$ evaluates to true if at least c time units have passed since t's most recent reset. Transition component $\alpha \in A = \mathbb{P}(V \times \mathrm{Expr}(V))$ denotes a set of value assignments to variables in V, according to expressions from $\mathrm{Expr}(V)$. A transition without assignments is associated with an empty set $\alpha = \varnothing$.

The work reported in (PELESKA et al., 2011) allows for the hierarchical decomposition of locations into *OR-states*, but for the purposes of our work flat machines are sufficient. Thus, every control state is also a *Basic Control State*, that is, a leaf in the hierarchy of control states. For a more comprehensive definition of the IMR notation, refer to (PELESKA et al., 2011).

Now, we present an overview of how an IMR can be extracted from requirement frames. Recall that the AGT role represents who performs the action. Thus, for each different AGT we create a new IMR component. Each system component comprises a state machine with a single location, and, based on the requirement frames, we infer self-transitions.

To illustrate this, we consider again the requirement REQ001 of the VM (reproduced below):

- REQ001 - *When the system mode is idle, and the coin sensor changes to true, the coffee machine system shall: reset the request timer, assign choice to the system mode.* [VM]

From its requirement frame, we extract the following transition: $\tau = (\ell_i, g_{i,j}, \alpha_{i,j}, \ell_i)$, where ℓ_i represents the location of the i_{th} state machine, which is the one corresponding to the *the coffee machine system*. The guard of this transition is $[mode = 1 \wedge sensor = true \wedge \_old\_sensor = false]$, where *mode* represents the system mode, *sensor* the current value of the coin sensor, and *_old_sensor* the last value of the coin sensor. The name *idle* is represented as 1, and *choice* as 0. Moreover, the actions associated with this transition are $\{(mode, 0), (request\_timer, 0)\}$, where *request_timer* represents the request timer.

To extract transitions like the one just presented, and to create an IMR from the requirement frames, we rely upon three main algorithms: identifying variables, identifying transitions, and creating IMRs. Hereafter, we present an intuition about these algorithms. A complete explanation can be found in (CARVALHO et al., 2014a).

In the IMR, variables are of three possible kinds: *input*, *output* and *global* variables. Besides that, we support the following data types: *integer*, *floating* point numbers, *boolean* and *clock* (timers). We note that there is a natural support for timers in this notation.

We consider inputs as variables provided to the SUT by the testing environment; their values cannot be modified by the system. Thus, a variable is classified as an input if, and only if, it appears only in conditions. All other variables, except the ones whose type is a clock, are classified as outputs. Clock variables (timers) are always classified as global variables. To distinguish between timers and other variables, we require the former to have the word "timer" as a suffix. Besides the input, output and clock variables, we create additional global variables do keep the old values of inputs and outputs. We know when old variables are required inspecting the content of the CAC role. If verbs such as *was*, *becomes*, among others are used, we know that the corresponding condition will refer to old value of variables. Old variables are prefixed by "*_old*". To infer the type of the variables we analyse the value associated with it in the requirement frame, which is the content of the roles: CTV, CFV, and TOV.

For each requirement frame, we also extract transitions: guard expressions and statements. The transitions are associated with their respective AGT representing a system component (one component per agent). The transitions are derived from the requirement conditions, whereas actions are the source for identifying assignments. Finally, we assemble the variables and transitions previously identified into one IMR top component, which comprises all system components identified after AGT roles.

As one might note, despite the particularities of each notation, the process described here and in Section 5.2.6 for deriving a formal model from requirement frames has many common points. This was our motivation for proposing the general model DFRS, from which different formal notations can be derived. This way, on which translation we only need to focus on the specific characteristics of each notation, since general aspects (such as identification of variables, and types) were already identified when generating the DFRS model.

For generating test cases with associated test data from IMR models, we rely on the RT-Tester tool (PELESKA et al., 2011). First, the model behaviour is formally encoded by means of a transition relation Φ. Following previous works (CLARKE; GRUMBERG; PELED, 1999), we describe transition relations relating pre- and post-states by means of first order predicates over unprimed and primed symbols from $BCS \cup V \cup \{\hat{t}\}$, where $BCS =_{\text{def}} \bigcup_{s \in SM} L(s)$ ("BCS" stands for "basic control states"). The unprimed symbols refer to the symbol value in the pre-state, and the primed symbols to post-state values. The variables with prefix "_old" are interpreted as unprimed symbols.

The transition relation distinguishes between *discrete transitions* Φ_D and *timed transitions* (also called *delay transitions*) Φ_T, allowing the model execution time $\hat{t}$ to advance and inputs to change, while the basic configuration, internal (excluding the special "old" variables) and output variables remain frozen. The delay transition is also responsible for updating the variables with prefix "_old". Thus, before changing the value of inputs, it copies the current value of each variable, which has an old version, to its old version. Discrete transitions take place whenever at least one state machine has an enabled transition, and they perform the assignments associated with the enabled transitions.

After encoding IMR models as transition relations, test cases are expressed as logical constraints identifying model computations which are suitable to investigate a given test objective. These constraints are *symbolic test cases* since at this stage no concrete test data to stimulate a model computation satisfying them exists. To solve these constraints we use the SMT solver of the RT-Tester. Thus, the solution can be seen as a test case composed by a sequence of test vectors where each test vector comprises the value of inputs and the system state with respect to a particular time moment.

5.3 Empirical evaluations

We evaluate the NAT2TEST strategy, in particular the specialisation that considers the process algebra CSP, considering examples from the literature, but also from the aerospace and the automotive industry (see Section 5.1). For each domain, we analyse two aspects: (i) the performance (Section 5.3.1) of each phase of the strategy (phase I – syntatic analysis; phase II – semantic analysis; phase III - DFRS generation; phase IV – CSP generation; and phase V – test generation), as well as the time required to execute the generated concrete test cases; and (ii) the ability to detect defects by means of mutation analysis. As a baseline we considered the generation and execution of random tests using Randoop (PACHECO et al., 2007).

It is important to bear in mind that threats to validity apply to this work. Despite that, the results allow interesting insights and provide some evidence about the feasibility of our proposal.

- Conclusion validity: design decisions considered when creating the reference Java programs might influence the results obtained, in particular, the ability of detecting errors via random testing. Moreover, the conclusions were not drawn with the aid of statistical analyses due to the few examples considered.
- Construct validity: our analysis regarding the ability to detect defects considered mutation analysis of Java programs, and random testing as baseline. Considering the nature of the domain of this work (timed reactive systems), it would be better to have considered embedded software instead of Java code.
- External validity: as we considered few examples, and they are of relatively small to medium size, we cannot statistically generalise the reached conclusions.

5.3.1 Performance analysis

In this section, all time measurements correspond to an environment configured by an i3 CPU with 2.27GHz equipped with 4 GB of RAM memory running the Ubuntu 14.04 LTS operating system. Table 5.2 presents the metrics related to our performance analysis considering the average time obtained from multiple runs (no outliers were eliminated).

Table 5.2: Performance metrics

NAT2TEST				
	VM	**NPP**	**PC**	**TIS**
# of requirements/words:	5/232	11/268	8/294	17/580
Time to process the reqs.:	0.11s	0.07s	0.14s	0.35s
# of cond./actions/TRs:	18/10/130	21/11/149	26/8/162	54/34/406
Time to identify the RFs:	0.10s	0.14s	0.10s	0.20s
# of I/O/T:	2/2/1	3/3/0	4/1/0	3/2/1
Time to generate the s-DFRS:	0.02s	0.01s	0.01s	0.01s
CSP specification (LOC):	134	135	112	189
Time to generate the CSP:	0.04s	0.02s	0.01s	0.01s
# of concrete test cases:	25	55	30	51
Time to create tests (FDR2/Z3):	94.87s/1.19s	39.23s/0.90s	4.52s/0.47s	778.28s/0.86s
Time to run the tests:	10.20s	2.91s	0.44s	12.44s
NAT2TEST$_{\text{CSP}}$ – Total Time:	**106.53s**	**43.28s**	**5.69s**	**792.16s**
Randoop				
# of test cases:	100	100	100	100
Time to generate the tests:	3.18s	1.52s	3.48s	4.50s
Time to run the tests:	25.10s	34.25s	22.16s	121.68s
Randoop – total time:	**28.28s**	**35.77s**	**25.64s**	**126.18s**

[Source: author]

The first three examples (VM, NPP, and PC) comprise simpler specifications, with a smaller lexicon (dictionary), besides less requirements and words, when compared with the TIS example. Within 0.11s, 0.07s, and 0.14s the respective requirements (5 for the VM, 11 for the NPP, and 8 for the PC) were parsed. Concerning the TIS (a larger specification with 682 words), 0.35s were necessary to analyze its 17 requirements. The actual number of words of the TIS is 682; however, using the aliases techniques described in Section 2.1.1.2, 102 were spared due to reuse.

After parsing the requirements, the obtained syntax trees are processed by the RF-Generator component. The contextual and inference rules presented in Section 2.2 were successfully used to infer the thematic roles from the requirements' syntax trees. By successfully we mean that the identified thematic roles were manually inspected and all of them corerespond to the expected result. This component identified 130, 149, 162 and 406 thematic roles (among 18/10, 21/11, 26/8, 54/34 conditions/actions, respectively) for each example, within 0.10s, 0.14s, 0.10s, and 0.20s, respectively.

From the requirement frames, we generate s-DFRS models. In these examples, it comprises systems with 2 (VM) to 4 (PC) inputs, and with 2 (VM, TIS) to 3 (NPP) outputs. Note that the examples NPP and PC do not have timers, that is, they do not have temporal-based behaviour. Therefore, no timers are required in these examples.

Afterwords, we encode the DFRS models as CSP processes. For the considered examples, the CSP models range from 112 Lines of Code (LOC) for the PC example to 189 LOC (TIS). The CSP specifications are generated in less than 0.05s.

Using FDR and Z3, we generate 5 concrete test cases for each requirement, except for the TIS system, in which case only three test cases were considered for each requirement. Here, we use the approach explained in Section 4.5 to select and generate test scenarios. Then, as explained in Section 5.2.5, we generate concrete test cases.

As one can clearly see in Table 5.2, the enumeration of test scenarios via FDR is the most time-consuming step. It is important to say that in these experiments we have not explored FDR compression and optimisation techniques yet.

Finally, the automatically generated test cases were used to find faults systematically created by code mutation in Java programs (see Section 5.3.2). Within 10.20s, 2.91s, 0.44s, and 12.44s the complete set of tests were ran against all mutants generated from the VM, NPP, PC and TIS programs, respectively. Therefore, the total amount of time required to perform the entire NAT2TEST$_{\text{CSP}}$ strategy, besides running the generated test cases, was about 13 minutes for a medium size example (TIS), whereas only 5.69s were necessary for the simplest example (PC).

To provide a baseline, we used Randoop (PACHECO et al., 2007) to randomly generate test cases from the same Java programs created for each analysed example. In these experiments we needed to consider a relatively small set of random test cases. The reason is not the time required to generate these tests (Table 5.2 shows that the tests were generated within 5s), but the amount of memory needed to run these random tests. With more test cases, we got a Java *OutOfMemory* exception concerning the Java *PermGen* space.

Each random test created by Randoop instantiates a Java object. With this object, Randoop performs several random manipulations, and finally asserts its state to check whether it is equal to the expected one. Hence, a huge number of objects are created within a short amount of time, and the Java garbage collector does not handle them as fast. To give some figures, concerning the TIS example, assuming an average of 10 objects being created per test, a set of 100 test cases and 1,000 mutants, one might have about 1,000,000 objects being created within a time frame that would not be enough for proper garbage collection.

5.3.2 Mutant-based strength analysis

The use of mutation operators yields a statistically trustworthy comparison of test cases strength because they create erroneous programs in a controlled and systematic way (ANDREWS; BRIAND; LABICHE, 2005). Thus, a good test case should be able to detect the introduced error (i.e., kill the mutant). Sometimes the alive mutant is equivalent to the correct program. However, in general, such a verification is undecidable and too error-prone to be made manually.

In this work we follow a conservative approach in which we assume that all alive mutants are considered different. This assumption makes the results of the empirical analysis the worst case. In other words, if some alive mutants are semantically equivalent to the original program, what might happen, the resulting mutation score would be higher as these mutants would not be considered. Therefore, as previously said, the results presented here consist of a worst-case analysis.

To perform the mutant-based strength analysis reported in this section, we created reference (concerning at least the automatically generated test cases by our strategy and by Randoop) Java implementations for the four examples considered in this work. The Java code was created according to the following approach: we created a single method whose body consists of if-else clauses written after the system requirements. The return of this method is a string with the expected state of the system after executing this method for the given inputs. To avoid a non-deterministic test result due to the unpredictability of time, we modelled the system global time as a system variable. Hence, the test procedure is able to control the system time evolving. Despite that, we still have the possibility of time-related faults. For instance, the mutation process can lead to situations where a timer variable is reset in an inappropriate moment or the system fails to read the correct value of the global time.

We generated mutants systematically and deterministically using the μJava tool (MA; OFFUTT; KWON, 2005). We considered all method-level mutant operators (a total of 15 operators) provided by μJava. Given a base class, the μJava tool automatically generates new classes applying its mutant operators. For example, the statement `x++` can be changed to `++x`,

Table 5.3: Metrics concerning mutant-based strength analysis

	VM	**NPP**	**PC**	**TIS**
Java (LOC):	48	66	46	94
Mutants:	238	319	135	1,127
		NAT2TEST$_{\text{CSP}}$		
Killed:	174	247	105	362
Mutation Score:	**73.11%**	**77.43%**	**77.78%**	**32.12%**
		Randoop		
Killed:	67	85	113	139
Mutation Score:	**28.15%**	**26.65%**	**83.70%**	**12.33%**

[Source: author]

`x--`, and `--x`, and thus the tool generates three new versions of the base class.

After generating mutants, we ran the generated test cases against each mutant to compute the mutation score, and thus evaluate the test suite (composed by all test cases) strength. Concerning our approach, we needed to manually instrument the Java code to link the test inputs/outputs with the Java code variables. Table 5.3 summarizes the collected metrics.

As it can be seen in Table 5.3, fewer mutants were generated for the first three Java programs (238, 319 and 135, respectively) than the last one (1,127). The best mutation score was obtained for the Embraer example (PC – 77.78%). Contrarily, we had the worst score with the TIS example (32.12%). The Randoop tests killed less mutants for all examples but the PC.

The PC example is a typical scenario where random testing can yield good results: the code comprises only a small set of boolean variables (5 variables to be precise – 4 inputs and 1 output), besides not considering time-dependent behaviour. Therefore, it is not hard to enumerate all possibilities (32). However, when we are dealing with more complex examples (more variables, besides time-dependent behaviour), Table 5.3 shows that our approach outperforms random testing (particularly using Randoop), that is, it was able to unveil more faults than a random test generation strategy and within considerably less amount of time.

5.3.3 Practical validation of DFRS models

Besides analysing performance and mutant-based strength analysis, we provide an empirical argument to whether the DFRS models are expressive enough to represent the behaviour of a timed reactive system as defined using natural language. It is an important evaluation, since DFRS models are a central element of our strategy.

Considering the same four examples (see Section 5.1), and supported by the mechanisation of the strategy particularly presented in Section 5.2.3 (the DFRS animator), we assess whether test cases, either independently written by domain specialists from industry or generated by a commercial tool (RT-Tester) from the same set of requirements, are compatible with the corresponding DFRS models. By being compatible, we consider that there is a sequence of delay and function transitions of the e-DFRS, which is obtained from the s-DFRS generated from the NL requirements, that illustrate the delays, the system inputs and the expected outputs described in the test case.

To analyse whether the test cases are compatible with the corresponding DFRS models, we have developed a depth-first search algorithm that explores the state space of an e-DFRS guided by a test case. We provide to the model the inputs described by each test vector, and check whether the outputs provided by the system are equal to those in the vector. This comparison is straightforward (that is, the test oracle is trivial) since we are dealing with primitive types.

The verdict of this compatibility analysis has been successful, since all test cases are compatible with the corresponding DFRS models and, thus, it gives some evidence that the DFRS models for the four examples did not fail to capture the underlying semantics of the natural-language requirements. The time required to performe this analysis ranged from 9ms (VM) to 192ms (TIS).

5.4 Concluding Remarks

This chapter presented the tool developed to support the NAT2TEST strategy. The NAT2TEST tool is written in Java (it is multi-platform), and its GUI is built using the Eclipse RCP framework. Each phase of the strategy is implemented by one different component to promote reuse. For instance, one might reuse the CNL-Parser, the RF-Generator, and the DFRS-generator to obtain DFRS models from natural-language requirements and, then, derive formal specifications, considering notations not targeted here, such as Coloured Petri Nets (CPNs) (JENSEN, 1996), via the implementation of a new generator (e.g., CPN-Generator). We have actually explored this particular path, as further discussed in Chapter 7.

Here, we also evaluated the NAT2TEST strategy, in particular the specialisation that considers the process algebra CSP – NAT2TEST$_{\text{CSP}}$, considering examples from the literature, but also from the aerospace and the automotive industry. For each example, we analysed two aspects: (i) performance, and (ii) the ability to detect defects by means of mutation analysis. As a baseline, we considered random testing (Randoop). In general, our strategy was able to unveil more faults than a purely random test generation strategy and within considerably less amount of time.

To compare our proposal of using refinement checking to generate test cases with strategies employed by the industry, we considered two alternatives for the NAT2TEST$_{\text{CSP}}$ strategy (NAT2TEST$_{\text{SCR}}$ and NAT2TEST$_{\text{IMR}}$), which are based on the notations SCR and IMR. In such cases, the test generation is performed with the aid of popular tools for this goal: T-VEC and RT-Tester, respectively. The achieved results are fully presented in (CARVALHO et al., 2014c) and (CARVALHO et al., 2014a).

It is important to clarify that the results achieved using T-VEC and RT-Tester shall not be used to say that one or other is better or worse than the generation of test cases via refinement

Table 5.4: Mutation score with respect to the NAT2TEST specialisations

	VM	**NPP**	**PC**	**TIS**
NAT2TEST$_{\mathbf{CSP}}$:	73.11%	77.43%	77.78%	32.12%
NAT2TEST$_{\mathbf{SCR}}$:	57.74%	48.28%	95.52%	37.48%
NAT2TEST$_{\mathbf{IMR}}$:	54.67%	69.04%	87.50%	98.05%

[Source: author]

checking (in particular, FDR). Such a comparison would be biased due to the fact that the focus of this research, since from its beginning, was to explore the use of refinement checking to generate test cases and, thus, we might have not employed the same investigative effort on the NAT2TEST specialisations that use SCR and IMR.

Nevertheless, it is an important exercise to apply part of our strategy, with the aid of commercial testing tools, to analyse whether the generation of test cases from natural-language requirements can lead to relevant results. With respect to the mutant-based strength analysis criterion, Table 5.4 summarises the achieved figures by these three specialisations of the NAT2TEST strategy.

As one can see, the CSP-based approach achieved higher mutation scores for the first two examples (VM and NPP). With respect to the VM example, the highest score was achieved with the aid of T-VEC, whereas for the TIS example, with the aid of the RT-Tester. Regarding the VM, we justify the higher results of T-VEC and RT-Tester to the adoption of better coverage criteria. Using FDR, we generated a fixed number of test cases for each requirement. However, T-VEC and RT-Tester offer more efficient coverage criteria such as Modified Condition/Decision Coverage (MC/DC) (HAYHURST et al., 2001). Therefore, we believe that if we considered better coverage criteria for selecting test scenarios, we would be able to unveil more errors and, thus, achieve higher mutation scores, without generating more test scenarios.

The high mutation score achieved with the aid of the RT-Tester with respect to the TIS example is quite intriguing, since the tool achieved significantly lower scores in less complex examples, such as the VM and the NPP, and no specific configuration was used when generating test cases for the TIS. Our main hypothesis for this result is that the generated test cases for the TIS are considerably longer (more steps) than the ones generated for the VM and NPP. Therefore, we believe that the longer test cases were able to reach system states that were not reached by the shortest test cases of the VM and the NPP. An interesting investigation, not performed since it is outside the scope of this research, is trying to force the RT-Tester to generate longer test cases for the VM and the NPP to compare the obtained results.

Regarding performance, as expected, the NAT2TEST$_{\text{CSP}}$ variant, mainly with respect to the TIS example, was considerably slower (792.12s) than the other two variants (NAT2TEST$_{\text{SCR}}$: 93.87s; NAT2TEST$_{\text{IMR}}$: 87.61s). We say that this was expected as these tools (T-VEC and RT-Tester) were designed and optimised for the particular goal of generating test cases, whereas FDR is a general-purpose model checker. Moreover, as previously said, we have not explored

FDR compression and optimisation techniques. This might have led to better performance.

Finally, other performed empirical evaluation concerned the expressiveness of DFRS models. We assessed whether test cases, either independently written by domain specialists from industry or generated by a commercial tool (RT-Tester) from the same set of requirements, are compatible with the corresponding DFRS models. The obtained results showed that all considered test cases are indeed compatible.

6 Related Work

In this chapter, we discuss the related work with respect to the two main scientific contributions of this research: formal modelling of timed-systems from Natural Language (NL) requirements (Section 6.1), and formal testing theories based on timed input-output conformance relations (Section 6.2).

6.1 Modelling timed-systems from NL requirements

Formal modelling natural languages is not a new-term research topic. Here, we analyse works from six distinct perspectives: (1) domain: whether the modelling approach is tailored for a specific domain; (2) input: how the system requirements are documented; (3) model: the underlying formal notation used to represent the system behaviour; (4) data: whether this notation can explicitly deal with variables; (5) time: whether this notation can explicitly deal with temporal behaviour; and (6) requirement analyses: which properties of the requirements can be analysed via this notation. Table 6.1 summarises our analyses of related work considering these six perspectives.

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

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

Table 6.1: Related work – modelling timed-systems from NL requirements

	Domain	**Input**	**Model**	**Data**	**Time**	**Analyses**
LEE; CHA; KWON	General	UCs	CMPN	No	No	Consistency Completeness
LIU et al.	General	UCs	Act. diagr.	No	No	Consistency Integrity
SCHWITTER	General	NL reqs.	FOL	No	No	Not reported
ACEITUNA; DO; SRINIVASAN	General	NL reqs.	CCM	Yes	No	Off-nominal
BACKES et al.	Emb.	NL reqs.	AADL	Yes	No	Realisability
BODDU et al.	General	NL reqs.	FMONA	Yes	No	Consistency
NOGUEIRA; SAMPAIO; MOTA	Mobile	UCs	CSP	Yes	No	Not reported
ESSER; STRUSS	General	NL reqs.	FRL	No	Yes	Not reported
SIEGL; HIELSCHER; GERMAN	General	NL reqs.	TUM	No	Yes	Consistency Completeness
AMBRIOLA; GERVASI	General	NL reqs.	CNM	Yes	Yes	Consistency Completeness Ambiguity
ILIC	General	NL reqs.	B method	Yes	Yes	Consistency
LEVESON et al.	Emb.	NL reqs.	RSML	Yes	Yes	Consistency Completeness
MILLER et al.	Emb.	NL reqs.	RSML^{-e}	Yes	Yes	Consistency Completeness Reachability
SCHNELTE	Auto.	NL reqs.	TQE	Yes	Yes	Reachability
SANTIAGO JUNIOR; VIJAYKUMAR	General	NL reqs.	Statecharts	Yes	Yes	Not reported
NAT2TEST	Emb.	NL reqs.	DFRS	Yes	Yes	Consistency Completeness Reachability Time lock

[Source: author]

time modelling, since only the sequencing of events is considered.

Considering these remarks, we group similar works into three distinct categories (see Table 6.1). While the first group comprises techniques that do not support data and time information on requirements, the second one supports at least one of these two concepts. The last group, to which our approach belongs, supports both of them. In what follows, we summarise the previous studies, while comparing them with our own work.

While some approaches are tailored for Use Cases (UCs) described in NL (LEE; CHA; KWON, 1998; LIU et al., 2014; NOGUEIRA; SAMPAIO; MOTA, 2014), processing NL requirements is more common. In (LEE; CHA; KWON, 1998), a variant of Petri Nets (Constraints-based Modular Petri Nets – CMPNs) is proposed for modelling use cases. To generate the corresponding CMPN model, one needs to fill an action-condition table manually, besides clarifying event names, which represent the actions described in the use cases. There is no support for data and time. On the other hand, using the CMPN model, it is possible to perform consistency and completeness analyses automatically.

In (LIU et al., 2014) and (NOGUEIRA; SAMPAIO; MOTA, 2014), use cases are used as source for the generation of formal models: the former uses a restricted and formal version of activity diagrams, and the latter CSP. The CNL considered in (NOGUEIRA; SAMPAIO; MOTA, 2014) is tailored for mobile applications, whereas the strategy described in (LIU et al., 2014) is for general purpose. Data is considered in (NOGUEIRA; SAMPAIO; MOTA, 2014) via annotations in the use cases. Only events are considered in (LIU et al., 2014), where it is also shown how the derived activity diagrams can be used to verify the consistency and integrity of requirements.

A previous work (SCHWITTER, 2002) propose a computer-processable CNL for writing unambiguous and precise requirements: PENG. The specification written in PENG can be deterministically translated into first-order predicate logic (FOL). Data and time aspects are not considered, nor is the analysis of properties of the requirements.

In (ACEITUNA; DO; SRINIVASAN, 2014), a Casual Component Model (CCM) is used to model the behaviour described by NL requirements. This formal model needs to be manually created from the specification. In CCM, the states can be used to model valuations of a variable (e.g., s1 – switch(off), s2 – switch(on)), from which a NuSMV specification (CIMATTI et al., 1999) is automatically derived. Then, temporal logic can be used to seek off-nominal (undesired) behaviour.

The approach of (BACKES et al., 2015) also considers variables, and it is tailored for embedded systems. It uses as internal notation AADL (Architecture Analysis and Design Language), where assume-guarantee contracts are manually created. In this work, it is possible to assess whether these contracts and the corresponding requirements are realisable. Differently from other approaches, the authors show how to perform this analysis in a compositional way.

The work reported in (BODDU et al., 2004) presents a requirements analysis tool called RETNA. This tool accepts NL requirements and, with user interaction, it translates the requirements into a logical notation: FMONA, which is a high-level language for describing weak monadic second-order logic. This model can then be used to analyse whether the NL requirements are consistent.

In (ESSER; STRUSS, 2007) requirements are written in a limited standardized format. The requirements need to be written according to a strict if-then sentence template, which, however, can be used to represent time properties. Despite describing how to translate these templates to the Formal Requirement Language (FRL), the work does not elaborate on how this model could be used to check properties of the requirements. In (SIEGL; HIELSCHER; GERMAN, 2010) it is also possible to represent timed-behaviour using Timed Usage Models (TUM), which are Markov Chain Usage Models (MCUM) with time information. This model is manually created from the system requirements. Consistency and completeness properties can be verified automatically. Differently from other approaches, this model can also take into account probabilistic properties.

The works analysed so far do not take into account both data and time aspects and, thus, differ from our approach: the DFRS models consider both of them. The approaches proposed in (AMBRIOLA; GERVASI, 2006; ILIC, 2007; LEVESON et al., 1994; MILLER et al., 2006; SCHNELTE, 2009; SANTIAGO JUNIOR; VIJAYKUMAR, 2012) are closer related to our work, since they share the fact that data and time are both supported.

The CIRCE environment, which enables analysis of natural-language requirements, is presented in (AMBRIOLA; GERVASI, 2006). In this environment, requirements are interpreted according to the CIRCE Native Meta-Model (CNM). Besides analysing properties of requirements, this environment allows the generation of UML models and code from the CNM notation. Differently from our strategy, it requires manual effort when modelling the system requirements. Here, one needs to create by hand designations and definitions. The former establishes equivalences between different terms that refer to the same entity, and the latter establishes notations for expressing requirements in a succinct way. While these two elements have a well-defined and formal structure, the requirements description statements are expected to be free-form text.

In (ILIC, 2007), the B method (ABRIAL, 1996) is used to construct formal models for system requirements. In this work, the requirements need to be translated to predefined templates for describing events, data and time, from which B specifications are systematically translated. Each template defines the information that is mandatory. Comparing to our approach, the thematic roles are the counterpart of these templates. As just said, in (ILIC, 2007), one needs to classify manually the requirements according to these templates, besides filling them. Differently, thematic roles are automatically inferred from the requirements in the NAT2TEST strategy.

In (LEVESON et al., 1994; MILLER et al., 2006), the Requirements State Machine Language (RSML) is used as formal model for NL requirements. A restricted version of this notation (RSML^{-e}) is adopted in (MILLER et al., 2006), where events are not allowed. This notation, which is argued to be tailored for embedded systems, is used to document the system requirement. In (LEVESON et al., 1994), this internal model is analysed by proposed algorithms to check whether the requirements are consistent and complete. In (MILLER et al., 2006), these analyses are automated with the aid of NuSMV and PVS (OWRE; RUSHBY; SHANKAR, 1992). These two studies require user intervention to classify and edit requirements. This is not a necessity within the NAT2TEST strategy.

In (SCHNELTE, 2009), assuming that the system specification is manually represented conforming to a set of templates, developed for automotive systems, a Temporal Qualified Expression (TQE) is derived. It is also necessary to identify manually signal names along with its possible values. Differently, in our approach, the signal names, their types, and possible values are automatically inferred.

In (SANTIAGO JUNIOR; VIJAYKUMAR, 2012), the SOLIMVA methodology is presented. The methodology has tool support to translate automatically NL requirements into statechart models. Another tool (GTSC) is used to generate test cases. In this work, besides writing the requirements, one needs to identify and partition inputs and outputs. This is not required in our approach. Moreover, this work does not explain how the statechart models can be used to analyse requirements, but considers this as a possible line for future work.

In summary, the NAT2TEST strategy formally describes system requirements using DFRS models, which are explained in details in Chapter 3. Similarly to other approaches previously identified, the formal model used to represent the system behaviour considers both data and time information. It can also be used to check system properties such as consistency, completeness, reachability, and absence of time lock. Our work stands out from the similar approaches reported here by the richness of the model generated solely from NL requirements without user intervention.

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

6.2 Timed input-output conformance relations

We compare our CSP timed input-output conformance relation (csptio) with the conformance relations listed below. As some of these relations have a similar name (*tioco*) we added subscripts to differentiate each of them.

- ioco_{DTA} (KHOUMSI; JÉRON; MARCHAND, 2003);

Table 6.2: Related work – timed input-output conformance relations

Relation	Tool	Notation	Data	Delay	Quiescience	Partial	Test
ioco_{DTA}	No	DTA	No	Arb.	Yes	N/M	Yes
tioco_{TTG}	TTG	TIOTS	No	Obs.	N/M	Yes	Yes
rtioco	T-UPPAAL	TIOTS	No	Obs.	Yes	No	Yes
tioco_{M}	N/M	TIOTS	No	Arb.	Yes	N/M	Yes
tioco_{TorX}	TorX	TIOTS	No	Arb.	Yes	N/M	Yes
tioco_{Sch}	N/M	TIOTS	No	Obs.	N/M	Yes	N/M
tioco_{M}^{R}	N/M	TIOTS	No	Arb.	Yes	Yes	N/M
tioco_{ζ}	N/M	TIOTS	No	Unb.	Yes	Yes	N/M
tioco_{And}	WIP	TIOSTS	Yes	N/M	N/M	N/M	Yes
csptio	FDR/Z3	CSP	Yes	Arb.	Yes	Yes	Yes

[Source: author]

- tioco_{TTG} (KRICHEN; TRIPAKIS, 2004);
- rtioco (LARSEN; MIKUCIONIS; NIELSEN, 2004);
- tioco_{M} (BRIONES; BRINKSMA, 2005);
- tioco_{TorX} (BOHNENKAMP; BELINFANTE, 2005);
- tioco_{Sch} (SCHMALTZ; TRETMANS, 2008);
- tioco_{M}^{R} (SCHMALTZ; TRETMANS, 2008);
- tioco_{ζ} (SCHMALTZ; TRETMANS, 2008);
- tioco_{And} (ANDRADE et al., 2011).

In our comparison, we considered the following aspects: (i) tool support, (ii) notation to model the specification, (iii) support for data communication, (iv) assumption concerning observability of delays, (v) quiescence definition, (vi) support for partial specifications, (vii) definition of a test generation strategy. Table 6.2 summarises the comparison (N/M means “not mentioned”, and WIP means “work in progress”).

Tool support. Most timed conformance relations do not have yet (or do not mention) tool to support its verification. The relation tioco_{TTG} is supported by the prototype tool TTG. This tool is built on top of the IF environment (BOZGA et al., 2000). This tool was applied to analyse a small case study and the K9 Martian Rover executive of NASA (BRAT et al., 2004). The relation rtioco is supported by the tool T-UPPAAL, which is built on top of the UPPAAL on-the-fly model checking tool engine. This tool was applied on an industrial case study provided by Danfoss Refrigeration Controls Divison that consists of an Electronic Cooling Controller (EKC) for industrial cooling plants.

The relation tioco_{TorX} is verified by a timed extension of the TorX, which is an on-the-fly testing tool that tests for ioco (TRETMANS, 1999) conformance. In this case, the work does not report a more detailed use of the tool. Finally, the tioco$_{And}$ mentions a tool that is currently being developed, and it reports an application with respect to a toy example. The verification of csptio is provided by means of FDR and Z3, as explained in Section 4.4.2.

Therefore, as it can be seen in Table 6.2, some timed input-output conformance relations are only theoretically defined; its mechanisation is not necessarily defined. Differently, as explained in Section 4.4.2, we propose a mechanisation for verification of csptio-conformance. Furthermore, we prove that our mechanisation is sound with respect to csptio definition.

Modelling notation. Almost all relations model the specification as a TIOTS. The exceptions are ioco$_{DTA}$ and tioco$_{And}$. The former uses a class of Determinizable Timed Automata (DTA) to model the specification, whereas the latter considers a symbolic TIOTS (TIOSTS).

Differently from these relations, csptio models the specification as a CSP process, and thus does not need to handle operational models like transition systems. It is mechanised in terms of a high-level strategy by reusing successful techniques and tools: refinement checking (FDR) and SMT solving (Z3). Besides that, the use of a process algebraic approach is more modular. An evidence of the modularity of a process algebraic approach is shown in (NOGUEIRA; SAMPAIO; MOTA, 2014): it is shown how to deal with control aspects and then how to consider data as a conservative extension. Handling time as we do here is also a conservative extension of the theory presented in (NOGUEIRA; SAMPAIO; MOTA, 2014). Particularly, Theorem 4.4.1 is an orthogonal extension of Theorem 5.1 (from (NOGUEIRA; SAMPAIO; MOTA, 2014)), which allowed us to lift both the refinement assertion and the proof of the untimed conformance verification. Furthermore, our algebraic and symbolic approach seems more suitable to explore compositional properties, which we plan to develop as future work.

Data support. The majority of the relations do not support data communication. The only exception is tioco$_{And}$, which symbolically deals with data aspects. Similarly, csptio also supports data. A worth mentioning difference between these two relations is that just time aspects are symbolically modelled in csptio, whereas tioco$_{And}$ deals with time and data symbolically. Therefore, the relation tioco$_{And}$ might have a better scalability than csptio since the operational model constructed from the TIOSTS tends to be smaller than the LTS obtained from CSP specifications of DFRS models.

Type of delay. Concerning the type of delay, we can identify two types of conformance relations. The first one comprises relations that assume that all delays are observable (referred to as "*Obs.*" in Table 6.2). This assumption is not too realistic as in practice a timeout needs to be defined when analysing the behaviour of a timed system, and thus only delays up to this timeout can be observable.

Concerning the second type, only delays up to some bound are observable. This is a more realistic assumption. Concerning this second type, we have two inner divisions: the relation tioco$_\zeta$ considers an unbounded delay (referred to as "*Unb.*" in Table 6.2), whereas others define an arbitrary upper bound (referred to as "*Arb.*" in Table 6.2). The first inner division is more general since the results of conformance verification is not limited to an upper bound, whereas in the second case it is said that an implementation is correct with respect to its specification only up to the defined upper bound. However, as previously said, in practice a timeout needs to be defined. The relation tioco$_{And}$ does not mention what is assumed with respect to the type of delays. Regarding verification of csptio in practice, as discussed in Section 4.4.2, our conformance relation fits the *arbitrary* upper bound category.

Quiescence definition. An important aspect of a conformance relation is whether it considers the notion of quiescence: an absence of output, which is considered to be observable. The notion of quiescence provides additional information about the system behaviour, and thus allows to distinguish faulty behaviours that is not related to the generation of wrong outputs. Concerning the analysed relations, most of them support quiescence. Regarding csptio, as discussed in Section 4.4, quiescence is an element considered by our relation. However, due to the type of systems dealt with by us (a class of embedded systems whose inputs and outputs are always available as signals), it is not necessary to annotate quiescence in the model, what is a standard practice in other relations.

Partial specifications. It is commonly useful to specify the system behaviour incrementally. To consider this characteristic, the conformance relation cannot be defined as a trace inclusion relation. Part of the analysed relations support this concept, whereas others do not mention clearly whether they consider partial specifications in their conformance relations. The relation rtioco is the only one that explicitly states that partial specifications are not allowed. As discussed in Section 4.4, we also allow partial specification. However, also due to the type of systems dealt with by us, we consider a different notion of partial specifications when compared to other conformance relations: the implementation shall consider the same set of input and output signals, but new value for them might be considered.

Test strategy. Most works define a sound testing strategy based on the proposed conformance relations. The exceptions are the relations tioco$_{Sch}$, tioco^R_M, and tioco$_\zeta$, which focus on the definition of a conformance relation. These works mention that the relations can form the basis for a theory of test generation, but they do not say whether it is a work in progress or a planned one. Concerning csptio, as presented in Section 4.5, we also consider a testing theory based on our conformance relation.

We believe that our timed input-output conformance relation distinguishes itself from other relations defined in the literature by the simultaneous support for reasoning about time and data aspects via a notation such as CSP. As previously discussed, csptio verification does not need to handle operational models like transition systems, and as an algebraic and symbolic approach, it seems more suitable to explore compositional properties, which can be a fundamental characteristic when analysing complex and large systems.

6.3 Concluding remarks

This chapter compared the two main scientific contributions of this research (DFRS models, and formal testing theory based on csptio) with respect to the state-of-the-art. Concerning the first contribution, we compared it with others according to aspects such as application domain, manipulation of variables, time support, and supported requirement analyses. Regarding the second contribution, tool support, considered notation, and data support were aspects taken into account by the analyses.

As discussed throughout this chapter, we believe that our contributions address interesting and challenging problems and, thus, they extend the state-of-the-art regarding formal modelling of timed systems from natural-language requirements, besides the research area of timed input-output conformance relations. The DFRS model is a symbolic, timed and state-rich automata-based notation for representation of natural-language requirements, which is derived automatically from requirements that specify in a high-level fashion how states can be reached via transitions. Finally, to the extent of our knowledge, our timed input-output conformance relation is the first conformance relation that simultaneously support symbolic time and data aspects using a process algebra.

7 Conclusions

This work presented a model-based testing strategy for generating test cases from natural-language requirements: NAT2TEST. Actually, it is a general framework that can be specialised via the adoption of different formal models and test generation tools. Nevertheless, we focus on a formal approach based on the CSP process algebra: NAT2TEST$_{\mathrm{CSP}}$.

We dispense the need to know the syntax and semantics of the underlying notations, besides allowing early use of MBT, by means of NLP. In this way, the formal and semi-formal models internally used by the NAT2TEST strategy are automatically generated from the natural language requirements.

The NAT2TEST$_{\mathrm{CSP}}$ strategy comprises five phases. The first phase verifies whether the system requirements are in accordance with the SysReq-CNL grammar, a CNL specially designed for editing requirements of data-flow reactive systems. For each valid input requirement, its corresponding syntax tree is identified. In the second phase (semantic analysis), the requirements are semantically analysed using the case grammar theory. In this theory, a sentence is not analysed in terms of the syntactic categories or grammatical functions, but in terms of the semantic (thematic) roles played by each word/group of words in the sentence. Therefore, for each syntax tree the group of words that correspond to a thematic role is identified. The collection of thematic roles for a requirement is called the requirement frame.

Afterwards, the third phase (DFRS generation) derives DFRS models — an intermediate formal characterisation of the system behaviour from which other formal notations can be derived — such as SCR, IMR, and CSP. The possibility of exploring different formal notations allows analyses from several perspectives, using different languages and tools. Besides that, it makes our strategy extensible.

The fourth phase consists of using the process algebra CSP as a target to DFRS models. Finally, in the fifth and last phase, we select and generate test scenarios, from which test cases are built with the aid of FDR and Z3. Here, we prove that test cases built using our CSP testing infrastructure are sound with respect to csptio: a CSP timed input-output conformance relation. By soundness, we mean that if a test execution leads to a fail verdict, it follows that the implementation does not conform to the specification according to csptio.

The NAT2TEST strategy is automated by the NAT2TEST tool, which is written in Java, and its GUI is built using the Eclipse RCP framework. The tool can be easily installed and it runs on multiple platforms.

The proposed strategy is evaluated considering examples from the literature, but also from the aerospace and the automotive industry. For each example, we analyse two aspects: (i) performance, and (ii) the ability to detect defects by means of mutation analysis. As a baseline we consider the generation and execution of random tests using Randoop. In general, our strategy outperformed the considered baseline: random testing.

Therefore, our contributions include theoretical results, tool implementation, as well as empirical evaluations. In this light, we might say that this research successfully achieved its goal providing an answer to its main research question: how to automatically and formally generate test cases from natural-language requirements, and, in particular, regarding timed reactive systems. In what follows, considering the two main scientific and challenging achievements of this work, we situate our contributions with respect to the state-of-the-art.

A formal model of timed-systems from NL requirements. The DFRS model provides a formal semantics for the system requirements. It is able to represent the system behaviour considering both data and time information. Actually, there are two representations of DFRSs: a symbolic (s-DFRS) and an expanded one (e-DFRS). The former, which is automatically derived from requirement frames, inherently avoids an explicit representation of possibly infinite sets of states and, thus, the state space explosion problem. The latter is built dynamically from its symbolic counterpart, possibly limited to some bound, and then used to bounded analyses such as consistency, completeness, reachability, and absence of time lock. Therefore, our work stands out from the similar approaches reported here by the richness of the model generated solely from NL requirements without user intervention.

A CSP timed input-output conformance relation and a strategy for sound test case generation. A CSP-TIO process is a tuple composed by a CSP process obtained from DFRS models, along with its distinct input, output, time, reset and auxiliary alphabets. Considering CSP-TIO processes, we propose a systematic mechanism for selecting and generating test scenarios. These test scenarios are used to generate test cases within our formal testing theory, which is based on a CSP timed input-output conformance relation (csptio). We prove that there is a sound mechanisation of conformance verification based on csptio, with the aid of a refinement checker (FDR) and a constraint solver (Z3). Furthermore, we prove that our process for creating CSP test cases from test scenarios is sound with respect to csptio. Our formal testing theory distinguishes itself from others by the simultaneous support for reasoning about time and data aspects via a notation such as CSP, which might be more suitable to explore compositional properties than approaches that need to handle operational models like transition systems.

7.1 Future work

As future work, we envisage the following tasks.

Reduce assumptions concerning CSP encoding of DFRSs. As explained in Section 4.2.6, our representation of DFRS models as CSP processes has some assumptions. If they are not respected, the CSP representation is not accurate. These assumptions are restrictive, since some valid DFRSs cannot be used to generate test cases via CSP. Therefore, a relevant future work consists of reflecting upon these restrictions to find ways of generating the CSP representation with less assumptions.

Propose other coverage criteria. Commercial testing tools such as T-VEC and RT-Tester support more possibilities of coverage criteria (e.g., MC/DC) than our testing strategy (requirement coverage). As discussed in Section 5.4, this can lead to the generation of more promising test suites, since more aspects of the system behaviour are tested and, thus, the possibility of unveiling errors tends to increase.

Define a sound testing strategy for concrete test cases. In this work we prove that our CSP-based testing theory is sound with respect to csptio. To execute CSP-based test cases (symbolic test cases), the implementation needs to be modelled as CSP processes too. When this is not the case, we offer the possibility of generating concrete test cases, which can be seen as purely test data. However, soundness is not guaranteed in this case. Therefore, the systematic generation of sound concrete test cases is an interesting topic for future research.

Investigate compositional properties of csptio. The verification of whether csptio holds for a specification and an implementation model is a costly operation. Therefore, studying compositional properties of csptio plays an important role since it allows the analysis of complex systems by individually verifying its constituent components. For instance, in (NOGUEIRA; SAMPAIO; MOTA, 2014), it is proved that the relation *cspio* is compositional considering that the specification is input-enabled, and the specification and the implementation have the same alphabet.

Investigate compositional properties of the test generation strategy. The definition of the conditions necessary to compositional testing based on the csptio-based generation strategy is an important research topic, since it would minimise the effort of testing a complex system by testing its constituent components. To illustrate this topic, suppose that tests were generated and executed for one component. Now, this component is combined (e.g., sequentially), such that some of its outputs signals are provided as input for a new component. It would minimise the test generation effort if we could reuse part of the tests generated for the first component, when testing the second one.

Evolve the NAT2TEST tool. Despite the current advanced state of the tool, new important features should be considered. For instance, the analyses described in Section 3.3.3 via bounded model checking has not been implemented yet. Other example consists of conformance verification based on csptio. In this case, the verification can be performed by using directly FDR and Z3; it is still to be integrated with the NAT2TEST tool infrastructure.

Perform more empirical analyses. As described in Chapter 5, the NAT2TEST strategy has already been evaluated considering four examples from different domains. Nevertheless, to investigate the potential practical application of the proposed strategy, more empirical analyses need to be performed, considering bigger and more complex systems.

Consider notations other than CSP for modelling DFRSs. For example, in (SILVA; CARVALHO; SAMPAIO, 2015), we started an investigation of using CPNs to represent DFRS models. In this way, we plan to exploit the maturity of tools such as the CPN tools to analyse properties of the system requirements, besides generating test cases. Therefore, investigating how we can represent DFRSs as other formal notations than CSP might allow us to perform analyses we are not able to carry out today.

Evolve the NAT2TEST strategy to deal with hybrid systems. Hybrid systems are a class of embedded systems with a great number of applications (e.g., in cyber-physical systems). In such systems, it is necessary to model the input and output trajectories of the system dynamics (AERTS; MOUSAVI; RENIERS, 2015). This modelling is usually achieved by considering difference and differential equations. Addressing the generation of test cases for hybrid systems will have a vertical impact on the NAT2TEST strategy. In other words, each of its constituent phases needs to evolve in the light of this new domain. For example, associating difference and differential equations to the system requirements, changing the DFRS models to consider this new information, adopting a testing strategy that supports hybrid systems, and so on. Regarding this last topic, in (KHAKPOUR; MOUSAVI, 2015) it is discussed conformance relations for hybrid systems. Therefore, dealing with hybrid systems within the NAT2TEST strategy is an interesting, challenging and relevant research topic for future work.

References

ABRIAL, J.-R. **The B-book**: assigning programs to meanings. New York: Cambridge University Press, 1996.

ACEITUNA, D.; DO, H.; SRINIVASAN, S. A Systematic Approach to Transforming System Requirements into Model Checking Specifications. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, New York. **Proceedings...** ACM, 2014. p.165–174. (ICSE Companion).

AERTS, A.; MOUSAVI, M. R.; RENIERS, M. A Tool Prototype for Model-Based Testing of Cyber-Physical Systems. In: INTERNATIONAL COLLOQUIUM ON THEORETICAL ASPECTS OF COMPUTING, Cali. **Proceedings...** Springer International Publishing, 2015. p.563–572.

AHO, A. V. et al. **Compilers**: principles, techniques, and tools. 2.ed. Essex: Prentice Hall, 2006.

ALLEN, J. **Natural Language Understanding**. California: Benjamin/Cummings, 1995.

AMBRIOLA, V.; GERVASI, V. On the Systematic Analysis of Natural Language Requirements with CIRCE. **Automated Software Engineering**, [S.l.], v.13, n.1, p.107–167, 2006.

ANDRADE, W. et al. Abstracting Time and Data for Conformance Testing of Real-Time Systems. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION WORKSHOPS, Berlin. **Proceedings...** [S.l.: s.n.], 2011. p.9–17.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is Mutation an Appropriate Tool for Testing Experiments? In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, New York. **Proceedings...** ACM, 2005. p.402–411.

ASD. **A Guide for the Preparation of Aircraft Maintenance Documentation in the International Aerospace Maintenance Language, Specification ASD-STE-100**. [S.l.]: AeroSpace and Defence - Industries Association of Europe, 2005. (Issue 3).

ATLEE, J. M.; GANNON, J. State-Based Model Checking of Event-Driven System Requirements. **IEEE Transactions on Software Engineering**, Piscataway, v.19, n.1, p.24–40, Jan. 1993.

BACKES, J. et al. Requirements Analysis of a Quad-Redundant Flight Control System. In: NASA FORMAL METHODS, USA. **Proceedings...** Springer International Publishing, 2015. p.82–96. (Lecture Notes in Computer Science, v.9058).

BERGERAND, J.-L. **Lustre, un Langage Déclaratif pour le Temps Réel** . 1986. Tese (Doutorado em Ciência da Computação) — INPG.

BLACKBURN, M.; BUSSER, R.; FONTAINE, J. Automatic Generation of Test Vectors for SCR-style Specifications. In: ANNUAL CONFERENCE ON COMPUTER ASSURANCE, USA. **Proceedings...** [S.l.: s.n.], 1997.

BODDU, R. et al. RETNA: from requirements to testing in a natural way. In: IEEE INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE, Japan. **Proceedings...** [S.l.: s.n.], 2004. p.262–271.

BOHNENKAMP, H.; BELINFANTE, A. Timed Testing with TorX. In: THE INTERNATIONAL CONFERENCE ON FORMAL METHODS, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2005. p.173–188. (FM'05).

BOZGA, M. et al. IF: a validation environment for timed asynchronous systems. In: COMPUTER AIDED VERIFICATION, Berlin Heidelberg. **Proceedings...** Springer-Verlag, 2000. p.543–547. (Lecture Notes in Computer Science, v.1855).

BRAT, G. et al. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. **Formal Methods in System Design**, Hingham, v.25, n.2-3, p.167–198, Sept. 2004.

BRIONES, L. B.; BRINKSMA, E. A test generation framework for quiescent real-time systems. In: INTERNATIONAL WORKSHOP ON FORMAL APPROACHES TO SOFTWARE TESTING, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2005. p.64–78.

CARVALHO, G. et al. Test Case Generation from Natural Language Requirements based on SCR Specifications. In: SYMPOSIUM ON APPLIED COMPUTING, Coimbra, Portugal. **Proceedings...** [S.l.: s.n.], 2013a. v.2, p.1217–1222.

CARVALHO, G. et al. Model-Based Testing from Controlled Natural Language Requirements. In: WORKSHOP ON FORMAL TECHNIQUES FOR SAFETY-CRITICAL SYSTEMS, New Zealand. **Proceedings...** Springer International Publishing, 2014a. p.19–35. (Communications in Computer and Information Science, v.419).

CARVALHO, G. et al. A Formal Model for Natural-Language Timed Requirements of Reactive Systems. In: INTERNATIONAL CONFERENCE ON FORMAL ENGINEERING METHODS, Luxembourg. **Proceedings...** Springer International Publishing, 2014b. p.43–58. (Lecture Notes in Computer Science, v.8829).

CARVALHO, G. et al. NAT2TEST$_{SCR}$: test case generation from natural language requirements based on SCR specifications. **Science of Computer Programming**, [S.l.], v.95, Part 3, n.0, p.275 – 297, 2014c.

CARVALHO, G. et al. NAT2TEST: from natural language requirements to test cases. **Aerospace America Magazine**, [S.l.], v.52, p.45, 2014d.

CARVALHO, G. et al. NAT2TEST Tool: from natural language requirements to test cases based on CSP. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND FORMAL METHODS, York. **Proceedings...** Springer International Publishing, 2015.

CARVALHO, G.; SAMPAIO, A.; MOTA, A. A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. In: INTERNATIONAL CONFERENCE ON FORMAL ENGINEERING METHODS, New Zealand. **Proceedings...** Springer Berlin Heidelberg, 2013b. p.148–164. (LNCS, v.8144).

CHARETTE, R. This Car Runs on Code. **IEEE Spectrum**, [S.l.], 2009.

CIMATTI, A. et al. NuSMV: a new symbolic model verifier. In: THE 11TH INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION, London. **Proceedings...** Springer-Verlag, 1999. p.495–499. (CAV '99).

CLARKE, E. M.; GRUMBERG, O.; PELED, D. A. **Model Checking**. Cambridge, Massachusetts: The MIT Press, 1999.

EDITION 6th (Ed.). **A Dictionary of Linguistics and Phonetics**. [S.l.]: Wiley-Blackwell, 2008.

EFKEMANN, C.; PELESKA, J. Model-Based Testing for the Second Generation of Integrated Modular Avionics. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION WORKSHOPS, Germany. **Proceedings...** [S.l.: s.n.], 2011. p.55–62.

ESSER, M.; STRUSS, P. Obtaining Models for Test Generation from Natural-Language like Functional Specifications. In: INTERNATIONAL WORKSHOP ON PRINCIPLES OF DIAGNOSIS. **Proceedings...** [S.l.: s.n.], 2007. p.75–82.

FAA. **Requirements Engineering Management Findings Report**. USA: U.S. Department of Transportation - Federal Aviation Administration, 2009.

FILLMORE, C. J. The Case for Case. In: UNIVERSALS IN LINGUISTIC THEORY, USA. **Proceedings...** New York: Holt: Rinehart: and Winston, 1968. p.1–88.

GAUDEL, M. claude. Testing Can Be Formal, too. In: INTERNATIONAL CONFERENCE OF THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT, Denmark. **Proceedings...** Springer-Verlag, 1995. p.82–96.

HAYHURST, K. et al. **A Practical Tutorial on Modified Condition/Decision Coverage**. [S.l.: s.n.], 2001.

HEITMEYER, C.; BHARADWAJ, R. Applying the SCR Requirements Method to the Light Control Case Study. **Journal of Universal Computer Science**, [S.l.], v.6, p.650–678, 2000.

ILIC, D. Deriving Formal Specifications from Informal Requirements. In: INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, China. **Proceedings...** [S.l.: s.n.], 2007. v.1, p.145–152.

ISO. **Z formal specification notation (ISO/IEC 13568)**. [S.l.]: International Organization for Standardization, 2002.

JENSEN, K. **Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use**. [S.l.]: Springer-Verlag1, 1996.

KAMPRATH, C. et al. Controlled Language for Multilingual Document Production: experience with caterpillar technical english. In: SECOND INTERNATIONAL WORKSHOP ON CONTROLLED LANGUAGE APPLICATIONS, Netherlands. **Proceedings...** [S.l.: s.n.], 1998.

KHAKPOUR, N.; MOUSAVI, M. R. Notions of Conformance Testing for Cyber-Physical Systems: overview and roadmap (invited paper). In: INTERNATIONAL CONFERENCE ON CONCURRENCY THEORY, Spain. **Proceedings...** [S.l.: s.n.], 2015. v.42, p.18–40.

KHOUMSI, A.; JÉRON, T.; MARCHAND, H. Test Cases Generation for Nondeterministic Real-Time Systems. In: INTERNATIONAL WORKSHOP ON FORMAL APPROACHES TO TESTING OF SOFTWARE, Canada. **Proceedings...** Springer Berlin Heidelberg, 2003. v.2931, p.131–146.

KRICHEN, M.; TRIPAKIS, S. Black-Box Conformance Testing for Real-Time Systems. In: INTERNATIONAL SPIN WORKSHOP, Spain. **Proceedings...** Springer Berlin Heidelberg, 2004. v.2989, p.109–126.

LARSEN, K.; MIKUCIONIS, M.; NIELSEN, B. Online Testing of Real-time Systems using Uppaal: status and future work. In: DAGSTUHL SEMINAR – PERSPECTIVES OF MODEL-BASED TESTING, Germany. **Proceedings...** [S.l.: s.n.], 2004. v.04371.

LEE, W. J.; CHA, S. D.; KWON, Y. R. Integration and analysis of use cases using modular Petri nets in requirements engineering. **IEEE Transactions on Software Engineering**, [S.l.], v.24, n.12, p.1115–1130, Dec 1998.

LEONARD, E.; HEITMEYER, C. Program Synthesis from Formal Requirements Specifications Using APTS. **Higher Order and Symbolic Computation**, Hingham, MA, USA, v.16, p.63–92, 2003.

LEVESON, N. G. et al. Requirements Specification for Process-Control Systems. **IEEE Transactions on Software Engineering**, Piscataway, v.20, n.9, p.684–707, Sep 1994.

LIU, S. et al. Automatic Early Defects Detection in Use Case Documents. In: THE ACM/IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, New York. **Proceedings...** ACM, 2014. p.785–790. (ASE '14).

MA, Y.-S.; OFFUTT, J.; KWON, Y. R. MuJava: an automated class mutation system. **Software Testing, Verification and Reliability**, Chichester, v.15, n.2, p.97–133, June 2005.

MILLER, S. P. et al. Proving the shalls. **International Journal on Software Tools for Technology Transfer**, [S.l.], v.8, n.4-5, p.303–319, 2006.

NOGUEIRA, S. **Test Generation and Compositional Conformance Verification from Input-Output CSP Models**. 2012. Tese (Doutorado em Ciência da Computação) — Centro de Informática - Universidade Federal de Pernambuco.

NOGUEIRA, S.; SAMPAIO, A.; MOTA, A. Test generation from state based use case models. **Formal Aspects of Computing**, [S.l.], v.26, n.3, p.441–490, 2014.

O'BRIEN, S. Controlling Controlled English - An Analysis of Several Controlled Language Rule Sets. In: JOINT CONFERENCE COMBINING THE 8TH INTERNATIONAL WORKSHOP OF THE EUROPEAN ASSOCIATION FOR MACHINE TRANSLATION (EAMT) AND THE 4TH CONTROLLED LANGUAGE APPLICATIONS WORKSHOP: CONTROLLED LANGUAGE TRANSLATION (CLAW), Ireland. **Proceedings...** [S.l.: s.n.], 2003. p.105–114.

OMG. **Unified Modeling Language**. [S.l.]: Object Management Group, 2015.

OWRE, S.; RUSHBY, J. M.; SHANKAR, N. PVS: A prototype verification system. In: INTERNATIONAL CONFERENCE ON AUTOMATED DEDUCTION, Saratoga. **Proceedings...** Springer-Verlag, 1992. p.748–752. (Lecture Notes in Artificial Intelligence, v.607).

PACHECO, C. et al. Feedback-directed Random Test Generation. In: THE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, Minneapolis. **Proceedings...** IEEE Computer Society, 2007.

PELESKA, J. et al. **Automated Model-Based Testing with RT-Tester**. [S.l.]: Universität Bremen, 2011.

PELESKA, J. et al. A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain. In: THE IFIP INTERNATIONAL CONFERENCE ON TESTING SOFTWARE AND SYSTEMS, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2011. p.146–161. (ICTSS'11).

REED, G. M.; ROSCOE, A. W. A Timed Model for Communicating Sequential Processes. **Theoretical Computer Science**, [S.l.], v.58, p.249–261, 1988.

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

ROSCOE, A. W.; HOARE, C. A. R.; BIRD, R. **The Theory and Practice of Concurrency**. [S.l.]: Prentice Hall PTR, 1997.

SANTIAGO JUNIOR, V.; VIJAYKUMAR, N. L. Generating Model-based Test Cases from Natural Language Requirements for Space Application Software. **Software Quality Journal**, [S.l.], v.20, p.77–143, 2012.

SCHMALTZ, J.; TRETMANS, J. On Conformance Testing for Timed Systems. In: INTERNATIONAL CONFERENCE ON FORMAL MODELLING AND ANALYSIS OF TIMED SYSTEMS, France. **Proceedings...** Springer Berlin Heidelberg, 2008. v.5215, p.250–264.

SCHNELTE, M. Generating Test Cases for Timed Systems from Controlled Natural Language Specifications. In: INTERNATIONAL CONFERENCE ON SYSTEM INTEGRATION AND RELIABILITY IMPROVEMENTS, USA. **Proceedings...** [S.l.: s.n.], 2009. p.348–353.

SCHWITTER, R. English as a Formal Specification Language. In: THE INTERNATIONAL WORKSHOP ON DATABASE AND EXPERT SYSTEMS APPLICATIONS, France. **Proceedings...** [S.l.: s.n.], 2002.

SIEGL, S.; HIELSCHER, K.-S.; GERMAN, R. Model Based Requirements Analysis and Testing of Automotive Systems with Timed Usage Models. In: THE IEEE INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE, Australia. **Proceedings...** [S.l.: s.n.], 2010. p.345–350.

SILVA, B. C. F.; CARVALHO, G.; SAMPAIO, A. Test Case Generation from Natural Language Requirements Using CPN Simulation. In: BRAZILIAN SYMPOSIUM ON FORMAL METHODS, Belo Horizonte, Brazil. **Proceedings...** [S.l.: s.n.], 2015. p.178–193.

SNEED, H. Testing against Natural Language Requirements. In: THE INTERNATIONAL CONFERENCE ON QUALITY SOFTWARE, USA. **Proceedings...** [S.l.: s.n.], 2007. p.380–387.

TOMITA, M. **Efficient Parsing for Natural Language**. [S.l.]: Kluwer Academic Publishers, 1986.

TRETMANS, J. Testing Concurrent Systems: a formal approach. In: INTERNATIONAL CONFERENCE ON CONCURRENCY THEORY, London. **Proceedings...** Springer-Verlag, 1999. p.46–65.

WEST, A. **NASA Study on Flight Software Complexity**. [S.l.]: NASA, 2009.

WYNER, A. et al. On Controlled Natural Languages: properties and prospects. In: CONTROLLED NATURAL LANGUAGE, Berlin, Heidelberg. **Proceedings...** Springer Berlin Heidelberg, 2010. p.281–289. (Lecture Notes in Computer Science, v.5972).

Appendix

A
List of requirements

A.1 Vending machine

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

A.2 Nuclear power plant

- When the water pressure becomes greater than or equal to 9, and the pressure mode is low, the Safety Injection System shall assign permitted to the pressure mode.

- When the water pressure becomes greater than or equal to 10, and the pressure mode is permitted, the Safety Injection System shall assign high to the pressure mode.
- When the water pressure becomes lower than 9, and the pressure mode is permitted, the Safety Injection System shall assign low to the pressure mode.
- When the water pressure becomes lower than 10, and the pressure mode is high, the Safety Injection System shall assign permitted to the pressure mode.
- When the blockage button becomes pressed, and the reset button is not pressed, and the pressure mode is not high, the Safety Injection System shall assign true to the overridden mode.
- When the reset button becomes pressed, and the pressure mode is not high, the Safety Injection System shall assign false to the overridden mode.
- When the pressure mode becomes high, the Safety Injection System shall assign false to the overridden mode.
- When the pressure mode becomes not high, the Safety Injection System shall assign false to the overridden mode.
- When the pressure mode is low, and the overridden mode is true, the Safety Injection System shall assign off to the safety injection mode.
- When the pressure mode is low, and the overridden mode is false, the Safety Injection System shall assign on to the safety injection mode.
- When the pressure mode is permitted or the pressure mode is high, the Safety Injection System shall assign off to the safety injection mode.

A.3 Priority command

These requirements are not presented since they concern private information.

A.4 Turn indicator system

- ACTION_1: Reset the flashing timer
- ACTION_4: Assign off to the indication lights
- ACTION_5: Assign left flashing to the indication lights
- ACTION_6: Assign right flashing to the indication lights

- ACTION_7: Assign both flashing to the indication lights
- When the voltage becomes lower than or equal to 80, the lights controller component shall: ACTION_4, ACTION_1.
- When the voltage becomes greater than 80 or the flashing mode becomes left flashing, and the voltage is greater than 80, and the flashing mode is left flashing, the lights controller component shall: ACTION_5, ACTION_1.
- When the voltage becomes greater than 80 or the flashing mode becomes right flashing, and the voltage is greater than 80, and the flashing mode is right flashing, the lights controller component shall: ACTION_6, ACTION_1.
- When the voltage becomes greater than 80 or the flashing mode becomes both flashing, and the voltage is greater than 80, and the flashing mode is both flashing, the lights controller component shall: ACTION_7, ACTION_1.
- When the voltage is greater than 80, and the flashing mode is no flashing, the lights controller component shall: ACTION_4, ACTION_1.
- When the voltage is greater than 80, and the flashing timer is greater than or equal to 340, and the indication lights are left flashing or the indication lights are right flashing, the lights controller component shall: ACTION_4, ACTION_1.
- When the voltage is greater than 80, and the flashing timer is greater than or equal to 220, and the indication lights are off, and the flashing mode is left flashing, the lights controller component shall: ACTION_5, ACTION_1.
- When the voltage is greater than 80, and the flashing timer is greater than or equal to 220, and the indication lights are off, and the flashing mode is right flashing, the lights controller component shall: ACTION_6, ACTION_1.
- When the voltage is greater than 80, and the flashing timer is greater than or equal to 220, and the indication lights are off, and the flashing mode is both flashing, the lights controller component shall: ACTION_7, ACTION_1.
- When the turn indicator lever changes to the left position, and the emergency flashing is off, the flashing mode component shall: assign left flashing to the flashing mode, ACTION_1.
- When the turn indicator lever changes to the right position, and the emergency flashing is off, the flashing mode component shall: assign right flashing to the flashing mode, ACTION_1.

- When the emergency flashing becomes on, the flashing mode component shall: assign both flashing to the flashing mode, ACTION_1.
- When the emergency flashing is on, and the emergency flashing was on, and the turn indicator lever changes to the left position, the flashing mode component shall: assign left flashing to the flashing mode, ACTION_1.
- When the emergency flashing is on, and the emergency flashing was on, and the turn indicator lever changes to the right position, the flashing mode component shall: assign right flashing to the flashing mode, ACTION_1.
- When the emergency flashing is on, and the emergency flashing was on, and the turn indicator lever changes to the idle position, and the flashing mode is not both flashing, the flashing mode component shall: assign both flashing to the flashing mode, ACTION_1.
- When the emergency flashing becomes off, and the turn indicator lever is on the left position, and the turn indicator lever was on the left position, and the flashing mode is not left flashing, the flashing mode component shall: assign left flashing to the flashing mode, ACTION_1.
- When the emergency flashing becomes off, and the turn indicator lever is on the right position, and the turn indicator lever was on the right position, and the flashing mode is not right flashing, the flashing mode component shall: assign right flashing to the flashing mode, ACTION_1.

B

DFRS – definitions and proofs

B.1 Definition and properties of an s-DFRS

B.1.1 Inputs, Outpus and Timers

$[NAME]$

$$\begin{array}{|l} gc : NAME \end{array}$$

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

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

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

B.1.2 Initial state

$BOOL\_VALUES ::= TRUE \mid FALSE$

$[R]$

$$R^+ : \mathbb{P}\,R$$
$$R^+ \subset R$$

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

$$STATE == NAME \rightarrow\!\!\!\!\!+ (VALUE \times VALUE)$$

$$previousValues : STATE \rightarrow (NAME \rightarrow\!\!\!\!\!+ VALUE)$$
$$\forall\, s : STATE \bullet previousValues(s) = \{n : NAME;\, v1, v2 : VALUE \mid (n, (v1, v2)) \in s \bullet (n, v1)\}$$

$$currentValues : STATE \rightarrow (NAME \rightarrow\!\!\!\!\!+ VALUE)$$
$$\forall\, s : STATE \bullet currentValues(s) = \{n : NAME;\, v1, v2 : VALUE \mid (n, (v1, v2)) \in s \bullet (n, v2)\}$$

$$DFRS\_INITIAL\_STATE == [s_0 : STATE]$$

B.1.3 Functions

$$values : TYPE \rightarrow \mathbb{P}\,VALUE$$
$$values(bool) = \operatorname{ran} b$$
$$values(int) = \operatorname{ran} i$$
$$values(nat) = \operatorname{ran} n$$
$$values(float) = \operatorname{ran} f$$
$$values(ufloat) = \operatorname{ran} uf$$

$$well\_typed\_var : \mathbb{P}(STATE \times NAME \times TYPE)$$
$$\forall\, s : STATE;\, n : NAME;\, t : TYPE \bullet (s, n, t) \in well\_typed\_var \Leftrightarrow n \in \operatorname{dom} s \land (s(n)).1 \in values(t) \land (s(n)).2 \in values(t)$$

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

$$\begin{array}{l}
\mathit{VAR} ::= \mathit{current}\langle\langle \mathit{VNAME} \rangle\rangle \mid \mathit{previous}\langle\langle \mathit{VNAME} \rangle\rangle \\
\mathit{OP} ::= \mathit{le} \mid \mathit{lt} \mid \mathit{eq} \mid \mathit{ne} \mid \mathit{gt} \mid \mathit{ge} \\
\mathit{BEXP} == \{v : \mathit{VAR}; \mathit{op} : \mathit{OP}; \mathit{literal} : \mathit{VALUE}\} \\
\mathit{DISJ} == \mathbb{F}_1\, \mathit{BEXP} \\
\mathit{CONJ} == \mathbb{F}\, \mathit{DISJ} \\
\mathit{EXP} == \mathit{CONJ}
\end{array}$$

$$\begin{array}{|l}
\mathit{varName} : \mathit{BEXP} \rightarrow \mathit{VNAME} \\
\hline
\forall \mathit{be} : \mathit{BEXP}; n : \mathit{VNAME} \bullet \mathit{varName}(\mathit{be}) = n \Leftrightarrow \\
\quad (\mathit{be}.1 \in \mathrm{ran}\, \mathit{previous} \Rightarrow \mathit{previous}(n) = \mathit{be}.1) \land \\
\quad (\mathit{be}.1 \in \mathrm{ran}\, \mathit{current} \Rightarrow \mathit{current}(n) = \mathit{be}.1)
\end{array}$$

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

$$\begin{array}{|l}
\mathit{var\_consistent\_dis} : \mathbb{P}(\mathit{DISJ} \times (\mathit{VNAME} \nrightarrow \mathit{TYPE}) \times (\mathit{VNAME} \nrightarrow \mathit{TYPE})) \\
\hline
\forall \mathit{dis} : \mathit{DISJ}; f, T : \mathit{VNAME} \nrightarrow \mathit{TYPE} \bullet \\
\quad (\mathit{dis}, f, T) \in \mathit{var\_consistent\_dis} \Leftrightarrow \\
\quad (\forall \mathit{be} : \mathit{dis} \bullet (\mathit{be}, f, T) \in \mathit{var\_consistent\_be})
\end{array}$$

$$\begin{array}{|l}
\mathit{var\_consistent\_conj} : \mathbb{P}(\mathit{CONJ} \times (\mathit{VNAME} \nrightarrow \mathit{TYPE}) \times (\mathit{VNAME} \nrightarrow \mathit{TYPE})) \\
\hline
\forall \mathit{conj} : \mathit{CONJ}; f, T : \mathit{VNAME} \nrightarrow \mathit{TYPE} \bullet \\
\quad (\mathit{conj}, f, T) \in \mathit{var\_consistent\_conj} \Leftrightarrow \\
\quad (\forall \mathit{dis} : \mathit{conj} \bullet (\mathit{dis}, f, T) \in \mathit{var\_consistent\_dis})
\end{array}$$

$$\begin{array}{|l}
var\_consistent\_exp : \mathbb{P}(EXP \times (VNAME \rightarrow\!\!\!\!\!\!+\; TYPE) \times (VNAME \rightarrow\!\!\!\!\!\!+\; TYPE)) \\
\hline
\forall exp : EXP; f, T : VNAME \rightarrow\!\!\!\!\!\!+\; TYPE \bullet \\
\qquad (exp, f, T) \in var\_consistent\_exp \Leftrightarrow (exp, f, T) \in var\_consistent\_conj
\end{array}$$

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

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

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

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

B.1.4 Complete definition

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

B.2 Definition and properties of an e-DFRS

B.2.1 Transition relation

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

$$R_1^+ : \mathbb{P} R^+$$

$$R_1^+ \subset R^+$$

$$\mathit{DELAY} ::= \mathit{discrete}\langle\langle \mathbb{N}_1 \rangle\rangle \mid \mathit{dense}\langle\langle R_1^+ \rangle\rangle$$
$$\mathit{TRANS\_LABEL} ::= \mathit{fun}\langle\langle \mathit{ASGMTS} \rangle\rangle \mid \mathit{del}\langle\langle \mathit{DELAY} \times \mathit{ASGMTS} \rangle\rangle$$
$$\mathit{TRANS} == (\mathit{STATE} \times \mathit{TRANS\_LABEL} \times \mathit{STATE})$$
$$\mathit{TRANSREL} == \mathbb{P}\, \mathit{TRANS}$$

$$\mathit{functionTransition} : \mathit{TRANS\_LABEL} \pfun \mathit{ASGMTS}$$

$$\operatorname{dom} \mathit{functionTransition} = \operatorname{ran} \mathit{fun}$$
$$\forall \mathit{label} : \mathit{TRANS\_LABEL} \mid \mathit{label} \in \operatorname{ran} \mathit{fun} \bullet \mathit{functionTransition}(\mathit{label}) = (\mathit{fun}^{\sim})(\mathit{label})$$

$$\mathit{delayTransition} : \mathit{TRANS\_LABEL} \pfun \mathit{DELAY} \times \mathit{ASGMTS}$$

$$\operatorname{dom} \mathit{delayTransition} = \operatorname{ran} \mathit{del}$$
$$\forall \mathit{label} : \mathit{TRANS\_LABEL} \mid \mathit{label} \in \operatorname{ran} \mathit{del} \bullet \mathit{delayTransition}(\mathit{label}) = (\mathit{del}^{\sim})(\mathit{label})$$

$$\mathit{well\_typed\_function\_transition} : \mathbb{P}(\mathit{TRANS\_LABEL} \times (\mathit{VNAME} \pfun \mathit{TYPE}) \times (\mathit{VNAME} \pfun \mathit{TYPE}))$$

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

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

$$\forall \mathit{label} : \mathit{TRANS\_LABEL}; I : \mathit{VNAME} \pfun \mathit{TYPE} \mid \mathit{label} \in \operatorname{ran} \mathit{del} \bullet (\mathit{label}, I) \in \mathit{well\_typed\_delay\_transition} \Leftrightarrow \operatorname{dom}(\mathit{delayTransition}(\mathit{label})).2 = \operatorname{dom} I$$

$$\begin{array}{l}
\mathit{clock\_compatible\_transition} : \mathbb{P}(\mathit{TRANS\_LABEL} \times (\mathit{NAME} \times \mathit{TYPE})) \\
\hline
\forall \mathit{label} : \mathit{TRANS\_LABEL} ; \mathit{gcvar} : \mathit{NAME} \times \mathit{TYPE} \bullet \\
\quad (\mathit{label}, \mathit{gcvar}) \in \mathit{clock\_compatible\_transition} \Leftrightarrow \\
\quad \mathit{label} \in \operatorname{ran} \mathit{del} \wedge \\
\quad ((\mathit{delayTransition}(\mathit{label})).1 \in \operatorname{ran} \mathit{discrete} \Rightarrow \mathit{gcvar}.2 = \mathit{nat}) \wedge \\
\quad ((\mathit{delayTransition}(\mathit{label})).1 \in \operatorname{ran} \mathit{dense} \Rightarrow \mathit{gcvar}.2 = \mathit{ufloat})
\end{array}$$

$$\begin{array}{l}
\mathit{well\_typed\_transition} : \mathbb{P}(\mathit{TRANS\_LABEL} \times (\mathit{VNAME} \nrightarrow \mathit{TYPE}) \times \\
\quad (\mathit{VNAME} \nrightarrow \mathit{TYPE}) \times (\mathit{VNAME} \nrightarrow \mathit{TYPE}) \times (\mathit{NAME} \times \mathit{TYPE})) \\
\hline
\forall \mathit{label} : \mathit{TRANS\_LABEL} ; I, O, T : \mathit{VNAME} \nrightarrow \mathit{TYPE} ; \mathit{gcvar} : \mathit{NAME} \times \mathit{TYPE} \bullet \\
\quad (\mathit{label}, I, O, T, \mathit{gcvar}) \in \mathit{well\_typed\_transition} \Leftrightarrow \\
\quad (\mathit{label} \in \operatorname{ran} \mathit{fun} \Rightarrow (\mathit{label}, O, T) \in \mathit{well\_typed\_function\_transition}) \wedge \\
\quad (\mathit{label} \in \operatorname{ran} \mathit{del} \Rightarrow (\mathit{label}, I) \in \mathit{well\_typed\_delay\_transition} \wedge \\
\qquad (\mathit{label}, \mathit{gcvar}) \in \mathit{clock\_compatible\_transition})
\end{array}$$

$$\begin{array}{l}
\mathit{DFRS\_TRANSITION\_RELATION} \\
\hline
\mathit{TR} : \mathit{TRANSREL} \\
\hline
\forall \mathit{trans}1, \mathit{trans}2 : \mathit{TR} \mid \mathit{trans}1.1 = \mathit{trans}2.1 \bullet \\
\quad \{\mathit{trans}1.2, \mathit{trans}2.2\} \subseteq \operatorname{ran} \mathit{fun} \vee \{\mathit{trans}1.2, \mathit{trans}2.2\} \subseteq \operatorname{ran} \mathit{del} \\
\forall \mathit{trans} : \mathit{TR} \bullet \neg\ (\mathit{trans}.1 = \mathit{trans}.3) \\
\hline
\end{array}$$

B.2.2 Complete definition

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

$$\mathit{addR} : (R \times R) \rightarrow R$$

```
e_DFRS
  DFRS_VARIABLES
  DFRS_STATES
  DFRS_TRANSITION_RELATION
  ----------------
  $\forall s : S \bullet (s, I \cup O \cup T \cup \{gcvar\}) \in well\_typed\_state$
  $\forall trans : TR \bullet \{trans.1, trans.3\} \subseteq S \land$
      $(trans.2, I, O, T, gcvar) \in well\_typed\_transition \land$
      $(trans.2 \in \operatorname{ran} fun \Rightarrow trans.3 =$
          $nextState(trans.1, T, functionTransition(trans.2))) \land$
      $(trans.2 \in \operatorname{ran} del \Rightarrow$
          $(((del^{\sim})(trans.2)).1 \in \operatorname{ran} discrete \Rightarrow trans.3 =$
              $nextState(trans.1, T, (delayTransition(trans.2)).2) \oplus$
              $\{(gc, ((trans.1(gc)).2,$
                  $n((n^{\sim})((trans.1(gc)).2) +$
                      $(discrete^{\sim})((delayTransition(trans.2)).1))))\}) \land$
          $(((del^{\sim})(trans.2)).1 \in \operatorname{ran} dense \Rightarrow trans.3 =$
              $nextState(trans.1, T, (delayTransition(trans.2)).2) \oplus$
              $\{(gc, ((trans.1(gc)).2,$
                  $uf(addR((uf^{\sim})((trans.1(gc)).2),$
                      $(dense^{\sim})((delayTransition(trans.2)).1)))))\}))$
```

B.3 From s-DFRSs to e-DFRSs

$$\begin{array}{l}
static\_bexps\_true : \mathbb{P}((NAME \pfun VALUE) \times BEXP) \\
\hline
\forall f : (NAME \pfun VALUE) ; be : BEXP \bullet (f, be) \in static\_bexps\_true \Leftrightarrow \\
\quad (f(varName(be)) \in \operatorname{ran} b \Rightarrow \\
\qquad (b^{\sim})(f(varName(be))) = (b^{\sim})(be.3)) \land \\
\quad (f(varName(be)) \in \operatorname{ran} i \Rightarrow \\
\qquad (be.2 = le \Rightarrow (i^{\sim})(f(varName(be))) \leq (i^{\sim})(be.3)) \land \\
\qquad (be.2 = lt \Rightarrow (i^{\sim})(f(varName(be))) < (i^{\sim})(be.3)) \land \\
\qquad (be.2 = eq \Rightarrow (i^{\sim})(f(varName(be))) = (i^{\sim})(be.3)) \land \\
\qquad (be.2 = ne \Rightarrow (i^{\sim})(f(varName(be))) \neq (i^{\sim})(be.3)) \land \\
\qquad (be.2 = gt \Rightarrow (i^{\sim})(f(varName(be))) > (i^{\sim})(be.3)) \land \\
\qquad (be.2 = ge \Rightarrow (i^{\sim})(f(varName(be))) \geq (i^{\sim})(be.3))) \land \\
\quad (f(varName(be)) \in \operatorname{ran} n \Rightarrow \\
\qquad (be.2 = le \Rightarrow (n^{\sim})(f(varName(be))) \leq (n^{\sim})(be.3)) \land \\
\qquad (be.2 = lt \Rightarrow (n^{\sim})(f(varName(be))) < (n^{\sim})(be.3)) \land \\
\qquad (be.2 = eq \Rightarrow (n^{\sim})(f(varName(be))) = (n^{\sim})(be.3)) \land \\
\qquad (be.2 = ne \Rightarrow (n^{\sim})(f(varName(be))) \neq (n^{\sim})(be.3)) \land \\
\qquad (be.2 = gt \Rightarrow (n^{\sim})(f(varName(be))) > (n^{\sim})(be.3)) \land \\
\qquad (be.2 = ge \Rightarrow (n^{\sim})(f(varName(be))) \geq (n^{\sim})(be.3)))
\end{array}$$

$$\begin{array}{l}
timed\_bexps\_true : \mathbb{P}((NAME \pfun VALUE) \times BEXP) \\
\hline
\forall f : (VNAME \pfun VALUE) ; be : BEXP \bullet (f, be) \in timed\_bexps\_true \Leftrightarrow \\
\quad (f(varName(be)) \in \operatorname{ran} n \Rightarrow \\
\qquad (be.2 = le \Rightarrow ((n^{\sim})(f(gc)) - (n^{\sim})(f(varName(be)))) \leq (n^{\sim})(be.3)) \land \\
\qquad (be.2 = lt \Rightarrow ((n^{\sim})(f(gc)) - (n^{\sim})(f(varName(be)))) < (n^{\sim})(be.3)) \land \\
\qquad (be.2 = eq \Rightarrow ((n^{\sim})(f(gc)) - (n^{\sim})(f(varName(be)))) = (n^{\sim})(be.3)) \land \\
\qquad (be.2 = ne \Rightarrow ((n^{\sim})(f(gc)) - (n^{\sim})(f(varName(be)))) \neq (n^{\sim})(be.3)) \land \\
\qquad (be.2 = gt \Rightarrow ((n^{\sim})(f(gc)) - (n^{\sim})(f(varName(be)))) > (n^{\sim})(be.3)) \land \\
\qquad (be.2 = ge \Rightarrow ((n^{\sim})(f(gc)) - (n^{\sim})(f(varName(be)))) \geq (n^{\sim})(be.3)))
\end{array}$$

$$
\begin{array}{|l}
static\_guards\_true : \mathbb{P}(STATE \times EXP \times (VNAME \pfun TYPE) \times (VNAME \pfun TYPE)) \\
\hline
\forall s : STATE ; sGuard : EXP ; IO, T : (VNAME \pfun TYPE) \bullet \\
\quad (s, sGuard, IO, T) \in static\_guards\_true \Leftrightarrow \\
\quad (sGuard, IO, T) \in var\_consistent\_exp \land \\
\quad \forall dis : sGuard \bullet \exists be : dis \bullet \\
\qquad (be.1 \in \operatorname{ran} previous \Rightarrow (previousValues(s), be) \in static\_bexps\_true) \land \\
\qquad (be.1 \in \operatorname{ran} current \Rightarrow (currentValues(s), be) \in static\_bexps\_true)
\end{array}
$$

$$
\begin{array}{|l}
timed\_guards\_true : \mathbb{P}(STATE \times EXP \times (VNAME \pfun TYPE)) \\
\hline
\forall s : STATE ; tGuard, tGuard : EXP ; T : (VNAME \pfun TYPE) \bullet \\
\quad (s, tGuard, T) \in timed\_guards\_true \Leftrightarrow \\
\quad (tGuard, T, T) \in var\_consistent\_exp \land \\
\quad \forall dis : tGuard \bullet \exists be : dis \bullet \\
\qquad (be.1 \in \operatorname{ran} previous \Rightarrow (previousValues(s), be) \in timed\_bexps\_true) \land \\
\qquad (be.1 \in \operatorname{ran} current \Rightarrow (currentValues(s), be) \in timed\_bexps\_true)
\end{array}
$$

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

$$
\begin{array}{|l}
enablingDelays : (STATE \times (NAME \pfun TYPE) \times (NAME \pfun TYPE) \times FUNCTION) \pfun \\
\quad \mathbb{P} DELAY \\
\hline
\operatorname{dom} enablingDelays = \{s : STATE ; IO : (NAME \pfun TYPE) ; T : (NAME \pfun TYPE); \\
\quad entry : FUNCTION \mid (s, IO, T, \{\{entry\}\}) \in is\_stable \bullet (s, IO, T, entry)\} \\
\forall s : STATE ; IO : (NAME \pfun TYPE) ; T : (NAME \pfun TYPE) ; entry : FUNCTION; \\
\quad delays : \mathbb{P} DELAY \bullet enablingDelays(s, IO, T, entry) = delays \Leftrightarrow \\
\quad ((s(gc)).2 \in values(nat) \Rightarrow delays = \{delay : DELAY ; next : STATE \mid next = s \\
\qquad \oplus \{(gc, ((s(gc)).2, n((n^{\sim})((s(gc)).2) + (discrete^{\sim})(delay))))\} \land \\
\qquad (next, IO, T, \{\{entry\}\}) \notin is\_stable \bullet delay\}) \land \\
\quad ((s(gc)).2 \in values(ufloat) \Rightarrow delays = \{delay : DELAY ; next : STATE \mid next = s \\
\qquad \oplus \{(gc, ((s(gc)).2, uf(addR((uf^{\sim})((s(gc)).2), (dense^{\sim})(delay)))))\} \land \\
\qquad (next, IO, T, \{\{entry\}\}) \notin is\_stable \bullet delay\})
\end{array}
$$

$$leqR : \mathbb{P}(R \times R)$$

$$\begin{array}{l}
maxDelays : (STATE \times (NAME \rightarrow\!\!\!\!\!+ TYPE) \times (NAME \rightarrow\!\!\!\!\!+ TYPE) \times \\
\quad (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION)) \rightarrow\!\!\!\!\!+ \mathbb{F}\, DELAY \\
\hline
\mathrm{dom}\, maxDelays = \{s : STATE ; IO : (NAME \rightarrow\!\!\!\!\!+ TYPE) ; T : (NAME \rightarrow\!\!\!\!\!+ TYPE); \\
\quad F : (\mathbb{F}_1\, \mathbb{F}_1\, FUNCTION) \mid (s, IO, T, F) \in is\_stable \bullet (s, IO, T, F)\} \\
\forall s : STATE ; IO : (NAME \rightarrow\!\!\!\!\!+ TYPE) ; T : (NAME \rightarrow\!\!\!\!\!+ TYPE) ; F : \mathbb{F}_1\, \mathbb{F}_1\, FUNCTION; \\
\quad maxdelays : \mathbb{F}\, DELAY \bullet maxDelays(s, IO, T, F) = maxdelays \Leftrightarrow \\
\quad ((s(gc)).2 \in values(nat) \Rightarrow maxdelays = \\
\qquad \{f : F ; entry : FUNCTION ; delays : \mathbb{P}\, DELAY ; upperBound : \mathbb{N}_1 \mid \\
\qquad\quad entry \in f \wedge delays = enablingDelays(s, IO, T, entry) \wedge \\
\qquad\quad discrete(upperBound) \in delays \wedge \\
\qquad\quad (\forall n : delays \bullet (discrete^{\sim})\,(n) \leq upperBound) \bullet \\
\qquad\qquad discrete(upperBound)\}) \wedge \\
\quad ((s(gc)).2 \in values(ufloat) \Rightarrow maxdelays = \\
\qquad \{f : F ; entry : FUNCTION ; delays : \mathbb{P}\, DELAY ; upperBound : R_1^{+} \mid \\
\qquad\quad entry \in f \wedge delays = enablingDelays(s, IO, T, entry) \wedge \\
\qquad\quad dense(upperBound) \in delays \wedge \\
\qquad\quad (\forall n : delays \bullet ((dense^{\sim})\,(n), upperBound) \in leqR) \bullet \\
\qquad\qquad dense(upperBound)\})
\end{array}$$

$$\begin{array}{l}
minimumDelay : \mathbb{F}_1\, DELAY \rightarrow\!\!\!\!\!+ DELAY \\
\hline
\mathrm{dom}\, minimumDelay \subseteq \{delays : \mathbb{F}_1\, DELAY \mid \\
\quad (\forall d : delays \bullet d \in \mathrm{ran}\, discrete) \vee (\forall d : delays \bullet d \in \mathrm{ran}\, dense)\} \\
\forall delays : \mathbb{F}_1\, DELAY ; lowerBound : DELAY \bullet \\
\quad minimumDelay(delays) = lowerBound \Leftrightarrow \\
\quad lowerBound \in delays \wedge \\
\quad ((\forall d : delays \bullet d \in \mathrm{ran}\, discrete) \Rightarrow \\
\qquad (\forall d : delays \bullet (discrete^{\sim})(lowerBound) \leq (discrete^{\sim})(d))) \wedge \\
\quad ((\forall d : delays \bullet d \in \mathrm{ran}\, dense) \Rightarrow \\
\qquad (\forall d : delays \bullet ((dense^{\sim})(lowerBound), (dense^{\sim})(d)) \in leqR))
\end{array}$$

$$
\begin{array}{l}
genPossibleDelays : (STATE \times (NAME \pfun TYPE) \times (NAME \pfun TYPE) \times \\
\quad (\mathbb{F}_1 \mathbb{F}_1 FUNCTION)) \pfun \mathbb{P} DELAY \\
\hline
\operatorname{dom} genPossibleDelays = \{ s : STATE ; IO : (NAME \pfun TYPE) ; T : (NAME \pfun TYPE); \\
\quad F : (\mathbb{F}_1 \mathbb{F}_1 FUNCTION) \mid (s, IO, T, F) \in is\_stable \bullet (s, IO, T, F) \} \\
\forall s : STATE ; IO : (NAME \pfun TYPE) ; T : (NAME \pfun TYPE) ; F : \mathbb{F}_1 \mathbb{F}_1 FUNCTION \bullet \\
\quad (maxDelays(s, IO, T, F) = \emptyset \Rightarrow \\
\quad\quad genPossibleDelays(s, IO, T, F) = \{ delay : DELAY \}) \land \\
\quad (maxDelays(s, IO, T, F) \neq \emptyset \Rightarrow \\
\quad\quad ((s(gc)).2 \in values(nat) \Rightarrow genPossibleDelays(s, IO, T, F) = \\
\quad\quad\quad \{ delay : DELAY \mid (discrete^{\sim})(delay) \leq \\
\quad\quad\quad\quad (discrete^{\sim})(minimumDelay(maxDelays(s, IO, T, F))) \}) \land \\
\quad\quad ((s(gc)).2 \in values(ufloat) \Rightarrow genPossibleDelays(s, IO, T, F) = \\
\quad\quad\quad \{ delay : DELAY \mid ((dense^{\sim})(delay), \\
\quad\quad\quad\quad (dense^{\sim})(minimumDelay(maxDelays(s, IO, T, F)))) \in leqR \}))
\end{array}
$$

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

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

$$expandedDFRS : s\_DFRS \rightarrow e\_DFRS$$

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

B.4 Formal model of TIOTSs

$[TIOTS\_ACTION]$

$TIOTS\_ACTIONS == \mathbb{P}\, TIOTS\_ACTION$

$TIOTS\_DELAY ::= tiots\_discrete\langle\langle \mathbb{N} \rangle\rangle \mid tiots\_dense\langle\langle R^+ \rangle\rangle$

$TIOTS\_DELAYS == \mathbb{P}\, TIOTS\_DELAY$

$$tiots\_time\_compatible : \mathbb{P}\, TIOTS\_DELAYS$$

$$\begin{aligned}
&\forall D : TIOTS\_DELAYS \bullet D \in tiots\_time\_compatible \Leftrightarrow \\
&\quad (\forall d : D \bullet d \in \operatorname{ran} tiots\_discrete) \lor \\
&\qquad (\forall d : D \bullet d \in \operatorname{ran} tiots\_dense)
\end{aligned}$$

TIOTS_LABELS

$I, O : TIOTS\_ACTIONS$

$D : TIOTS\_DELAYS$

$\text{disjoint}\ \langle I, O \rangle$

$D \in tiots\_time\_compatible$

$[TIOTS\_STATE]$

$TIOTS\_STATES\_SET == \mathbb{P}_1\, TIOTS\_STATE$

$TIOTS\_STATES == [Q : TIOTS\_STATES\_SET ; q_0 : TIOTS\_STATE \mid q_0 \in Q]$

$TIOTS\_TRANS\_LABEL ::= in\langle\langle TIOTS\_ACTION \rangle\rangle \mid out\langle\langle TIOTS\_ACTION \rangle\rangle \mid$
$\quad tiots\_del\langle\langle TIOTS\_DELAY \rangle\rangle \mid tau$

$TIOTS\_TRANS == (TIOTS\_STATE \times TIOTS\_TRANS\_LABEL \times TIOTS\_STATE)$

$TIOTS\_TRANSREL == \mathbb{P}\, TIOTS\_TRANS$

$$\begin{array}{|l}
\textit{TIOTS\_TRANSITION\_RELATION} \\
\hline
T : \textit{TIOTS\_TRANSREL} \\
\hline
\end{array}$$

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

$$\begin{array}{|l}
\textit{TIOTS} \\
\hline
\textit{TIOTS\_LABELS} \\
\textit{TIOTS\_STATES} \\
\textit{TIOTS\_TRANSITION\_RELATION} \\
\hline
\forall \textit{entry} : T \bullet \{\textit{entry}.1, \textit{entry}.3\} \subseteq Q \land (\textit{entry}.2, I, O, D) \in \textit{well\_typed\_tiots\_transition} \\
\hline
\end{array}$$

B.5 From e-DFRSs to TIOTSs

$$\begin{array}{|l}
\textit{mapState} : \textit{STATE} \rightarrowtail \textit{TIOTS\_STATE}
\end{array}$$

$$\begin{array}{|l}
\textit{genAction} : (\textit{NAME} \nrightarrow \textit{VALUE}) \mathrel{\rightarrowtail\!\!\!\!\!\nrightarrow} \textit{TIOTS\_ACTION} \\
\hline
\operatorname{dom} \textit{genAction} \neq \emptyset
\end{array}$$

$$\begin{array}{|l}
\textit{mapDelay} : \textit{DELAY} \rightarrowtail \textit{TIOTS\_DELAY} \\
\hline
\forall n1 : \mathbb{N}_1 ; n2 : R_1^{+} \bullet \\
\qquad \textit{mapDelay}(\textit{discrete}(n1)) = \textit{tiots\_discrete}(n1) \land \\
\qquad \textit{mapDelay}(\textit{dense}(n2)) = \textit{tiots\_dense}(n2)
\end{array}$$

$$mapFunTransitionOut : (TRANS \times (NAME \pfun TYPE)) \pfun TIOTS\_TRANSREL$$

$$\begin{aligned}
&dom(mapFunTransitionOut) = (STATE \times \operatorname{ran} fun \times STATE) \times (NAME \pfun TYPE)\\
&\forall trans : TRANS ; O : (NAME \pfun TYPE) \mid trans.2 \in \operatorname{ran} fun \bullet\\
&\quad mapFunTransitionOut(trans, O) = \{(mapState(trans.1),\\
&\qquad out(genAction(currentValues(\operatorname{dom} O \lhd trans.3))), mapState(trans.3))\}
\end{aligned}$$

$$mapFunTransitionTau : TRANS \pfun TIOTS\_TRANSREL$$

$$\begin{aligned}
&dom(mapFunTransitionTau) = (STATE \times \operatorname{ran} fun \times STATE)\\
&\forall trans : TRANS \mid trans.2 \in \operatorname{ran} fun \bullet\\
&\quad mapFunTransitionTau(trans) = \{(mapState(trans.1), tau, mapState(trans.3))\}
\end{aligned}$$

$$\begin{aligned}
&mapFunTransitions : (TRANSREL \times TRANSREL \times (NAME \pfun TYPE)) \pfun\\
&\quad TIOTS\_TRANSREL
\end{aligned}$$

$$\begin{aligned}
&\operatorname{dom} mapFunTransitions = \{tr : TRANSREL \mid \forall trans : tr \bullet trans.2 \in \operatorname{ran} fun\} \times\\
&\quad TRANSREL \times (NAME \pfun TYPE)\\
&\forall funTR, tr : TRANSREL ; O : (NAME \pfun TYPE) \bullet\\
&\quad (funTR = \emptyset \Rightarrow mapFunTransitions(funTR, tr, O) = \emptyset) \land\\
&\quad (funTR \neq \emptyset \Rightarrow \exists trans : funTR ; tr1 : TIOTS\_TRANSREL \bullet\\
&\qquad ((\exists trans2 : tr \bullet trans2.1 = trans.3 \land trans2.2 \in \operatorname{ran} del) \Rightarrow\\
&\qquad\quad tr1 = mapFunTransitionOut(trans, O)) \land\\
&\qquad (\neg\ (\exists trans2 : tr \bullet trans2.1 = trans.3 \land trans2.2 \in \operatorname{ran} del) \Rightarrow\\
&\qquad\quad tr1 = mapFunTransitionTau(trans)) \land\\
&\qquad mapFunTransitions(funTR, tr, O) = tr1 \cup\\
&\qquad\quad mapFunTransitions(funTR \setminus \{trans\}, tr, O))
\end{aligned}$$

$$TRANSREL\_NTD : \mathbb{P}\, TRANSREL$$

$$\begin{aligned}
&\forall tr : TRANSREL\_NTD \bullet tr \neq \emptyset \land\\
&\quad \forall trans1, trans2 : tr \bullet trans1.1 = trans2.1 \land\\
&\qquad trans1.2 \in \operatorname{ran} del \land trans2.2 \in \operatorname{ran} del \land\\
&\qquad (delayTransition(trans1.2)).1 = (delayTransition(trans2.2)).1
\end{aligned}$$

$$getNTDDelay : TRANSREL\_NTD \to DELAY$$

$$\begin{aligned}
&\forall tr : TRANSREL\_NTD \bullet \exists trans : tr \bullet\\
&\quad getNTDDelay(tr) = (delayTransition(trans.2)).1
\end{aligned}$$

$$
\begin{array}{|l}
TRANSREL\_PART : \mathbb{P}(\mathbb{P}\, TRANSREL\_NTD) \\
\hline
\forall tr : TRANSREL\_PART \bullet \forall p1, p2 : tr \bullet \\
\quad p1 \cap p2 = \emptyset \land getNTDDelay(p1) \neq getNTDDelay(p2)
\end{array}
$$

$$
\begin{array}{|l}
groupNTDDelays : TRANSREL \pfun TRANSREL\_PART \\
\hline
\mathrm{dom}\, groupNTDDelays = \{tr : TRANSREL \mid \forall trans : tr \bullet trans.2 \in \mathrm{ran}\, del\} \\
\forall tr : TRANSREL \bullet \bigcup(groupNTDDelays(tr)) = tr
\end{array}
$$

$$
\begin{array}{|l}
getStates : TIOTS\_TRANSREL \rightarrow TIOTS\_STATES\_SET \\
\hline
\forall tr : TIOTS\_TRANSREL \bullet getStates(tr) = \bigcup\{trans : tr \bullet \{trans.1, trans.3\}\}
\end{array}
$$

$$
\begin{array}{|l}
mapTDDelTransition : (TRANS \times TRANSREL \times TIOTS\_STATES\_SET \times \\
\quad (NAME \pfun TYPE) \times (NAME \pfun TYPE)) \pfun TIOTS\_TRANSREL \\
\hline
\mathrm{dom}\, mapTDDelTransition = (STATE \times \mathrm{ran}\, del \times STATE) \times TRANSREL \times \\
\quad TIOTS\_STATES\_SET \times (NAME \pfun TYPE) \times (NAME \pfun TYPE) \\
\forall trans : TRANS; \\
tr : TRANSREL; used : TIOTS\_STATES\_SET; I, O : (NAME \pfun TYPE) \mid \\
\quad trans.2 \in \mathrm{ran}\, del \bullet \exists q3 : TIOTS\_STATE; tr1 : TIOTS\_TRANSREL \bullet q3 \notin used \land \\
\quad\quad ((\exists trans2 : tr \bullet trans2.1 = trans.3 \land trans2.2 \in \mathrm{ran}\, del) \Rightarrow \\
\quad\quad\quad \exists q4 : TIOTS\_STATE \bullet q4 \notin used \cup \{q3\} \land \\
\quad\quad\quad\quad tr1 = \{(q3, in(genAction(currentValues(\mathrm{dom}\, I \lhd trans.3))), q4), \\
\quad\quad\quad\quad\quad (q4, out(genAction(currentValues(\mathrm{dom}\, O \lhd trans.3))), \\
\quad\quad\quad\quad\quad mapState(trans.3))\}) \land \\
\quad\quad (\neg\, (\exists trans2 : tr \bullet trans2.1 = trans.3 \land trans2.2 \in \mathrm{ran}\, del) \Rightarrow \\
\quad\quad\quad tr1 = \{(q3, in(genAction(currentValues(\mathrm{dom}\, I \lhd trans.3))), \\
\quad\quad\quad\quad mapState(trans.3))\}) \land \\
\quad\quad mapTDDelTransition(trans, tr, used, I, O) = tr1 \cup \{(mapState(trans.1), \\
\quad\quad\quad tiots\_del(mapDelay(delayTransition(trans.2)).1), q3)\}
\end{array}
$$

$$
\begin{array}{l}
mapTDDelTransitions : (TRANSREL \times TRANSREL \times TIOTS\_STATES\_SET \times \\
\quad (NAME \pfun TYPE) \times (NAME \pfun TYPE)) \pfun TIOTS\_TRANSREL \\
\hline
\operatorname{dom} mapTDDelTransitions = \{tr : TRANSREL \mid (\forall trans : tr \bullet trans.2 \in \operatorname{ran} del) \land \\
\quad (\forall trans1, trans2 : tr \mid trans1 \neq trans2 \bullet trans1.1 \neq trans2.1 \lor \\
\qquad (delayTransition(trans1.2)).1 \neq (delayTransition(trans2.2)).1)\} \times \\
\quad TRANSREL \times TIOTS\_STATES\_SET \times (NAME \pfun TYPE) \times (NAME \pfun TYPE) \\
\forall delTR, tr : TRANSREL; used : TIOTS\_STATES\_SET; I, O : (NAME \pfun TYPE) \bullet \\
\quad (delTR = \emptyset \Rightarrow mapTDDelTransitions(delTR, tr, used, I, O) = \emptyset) \land \\
\quad (delTR \neq \emptyset \Rightarrow \exists trans : delTR; tr1 : TIOTS\_TRANSREL \bullet \\
\qquad\qquad tr1 = mapTDDelTransition(trans, tr, used, I, O) \land \\
\qquad\qquad mapTDDelTransitions(delTR, tr, used, I, O) = tr1 \cup \\
\qquad\qquad\quad mapTDDelTransitions(delTR \setminus \{trans\}, \\
\qquad\qquad\qquad tr, used \cup getStates(tr1), I, O))
\end{array}
$$

$$
\begin{array}{l}
finite\_partition : (\mathbb{P}\, TRANSREL\_NTD) \\
\hline
\forall part : TRANSREL\_NTD \bullet part \in finite\_partition \Leftrightarrow \\
(\exists n : \mathbb{N} \bullet \exists f : 1..n \rightarrowtail\!\!\!\!\rightarrow part \bullet true)
\end{array}
$$

$$
\begin{array}{l}
\mathit{mapNTDDelTargetStates} : (\mathit{TRANSREL\_NTD} \times \mathit{TRANSREL} \times \mathit{TIOTS\_STATE} \times \\
\quad \mathit{TIOTS\_STATES\_SET} \times (\mathit{NAME} \pfun \mathit{TYPE}) \times (\mathit{NAME} \pfun \mathit{TYPE})) \pfun \\
\quad \mathit{TIOTS\_TRANSREL} \\
\hline
\mathrm{dom}\,\mathit{mapNTDDelTargetStates} = \{\mathit{part} : \mathit{TRANSREL\_NTD} \mid \\
\qquad (\mathit{part} \in \mathit{finite\_partition} \land \#\mathit{part} > 1) \lor \mathit{part} \notin \mathit{finite\_partition}\} \times \\
\quad \mathit{TRANSREL} \times \mathit{TIOTS\_STATE} \times \mathit{TIOTS\_STATES\_SET} \times \\
\quad (\mathit{NAME} \pfun \mathit{TYPE}) \times (\mathit{NAME} \pfun \mathit{TYPE}) \\
\forall \mathit{part} : \mathit{TRANSREL\_NTD}; \mathit{tr} : \mathit{TRANSREL}; q3 : \mathit{TIOTS\_STATE}; \\
\mathit{used} : \mathit{TIOTS\_STATES\_SET}; I, O : (\mathit{NAME} \pfun \mathit{TYPE}) \bullet \\
\quad (\mathit{part} = \emptyset \Rightarrow \mathit{mapNTDDelTargetStates}(\mathit{part}, \mathit{tr}, q3, \mathit{used}, I, O) = \emptyset) \land \\
\quad (\mathit{part} \neq \emptyset \Rightarrow \exists \mathit{trans} : \mathit{part}; \mathit{tr}1 : \mathit{TIOTS\_TRANSREL} \bullet \\
\qquad ((\exists \mathit{trans}2 : \mathit{tr} \bullet \mathit{trans}2.1 = \mathit{trans}.3 \land \mathit{trans}2.2 \in \mathrm{ran}\,\mathit{del}) \Rightarrow \\
\qquad\quad \exists q4 : \mathit{TIOTS\_STATE} \bullet q4 \notin \mathit{used} \cup \{q3\} \land \\
\qquad\qquad \mathit{tr}1 = \{(q3, \mathit{in}(\mathit{genAction}(\mathit{currentValues}(\mathrm{dom}\,I \lhd \mathit{trans}.3))), q4), \\
\qquad\qquad\quad (q4, \mathit{out}(\mathit{genAction}(\mathit{currentValues}(\mathrm{dom}\,O \lhd \mathit{trans}.3))), \\
\qquad\qquad\quad \mathit{mapState}(\mathit{trans}.3))\}) \land \\
\qquad (\lnot\, (\exists \mathit{trans}2 : \mathit{tr} \bullet \mathit{trans}2.1 = \mathit{trans}.3 \land \mathit{trans}2.2 \in \mathrm{ran}\,\mathit{del}) \Rightarrow \\
\qquad\quad \mathit{tr}1 = \{(q3, \mathit{in}(\mathit{genAction}(\mathit{currentValues}(\mathrm{dom}\,I \lhd \mathit{trans}.3))), \\
\qquad\quad \mathit{mapState}(\mathit{trans}.3))\}) \land \\
\qquad\quad \mathit{mapNTDDelTargetStates}(\mathit{part}, \mathit{tr}, q3, \mathit{used}, I, O) = \mathit{tr}1 \cup \\
\qquad\qquad \mathit{mapNTDDelTargetStates}(\mathit{part} \setminus \{\mathit{trans}\}, \mathit{tr}, q3, \\
\qquad\qquad\quad \mathit{used} \cup \mathit{getStates}(\mathit{tr}1), I, O))
\end{array}
$$

$$
\begin{array}{l}
\mathit{mapNTDDelTransitions} : (\mathit{TRANSREL\_NTD} \times \mathit{TRANSREL} \times \mathit{TIOTS\_STATES\_SET} \times \\
\quad (\mathit{NAME} \pfun \mathit{TYPE}) \times (\mathit{NAME} \pfun \mathit{TYPE})) \pfun \mathit{TIOTS\_TRANSREL} \\
\hline
\mathrm{dom}\,\mathit{mapNTDDelTransitions} = \{\mathit{part} : \mathit{TRANSREL\_NTD} \mid \\
\qquad (\mathit{part} \in \mathit{finite\_partition} \land \#\mathit{part} > 1) \lor \mathit{part} \notin \mathit{finite\_partition}\} \\
\quad \times \mathit{TRANSREL} \times \mathit{TIOTS\_STATES\_SET} \times \\
\quad (\mathit{NAME} \pfun \mathit{TYPE}) \times (\mathit{NAME} \pfun \mathit{TYPE}) \\
\forall \mathit{part} : \mathit{TRANSREL\_NTD}; \mathit{tr} : \mathit{TRANSREL}; \\
\mathit{used} : \mathit{TIOTS\_STATES\_SET}; I, O : (\mathit{NAME} \pfun \mathit{TYPE}) \bullet \\
\quad \exists q3 : \mathit{TIOTS\_STATE}; \mathit{trans} : \mathit{part} \mid q3 \notin \mathit{used} \bullet \\
\qquad \mathit{mapNTDDelTransitions}(\mathit{part}, \mathit{tr}, \mathit{used}, I, O) = \\
\qquad\quad \mathit{mapNTDDelTargetStates}(\mathit{part}, \mathit{tr}, q3, \mathit{used} \cup \{q3\}, I, O) \cup \\
\qquad\quad \{(\mathit{mapState}(\mathit{trans}.1), \mathit{tiots\_del}(\mathit{mapDelay}(\mathit{delayTransition}(\mathit{trans}.2)).1), \\
\qquad\qquad q3)\}
\end{array}
$$

$$
\begin{array}{l}
mapSetOfNTDDelTransitions : (TRANSREL\_PART \times TRANSREL \times \\
\quad TIOTS\_STATES\_SET \times (NAME \pfun TYPE) \times (NAME \pfun TYPE)) \pfun \\
\quad TIOTS\_TRANSREL \\
\hline
\mathrm{dom}\, mapSetOfNTDDelTransitions = \{trs : TRANSREL\_PART \mid \forall part : trs \bullet \\
\quad (part \in finite\_partition \land \# part > 1) \lor part \notin finite\_partition\} \\
\quad \times TRANSREL \times TIOTS\_STATES\_SET \times (NAME \pfun TYPE) \times (NAME \pfun TYPE) \\
\forall trs : TRANSREL\_PART ; trB : TRANSREL; \\
used : TIOTS\_STATES\_SET ; I, O : (NAME \pfun TYPE) \bullet \\
\quad (trs = \emptyset \Rightarrow mapSetOfNTDDelTransitions(trs, trB, used, I, O) = \emptyset) \land \\
\quad (trs \neq \emptyset \Rightarrow \exists trA : trs ; tr1 : TIOTS\_TRANSREL \bullet \\
\qquad tr1 = mapNTDDelTransitions(trA, trB, used, I, O) \land \\
\qquad mapSetOfNTDDelTransitions(trs, trB, used, I, O) = tr1 \cup \\
\qquad\quad mapSetOfNTDDelTransitions(trs \setminus \{trA\}, trB, used \cup getStates(tr1), \\
\qquad\qquad I, O))
\end{array}
$$

$$
\begin{array}{l}
getTDDelays : TRANSREL\_PART \rightarrow \mathbb{P}\, TRANSREL \\
\hline
\forall trs : TRANSREL\_PART \bullet getTDDelays(trs) = \\
\quad \{part : trs \mid part \in finite\_partition \land \# part = 1\}
\end{array}
$$

$$
\begin{array}{l}
mapDelTransitions : (TRANSREL \times TRANSREL \times TIOTS\_STATES\_SET \times \\
\quad (NAME \pfun TYPE) \times (NAME \pfun TYPE)) \pfun TIOTS\_TRANSREL \\
\hline
\mathrm{dom}\, mapDelTransitions = \{tr : TRANSREL \mid \forall trans : tr \bullet trans.2 \in \mathrm{ran}\, del\} \times \\
\quad TRANSREL \times TIOTS\_STATES\_SET \times (NAME \pfun TYPE) \times (NAME \pfun TYPE) \\
\forall delTR, tr : TRANSREL; used : TIOTS\_STATES\_SET ; I, O : (NAME \pfun TYPE) \bullet \\
\quad \exists tr1, tr2 : TIOTS\_TRANSREL ; parts : TRANSREL\_PART \bullet \\
\qquad parts = groupNTDDelays(delTR) \land \\
\qquad tr1 = mapTDDelTransitions(\bigcup(getTDDelays(parts)), tr, used, I, O) \land \\
\qquad tr2 = mapSetOfNTDDelTransitions(parts \setminus getTDDelays(parts), tr, \\
\qquad\quad used \cup getStates(tr1), I, O) \land \\
\qquad mapDelTransitions(delTR, tr, used, I, O) = tr1 \cup tr2
\end{array}
$$

$$
\begin{array}{l}
getTransitions : (TRANSREL \times (\mathbb{P}\, TRANS\_LABEL)) \rightarrow TRANSREL \\
\hline
\forall tr : TRANSREL ; labelType : \mathbb{P}\, TRANS\_LABEL \bullet \\
\quad getTransitions(tr, labelType) = \{trans : tr \mid trans.2 \in labelType\}
\end{array}
$$

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

$$\begin{array}{l} getInputActions : TIOTS\_TRANSREL \rightarrow TIOTS\_ACTIONS \\ \hline \forall tr : TIOTS\_TRANSREL \bullet getInputActions(tr) = \\ \quad \{trans : tr \mid trans.2 \in \mathrm{ran}\, in \bullet (in^{\sim})(trans.2)\} \end{array}$$

$$\begin{array}{l} getOutputActions : TIOTS\_TRANSREL \rightarrow TIOTS\_ACTIONS \\ \hline \forall tr : TIOTS\_TRANSREL \bullet getOutputActions(tr) = \\ \quad \{trans : tr \mid trans.2 \in \mathrm{ran}\, out \bullet (out^{\sim})(trans.2)\} \end{array}$$

$$\begin{array}{l} getDelays : TIOTS\_TRANSREL \rightarrow TIOTS\_DELAYS \\ \hline \forall tr : TIOTS\_TRANSREL \bullet getDelays(tr) = \\ \quad \{trans : tr \mid trans.2 \in \mathrm{ran}\, tiots\_del \bullet (tiots\_del^{\sim})(trans.2)\} \end{array}$$

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

B.6 Soundness of *fromDFRStoTIOTS*

B.6.1 Disjointness of $t.I$ and $t.O$

B.6.1.1 Transformation: implies-and to and-or

Lemma 1.

$(a \Rightarrow b) \land (\neg\, a \Rightarrow c) \Leftrightarrow (a \land b) \lor (\neg\, a \land c)$
provided
$\neg\, (b \land c)$

proof

$(a \Rightarrow b) \land (\neg\, a \Rightarrow c)$

$\Leftrightarrow$ *[propositional calculus]*

$(\neg\, a \lor b) \land (a \lor c)$

$\Leftrightarrow$ *[propositional calculus]*

$(\neg\, a \land a) \lor (\neg\, a \land c) \lor (b \land a) \lor (b \land c)$

$\Leftrightarrow$ *[propositional calculus]*

$(\neg\, a \land c) \lor (b \land a) \lor (b \land c)$

$\Leftrightarrow$ *[proviso]*

$(\neg\, a \land c) \lor (b \land a) \lor (\mathit{false})$

$\Leftrightarrow$ *[propositional calculus]*

$(a \land b) \lor (\neg\, a \land c)$

■

B.6.1.2 Inductive property: 2^{nd} template

Lemma 2.

$$\begin{array}{l}
\forall f : \mathbb{P}X \times A \times B \rightarrow\!\!\!\!\!\mapsto \mathbb{P}Z ; g : X \times B \rightarrow\!\!\!\!\!\mapsto \mathbb{P}Z ; h : X \rightarrow\!\!\!\!\!\mapsto \mathbb{P}Z ; xx : \mathbb{P}X ; a : A ; b : B ; z : Z \bullet \\
\quad z \in f(xx,a,b) \Rightarrow \exists x : xx \bullet h(x) \neq g(x,b) \wedge \\
\qquad (P(x,a) \Rightarrow z \in h(x)) \wedge (\neg\, (P(x,a)) \Rightarrow z \in g(x,b))
\end{array}$$

provided

$$\begin{array}{l}
\forall xx : \mathbb{P}X ; a : A ; b : B \bullet \\
\quad (xx = \emptyset \Rightarrow f(xx,a,b) = \emptyset) \wedge \\
\quad (xx \neq \emptyset \Rightarrow \exists x : xx \bullet h(x) \neq g(x,b) \wedge \\
\qquad (P(x,a) \Rightarrow f(xx,a,b) = h(x) \cup f(xx \setminus \{x\},a,b)) \wedge \\
\qquad (\neg\, (P(x,a)) \Rightarrow f(xx,a,b) = g(x,b) \cup f(xx \setminus \{x\},a,b)))
\end{array}$$

proof

Let xx, a, b, and z range over values in the sets $\mathbb{P}$ *X, A, B, and Z, respectively; and f, g and h be of the following types:* $\mathbb{P}\ X \times A \times B \rightarrow\!\!\!\!\!\mapsto \mathbb{P}\ Z, X \times B \rightarrow\!\!\!\!\!\mapsto \mathbb{P}\ Z$*, and* $X \rightarrow\!\!\!\!\!\mapsto \mathbb{P}\ Z$*, respectively. We prove the result by induction on the size of xx.*

Base case 1*: suppose* $\#xx = 0$

$z \in f(\emptyset,a,b)$

$\Rightarrow$ *[proviso]*

$z \in \emptyset$

$\Leftrightarrow$ *[property of sets]*

false

$\Rightarrow$ *[propositional calculus]*

$$\begin{array}{l}
\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad (P(x,a) \Rightarrow z \in h(x)) \wedge (\neg\, (P(x,a)) \Rightarrow z \in g(x,b))
\end{array}$$

■

***Inductive step**: assume the result holds for $\#xx = n$.*
We now prove that it holds for $\#xx = n+1$

$z \in f(xx,a,b)$

$\Rightarrow$ *[proviso]*

$$\begin{aligned}&\exists\, x : xx \bullet z \in f(xx,a,b) \wedge h(x) \neq g(x,b) \wedge \\ &\quad (P(x,a) \Rightarrow f(xx,a,b) = h(x) \cup f(xx \setminus \{x\},a,b)) \wedge \\ &\quad (\neg (P(x,a)) \Rightarrow f(xx,a,b) = g(x,b) \cup f(xx \setminus \{x\},a,b))\end{aligned}$$

$\Leftrightarrow$ *[Lemma 1, provided*
$\neg (f(xx,a,b) = h(x) \cup f(xx \setminus \{x\},a,b) \wedge$
$f(xx,a,b) = g(x,b) \cup f(xx \setminus \{x\},a,b))$
since $h(x) \neq g(x,b)$]

$$\begin{aligned}&\exists\, x : xx \bullet z \in f(xx,a,b) \wedge h(x) \neq g(x,b) \wedge \\ &\quad ((P(x,a) \wedge f(xx,a,b) = h(x) \cup f(xx \setminus \{x\},a,b)) \vee \\ &\quad (\neg (P(x,a)) \wedge f(xx,a,b) = g(x,b) \cup f(xx \setminus \{x\},a,b)))\end{aligned}$$

$\Rightarrow$ *[property of $\cup$]*

$$\begin{aligned}&\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\ &\quad ((P(x,a) \wedge (z \in h(x) \vee z \in f(xx \setminus \{x\},a,b))) \vee \\ &\quad (\neg (P(x,a)) \wedge (z \in g(x,b) \vee z \in f(xx \setminus \{x\},a,b))))\end{aligned}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{aligned}&(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\ &\quad P(x,a) \wedge (z \in h(x) \vee z \in f(xx \setminus \{x\},a,b))) \vee \\ &(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\ &\quad \neg (P(x,a)) \wedge (z \in g(x,b) \vee z \in f(xx \setminus \{x\},a,b)))\end{aligned}$$

$\Leftrightarrow$ *[propositional calculus]*

$$\begin{aligned}&(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\ &\quad ((P(x,a) \wedge z \in h(x)) \vee (P(x,a) \wedge z \in f(xx \setminus \{x\},a,b)))) \vee \\ &(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\ &\quad ((\neg (P(x,a)) \wedge z \in g(x,b)) \vee (\neg (P(x,a) \wedge z \in f(xx \setminus \{x\},a,b)))))\end{aligned}$$

$\Rightarrow$ *[propositional calculus]*

$$\begin{array}{l}
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad ((P(x,a) \wedge z \in h(x)) \vee z \in f(xx \setminus \{x\}, a, b))) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad ((\neg (P(x,a)) \wedge z \in g(x,b)) \vee z \in f(xx \setminus \{x\}, a, b)))
\end{array}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l}
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge P(x,a) \wedge z \in h(x)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge z \in f(xx \setminus \{x\}, a, b)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \neg (P(x,a)) \wedge z \in g(x,b)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge z \in f(xx \setminus \{x\}, a, b))
\end{array}$$

$\Leftrightarrow$ *[propositional calculus]*

$$\begin{array}{l}
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge P(x,a) \wedge z \in h(x)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \neg (P(x,a)) \wedge z \in g(x,b)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge z \in f(xx \setminus \{x\}, a, b))
\end{array}$$

$\Rightarrow$ *[inductive hypothesis]*

$$\begin{array}{l}
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge P(x,a) \wedge z \in h(x)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \neg (P(x,a)) \wedge z \in g(x,b)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad \exists\, x2 : xx \setminus \{x\} \bullet h(x2) \neq g(x2,b) \wedge \\
\qquad (P(x2,a) \Rightarrow z \in h(x2)) \wedge (\neg (P(x2,a)) \Rightarrow z \in g(x2,b)))
\end{array}$$

$\Rightarrow$ *[propositional calculus]*

$$\begin{array}{l}
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge P(x,a) \wedge z \in h(x)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \neg (P(x,a)) \wedge z \in g(x,b)) \vee \\
(\exists\, x : xx \bullet \\
\quad \exists\, x2 : xx \setminus \{x\} \bullet h(x2) \neq g(x2,b) \wedge \\
\qquad (P(x2,a) \Rightarrow z \in h(x2)) \wedge (\neg (P(x2,a)) \Rightarrow z \in g(x2,b)))
\end{array}$$

$\Rightarrow$ *[property of sets]*

$$\begin{array}{l}
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge P(x,a) \wedge z \in h(x)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \neg\,(P(x,a)) \wedge z \in g(x,b)) \vee \\
(\exists\, x : xx \bullet \\
\quad\quad \exists\, x2 : xx \bullet h(x2) \neq g(x2,b) \wedge \\
\quad\quad\quad\quad (P(x2,a) \Rightarrow z \in h(x2)) \wedge (\neg\,(P(x2,a)) \Rightarrow z \in g(x2,b)))
\end{array}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l}
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge P(x,a) \wedge z \in h(x)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \neg\,(P(x,a)) \wedge z \in g(x,b)) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad\quad (P(x,a) \Rightarrow z \in h(x)) \wedge (\neg\,(P(x,a)) \Rightarrow z \in g(x,b)))
\end{array}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l}
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad\quad ((P(x,a) \wedge z \in h(x)) \vee (\neg\,(P(x,a)) \wedge z \in g(x,b)))) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad\quad (P(x,a) \Rightarrow z \in h(x)) \wedge (\neg\,(P(x,a)) \Rightarrow z \in g(x,b)))
\end{array}$$

$\Leftrightarrow$ *[Lemma 1, provided $h(x) \neq g(x,b)$]*

$$\begin{array}{l}
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad\quad (P(x,a) \Rightarrow z \in h(x)) \wedge (\neg\,(P(x,a)) \Rightarrow z \in g(x,b))) \vee \\
(\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad\quad (P(x,a) \Rightarrow z \in h(x)) \wedge (\neg\,(P(x,a)) \Rightarrow z \in g(x,b)))
\end{array}$$

$\Leftrightarrow$ *[propositional calculus]*

$$\begin{array}{l}
\exists\, x : xx \bullet h(x) \neq g(x,b) \wedge \\
\quad\quad (P(x,a) \Rightarrow z \in h(x)) \wedge (\neg\,(P(x,a)) \Rightarrow z \in g(x,b))
\end{array}$$

■

B.6.1.3 Inductive property: 3rd template

Lemma 3.

$$\begin{aligned}
&\forall f : \mathbb{P}X \times A \times B \times C \times D \pfun \mathbb{P}Z;\\
&g : X \times A \times B \times C \times D \pfun \mathbb{P}Z; xx : \mathbb{P}X; a : A; b : B; c : C; d : D; z : Z \bullet\\
&\quad z \in f(xx,a,b,c,d) \Rightarrow (\exists x : xx; b2 : B \bullet z \in g(x,a,b2,c,d))\\
&\textit{provided}\\
&\forall xx : \mathbb{P}X; a : A; b : B; c : C; d : D \bullet\\
&\quad (xx = \emptyset \Rightarrow f(xx,a,b,c,d) = \emptyset) \land\\
&\quad (xx \neq \emptyset \Rightarrow \exists x : xx; b2 : B \bullet\\
&\qquad f(xx,a,b,c,d) = g(x,a,b,c,d) \cup f(xx \setminus \{x\},a,b2,c,d))
\end{aligned}$$

proof

Let xx, a, b, c, d and z range over values in the sets $\mathbb{P}$ *X, A, B, C, D, and Z, respectively; and f, and g be of the following types:* $\mathbb{P}\ X \times A \times B \times C \times D \pfun \mathbb{P}\ Z, X \times A \times B \times C \times D \pfun \mathbb{P}\ Z$*, respectively. We prove the result by induction on the size of xx.*

***Base case 1**: suppose* $\#xx = 0$

$z \in f(\emptyset,a,b,c,d)$

$\Rightarrow$ *[proviso]*

$z \in \emptyset$

$\Leftrightarrow$ *[property of sets]*

false

$\Rightarrow$ *[propositional calculus]*

$\exists\ x : xx;\ b2 : B \bullet z \in g(x,a,b2,c,d)$

■

Inductive step*: assume the result holds for* $\#xx = n$.
We now prove that it holds for $\#xx = n+1$

$z \in f(xx,a,b,c,d)$

$\Rightarrow$ *[proviso]*

$$\exists\, x : xx;\ b2 : B \bullet z \in f(xx,a,b,c,d) \wedge$$
$$f(xx,a,b,c,d) = g(x,a,b,c,d) \cup f(xx \setminus \{x\},a,b2,c,d)$$

$\Rightarrow$ *[property of* $\cup$*]*

$$\exists\, x : xx;\ b2 : B \bullet$$
$$z \in g(x,a,b,c,d) \vee z \in f(xx \setminus \{x\},a,b2,c,d)$$

$\Leftrightarrow$ *[predicate calculus]*

$$(\exists\, x : xx \bullet z \in g(x,a,b,c,d)) \vee$$
$$(\exists\, x : xx;\ b2 : B \bullet z \in f(xx \setminus \{x\},a,b2,c,d))$$

$\Rightarrow$ *[inductive hypothesis]*

$$(\exists\, x : xx \bullet z \in g(x,a,b,c,d)) \vee$$
$$(\exists\, x : xx;\ b2 : B \bullet (\exists\, x2 : xx \setminus \{x\};\ b3 : B \bullet z \in g(x2,a,b3,c,d)))$$

$\Rightarrow$ *[property of sets]*

$$(\exists\, x : xx \bullet z \in g(x,a,b,c,d)) \vee$$
$$(\exists\, x : xx;\ b2 : B \bullet (\exists\, x2 : xx;\ b3 : B \bullet z \in g(x2,a,b3,c,d)))$$

$\Leftrightarrow$ *[predicate calculus]*

$$(\exists\, x : xx;\ b2 : B \bullet z \in g(x,a,b2,c,d) \wedge b2 = b) \vee$$
$$(\exists\, x : xx;\ b2 : B \bullet z \in g(x,a,b2,c,d))$$

$\Rightarrow$ *[propositional calculus]*

$$(\exists\, x : xx;\ b2 : B \bullet z \in g(x,a,b2,c,d)) \vee$$
$$(\exists\, x : xx;\ b2 : B \bullet z \in g(x,a,b2,c,d))$$

$\Leftrightarrow$ *[propositional calculus]*

$\exists\, x : xx;\ b2 : B \bullet z \in g(x, a, b2, c, d)$

■

B.6.1.4 Mapping function transitions

Lemma 4.

$$\begin{array}{l}
\forall t : TIOTS \bullet \forall d : e\_DFRS \bullet \forall trans : t.T \bullet \\
\quad trans \in mapFunTransitions(getTransitions(d.TR, \operatorname{ran} fun), d.TR, d.O) \Rightarrow \\
\quad ((trans.2 \in \operatorname{ran} out \wedge \exists trans2 : d.TR \mid trans2.2 \in \operatorname{ran} fun \bullet \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \vee \\
\quad (trans.2 \in \operatorname{ran} tau))
\end{array}$$

proof

Let t, d, and trans range over values in the sets TIOTS, e_DFRS, and TIOTS_TRANS, respectively.

$trans \in mapFunTransitions(getTransitions(d.TR, \operatorname{ran} fun), d.TR, d.O)$

$\Rightarrow$ *[Lemma 2*

$f \equiv mapFunctionTransitions,\ g \equiv mapFunTransitionOut,\ h \equiv mapFunTransitionTau,$
$xx \equiv funTR,\ X \equiv TRANS,\ a \equiv tr,\ A \equiv TRANSREL$
$b \equiv O,\ B \equiv (NAME \nrightarrow TYPE),\ z \equiv trans,\ Z \equiv TIOTS\_TRANS]$

$$\begin{array}{l}
\exists\, trans2 : getTransitions(d.TR, \operatorname{ran} fun) \bullet \\
\quad ((\exists\, transA : d.TR \bullet transA.1 = trans2.3 \wedge transA.2 \in \operatorname{ran} del) \Rightarrow \\
\qquad trans \in mapFunTransitionOut(trans2, d.O)) \wedge \\
\quad (\neg\, (\exists\, transA : d.TR \bullet transA.1 = trans2.3 \wedge transA.2 \in \operatorname{ran} del) \Rightarrow \\
\qquad trans \in mapFunTransitionTau(trans2))
\end{array}$$

$\Rightarrow$ *[definition of mapFunTransitionOut, and mapFunTransitionTau]*

$$\exists\, trans2 : getTransitions(d.TR, \operatorname{ran} fun) \bullet$$
$$((\exists\, transA : d.TR \bullet transA.1 = trans2.3 \wedge transA.2 \in \operatorname{ran} del) \Rightarrow$$
$$trans \in$$
$$\{(mapState(trans2.1),$$
$$out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))),$$
$$mapState(trans2.3))\}) \wedge$$
$$(\neg\, (\exists\, transA : d.TR \bullet transA.1 = trans2.3 \wedge transA.2 \in \operatorname{ran} del) \Rightarrow$$
$$trans \in \{(mapState(trans2.1), tau, mapState(trans2.3))\})$$

$\Rightarrow$ *[property of sets]*

$$\exists\, trans2 : getTransitions(d.TR, \operatorname{ran} fun) \bullet$$
$$((\exists\, transA : d.TR \bullet transA.1 = trans2.3 \wedge transA.2 \in \operatorname{ran} del) \Rightarrow$$
$$trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \wedge$$
$$(\neg\, (\exists\, transA : d.TR \bullet transA.1 = trans2.3 \wedge transA.2 \in \operatorname{ran} del) \Rightarrow$$
$$trans.2 = tau)$$

$\Leftrightarrow$ *[Lemma 1, provided*
$\neg\, (trans.2 = out(\_) \wedge trans.2 = tau)$
due to the definition of TIOTS_TRANS_LABEL]

$$\exists\, trans2 : getTransitions(d.TR, \operatorname{ran} fun) \bullet$$
$$((\exists\, transA : d.TR \bullet transA.1 = trans2.3 \wedge transA.2 \in \operatorname{ran} del) \wedge$$
$$trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \vee$$
$$(\neg\, (\exists\, transA : d.TR \bullet transA.1 = trans2.3 \wedge transA.2 \in \operatorname{ran} del) \wedge$$
$$trans.2 = tau)$$

$\Rightarrow$ *[propositional calculus]*

$$\exists\, trans2 : getTransitions(d.TR, \operatorname{ran} fun) \bullet$$
$$trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \vee$$
$$trans.2 = tau$$

$\Leftrightarrow$ *[predicate calculus]*

$(\exists\ trans2 : getTransitions(d.TR, \text{ran } fun) \bullet$
$\quad trans.2 = out(genAction(currentValues(\text{dom } d.O \lhd trans2.3)))) \lor$
$(trans.2 = tau)$

$\Leftrightarrow$ *[property of functions]*

$(\exists\ trans2 : getTransitions(d.TR, \text{ran } fun) \bullet trans.2 \in \text{ran } out \land$
$\quad trans.2 = out(genAction(currentValues(\text{dom } d.O \lhd trans2.3)))) \lor$
$(trans.2 \in \text{ran } tau)$

$\Leftrightarrow$ *[predicate calculus]*

$(trans.2 \in \text{ran } out \land$
$\quad \exists\ trans2 : getTransitions(d.TR, \text{ran } fun) \bullet$
$\quad\quad trans.2 = out(genAction(currentValues(\text{dom } d.O \lhd trans2.3)))) \lor$
$(trans.2 \in \text{ran } tau)$

$\Leftrightarrow$ *[definition of getTransitions]*

$(trans.2 \in \text{ran } out \land$
$\quad \exists\ trans2 : TRANSREL \mid trans2 \in \{ transA : d.TR \mid transA.2 \in \text{ran } fun \} \bullet$
$\quad\quad trans.2 = out(genAction(currentValues(\text{dom } d.O \lhd trans2.3)))) \lor$
$(trans.2 \in \text{ran } tau)$

$\Leftrightarrow$ *[definition of set comprehension]*

$(trans.2 \in \text{ran } out \land$
$\quad \exists\ trans2 : TRANSREL \mid (\exists\ transA : TRANSREL \bullet transA \in d.TR \land$
$\quad\quad transA.2 \in \text{ran } fun \land trans2 = transA) \bullet$
$\quad\quad\quad trans.2 = out(genAction(currentValues(\text{dom } d.O \lhd trans2.3)))) \lor$
$(trans.2 \in \text{ran } tau)$

$\Leftrightarrow$ *[predicate calculus – one-point rule]*

$(trans.2 \in \text{ran } out \land$
$\quad \exists\ trans2 : TRANSREL \mid trans2 \in d.TR \land trans2.2 \in \text{ran } fun \bullet$
$\quad\quad trans.2 = out(genAction(currentValues(\text{dom } d.O \lhd trans2.3)))) \lor$
$(trans.2 \in \text{ran } tau)$

$\Leftrightarrow$ *[predicate calculus]*

$(trans.2 \in \mathrm{ran}\ out \wedge \exists\, trans2 : d.TR \mid trans2.2 \in \mathrm{ran}\ fun \bullet$
$\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\ d.O \lhd trans2.3)))) \vee$
$(trans.2 \in \mathrm{ran}\ tau)$

■

B.6.1.5 Transformation: implies-or-and to or

Lemma 5.

$((a \Rightarrow b \vee c) \wedge (\neg\, a \Rightarrow b)) \Rightarrow (b \vee c)$
provided
$\neg\,(b \wedge c)$

proof

$(a \Rightarrow b \vee c) \wedge (\neg\, a \Rightarrow b)$

$\Leftrightarrow$ *[propositional calculus]*

$((a \Rightarrow b) \vee (a \Rightarrow c)) \wedge (\neg\, a \Rightarrow b)$

$\Leftrightarrow$ *[propositional calculus]*

$((a \Rightarrow b) \wedge (\neg\, a \Rightarrow b)) \vee ((a \Rightarrow c) \wedge (\neg\, a \Rightarrow b))$

$\Leftrightarrow$ *[propositional calculus]*

$((\neg\, a \vee b) \wedge (a \vee b)) \vee ((a \Rightarrow c) \wedge (\neg\, a \Rightarrow b))$

$\Leftrightarrow$ *[propositional calculus]*

$((\neg\, a \wedge b) \vee (b \wedge a) \vee b) \vee ((a \Rightarrow c) \wedge (\neg\, a \Rightarrow b))$

$\Leftrightarrow$ *[propositional calculus]*

$(((\neg\, a \vee a) \wedge b) \vee b) \vee ((a \Rightarrow c) \wedge (\neg\, a \Rightarrow b))$

$\Leftrightarrow$ *[propositional calculus]*

$b \lor ((a \Rightarrow c) \land (\neg\, a \Rightarrow b))$

$\Leftrightarrow$ *[Lemma 1, provided $\neg\,(b \land c)$]*

$b \lor (a \land c) \lor (\neg\, a \land b)$

$\Rightarrow$ *[propositional calculus]*

$b \lor c \lor b$

$\Leftrightarrow$ *[propositional calculus]*

$b \lor c$

■

B.6.1.6 Mapping delay transitions

Lemma 6.

$$\begin{aligned}
&\forall t : TIOTS \bullet \forall d : e\_DFRS \bullet \forall trans : t.T \bullet\\
&\quad trans \in mapDelTransitions(\\
&\qquad getTransitions(d.TR, \operatorname{ran} del), d.TR, \operatorname{ran} mapState, d.I, d.O) \Rightarrow\\
&\quad ((trans.2 \in \operatorname{ran} in \land \exists trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet\\
&\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3)))) \lor\\
&\quad (trans.2 \in \operatorname{ran} tiots\_del \land \exists trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet\\
&\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor\\
&\quad (trans.2 \in \operatorname{ran} out \land \exists trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet\\
&\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))))
\end{aligned}$$

proof
Let t, d, and trans range over values in the sets TIOTS, e_DFRS, and TIOTS_TRANS, respectively.

$trans \in mapDelTransitions(getTransitions(d.TR, \operatorname{ran} del),$
$\quad d.TR, \operatorname{ran} mapState, d.I, d.O)$

$\Rightarrow$ *[definition of mapDelTransitions]*

$$\begin{array}{l}
\exists\, tr1,\ tr2 : \mathit{TIOTS\_TRANSREL};\ parts : \mathit{TRANSREL\_PART} \bullet \\
\quad parts = groupNTDDelays(getTransitions(d.TR,\ \mathrm{ran}\ del)) \wedge \\
\quad tr1 = mapTDDelTransitions(\\
\qquad \bigcup(getTDDelays(parts)),\ d.TR,\ \mathrm{ran}\ mapState,\ d.I,\ d.O) \wedge \\
\quad tr2 = mapSetOfNTDDelTransitions(parts \setminus getTDDelays(parts),\ d.TR, \\
\qquad \mathrm{ran}\ mapState \cup getStates(tr1),\ d.I,\ d.O) \wedge \\
\quad mapDelTransitions(getTransitions(d.TR,\ \mathrm{ran}\ del),\ d.TR, \\
\qquad \mathrm{ran}\ mapState,\ d.I,\ d.O) = tr1 \cup tr2 \wedge \\
\quad trans \in mapDelTransitions(getTransitions(d.TR,\ \mathrm{ran}\ del), \\
\qquad d.TR,\ \mathrm{ran}\ mapState,\ d.I,\ d.O)
\end{array}$$

$\Rightarrow$ *[property of sets]*

$$\begin{array}{l}
\exists\, tr1 : \mathit{TIOTS\_TRANSREL};\ parts : \mathit{TRANSREL\_PART} \mid \\
\quad parts = groupNTDDelays(getTransitions(d.TR,\ \mathrm{ran}\ del)) \bullet \\
\qquad trans \in mapTDDelTransitions(\bigcup(getTDDelays(parts)),\ d.TR, \\
\qquad\quad \mathrm{ran}\ mapState,\ d.I,\ d.O) \vee \\
\qquad trans \in mapSetOfNTDDelTransitions(parts \setminus getTDDelays(parts),\ d.TR, \\
\qquad\quad \mathrm{ran}\ mapState \cup getStates(tr1),\ d.I,\ d.O)
\end{array}$$

$\Rightarrow$ *[Lemma 3*
$f \equiv mapTDDelTransitions,\ g \equiv mapTDDelTransition$
$xx \equiv delTR,\ X \equiv TRANS,\ a \equiv tr,\ A \equiv TRANSREL$
$b \equiv used,\ B \equiv TIOTS\_STATES\_SET,\ c \equiv I,\ d \equiv O]$
$C, D \equiv (NAME \nrightarrow TYPE),\ z \equiv trans,\ Z \equiv TIOTS\_TRANS]$

$$\begin{array}{l}
\exists\, tr1 : \mathit{TIOTS\_TRANSREL};\ parts : \mathit{TRANSREL\_PART} \mid \\
\quad parts = groupNTDDelays(getTransitions(d.TR,\ \mathrm{ran}\ del)) \bullet \\
\qquad (\exists\, trans2 : \bigcup(getTDDelays(parts));\ used2 : \mathit{TIOTS\_STATES\_SET} \bullet \\
\qquad\quad trans \in mapTDDelTransition(trans2,\ d.TR,\ used2,\ d.I,\ d.O)) \vee \\
\qquad trans \in mapSetOfNTDDelTransitions(parts \setminus getTDDelays(parts),\ d.TR, \\
\qquad\quad \mathrm{ran}\ mapState \cup getStates(tr1),\ d.I,\ d.O)
\end{array}$$

$\Rightarrow$ *[definition of mapTDDelTransition]*

$$\begin{array}{l}
\exists\, tr1 : TIOTS\_TRANSREL;\ parts : TRANSREL\_PART \mid \\
\quad parts = groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)) \bullet \\
\quad\quad (\exists\, trans2 : \bigcup(getTDDelays(parts)) \bullet \\
\quad\quad\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\quad\quad\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\quad\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\quad\quad trans \in mapSetOfNTDDelTransitions(parts \setminus getTDDelays(parts), d.TR, \\
\quad\quad\quad \operatorname{ran} mapState \cup getStates(tr1), d.I, d.O)
\end{array}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l}
(\exists\, trans2 : \bigcup(getTDDelays(groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)))) \bullet \\
\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
(\exists\, tr1 : TIOTS\_TRANSREL;\ parts : TRANSREL\_PART \mid \\
\quad parts = groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)) \bullet \\
\quad\quad trans \in mapSetOfNTDDelTransitions(parts \setminus getTDDelays(parts), d.TR, \\
\quad\quad\quad \operatorname{ran} mapState \cup getStates(tr1), d.I, d.O))
\end{array}$$

$\Rightarrow$ *[Lemma 3*
$f \equiv mapSetOfNTDDelTransitions$, $g \equiv mapNTDDelTransitions$
$xx \equiv trs$, $X \equiv \mathbb{P}\ TRANSREL\_NTD$, $a \equiv trB$, $A \equiv TRANSREL$
$b \equiv used$, $B \equiv TIOTS\_STATES\_SET$, $c \equiv I$, $d \equiv O]$
$C, D \equiv (NAME \nrightarrow TYPE)$, $z \equiv trans$, $Z \equiv TIOTS\_TRANS]$

$$\begin{array}{l}
(\exists\, trans2 : \bigcup(getTDDelays(groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)))) \bullet \\
\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
(\exists\, parts : TRANSREL\_PART \mid \\
parts = groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)) \bullet \\
\quad \exists\, part : parts \setminus getTDDelays(parts);\ used2 : TIOTS\_STATES\_SET \bullet \\
\quad\quad trans \in mapNTDDelTransitions(part, d.TR, used2, d.I, d.O))
\end{array}$$

$\Rightarrow$ *[definition of mapNTDDelTransitions]*

$$
\begin{array}{l}
(\exists\ trans2 : \bigcup(getTDDelays(groupNTDDelays(getTransitions(d.TR,\ \mathrm{ran}\ del)))) \bullet \\
\quad trans.2\ =\ in(genAction(currentValues(\mathrm{dom}\ d.I \lhd trans2.3)))\ \vee \\
\quad trans.2\ =\ out(genAction(currentValues(\mathrm{dom}\ d.O \lhd trans2.3)))\ \vee \\
\quad trans.2\ =\ tiots\_del(mapDelay(delayTransition(trans2.2)).1))\ \vee \\
(\exists\ parts : TRANSREL\_PART\ \mid \\
parts\ =\ groupNTDDelays(getTransitions(d.TR,\ \mathrm{ran}\ del)) \bullet \\
\quad \exists\ part : parts \setminus getTDDelays(parts);\ used2 : TIOTS\_STATES\_SET \bullet \\
\qquad \exists\ q3 : TIOTS\_STATE;\ trans2 : part\ \mid\ q3 \notin used2 \bullet \\
\qquad\quad trans\ \in\ mapNTDDelTargetStates(part,\ d.TR,\ q3, \\
\qquad\qquad used2 \cup \{q3\},\ d.I,\ d.O)\ \vee \\
\qquad\quad trans\ \in\ \{(mapState(trans2.1), \\
\qquad\qquad tiots\_del(mapDelay(delayTransition(trans2.2)).1),\ q3)\})
\end{array}
$$

$\Rightarrow$ *[Similarly to Lemma 2, $f\ \equiv\ mapNTDDelTargetStates$]*

$$
\begin{array}{l}
(\exists\ trans2 : \bigcup(getTDDelays(groupNTDDelays(getTransitions(d.TR,\ \mathrm{ran}\ del)))) \bullet \\
\quad trans.2\ =\ in(genAction(currentValues(\mathrm{dom}\ d.I \lhd trans2.3)))\ \vee \\
\quad trans.2\ =\ out(genAction(currentValues(\mathrm{dom}\ d.O \lhd trans2.3)))\ \vee \\
\quad trans.2\ =\ tiots\_del(mapDelay(delayTransition(trans2.2)).1))\ \vee \\
(\exists\ parts : TRANSREL\_PART\ \mid \\
parts\ =\ groupNTDDelays(getTransitions(d.TR,\ \mathrm{ran}\ del)) \bullet \\
\quad \exists\ part : parts \setminus getTDDelays(parts);\ used2 : TIOTS\_STATES\_SET \bullet \\
\qquad \exists\ q3 : TIOTS\_STATE;\ trans2 : part\ \mid\ q3 \notin used2 \bullet \\
\qquad\quad (((\exists\ transA : d.TR \bullet transA.1\ =\ trans2.3\ \wedge\ transA.2\ \in\ \mathrm{ran}\ del)\ \Rightarrow \\
\qquad\qquad \exists\ q4 : TIOTS\_STATE \bullet q4 \notin used2 \cup \{q3\}\ \wedge\ trans\ \in \\
\qquad\qquad\quad \{(q3, in(genAction(currentValues(\mathrm{dom}\ d.I \lhd trans2.3))),\ q4), \\
\qquad\qquad\quad (q4,\ out(genAction(currentValues(\mathrm{dom}\ d.O \lhd trans2.3))), \\
\qquad\qquad\qquad mapState(trans2.3))\})\ \wedge \\
\qquad\quad (\neg\ (\exists\ transA : d.TR \bullet transA.1\ =\ trans2.3\ \wedge\ transA.2\ \in\ \mathrm{ran}\ del)\ \Rightarrow \\
\qquad\qquad trans\ \in\ \{(q3,\ in(genAction(currentValues(\mathrm{dom}\ d.I \lhd trans2.3))), \\
\qquad\qquad\quad mapState(trans2.3))\}))\ \vee \\
\qquad\quad trans\ \in\ \{(mapState(trans2.1), \\
\qquad\qquad tiots\_del(mapDelay(delayTransition(trans2.2)).1),\ q3)\})
\end{array}
$$

$\Rightarrow$ *[property of sets]*

$$
\begin{array}{l}
(\exists\ trans2 : \bigcup(getTDDelays(groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)))) \bullet \\
\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
(\exists\ parts : TRANSREL\_PART \mid \\
parts = groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)) \bullet \\
\quad \exists\ part : parts \setminus getTDDelays(parts) \bullet \exists\ trans2 : part \bullet \\
\qquad (((\exists\ transA : d.TR \bullet transA.1 = trans2.3 \land transA.2 \in \operatorname{ran} del) \Rightarrow \\
\qquad\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \land \\
\qquad (\neg\ (\exists\ transA : d.TR \bullet transA.1 = trans2.3 \land transA.2 \in \operatorname{ran} del) \Rightarrow \\
\qquad\quad tranA.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{array}
$$

$\Rightarrow$ *[Lemma 5*

$a \equiv \exists\ transA : d.TR \bullet transA.1 = trans2.3 \land transA.2 \in \operatorname{ran} del,$

$b \equiv trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))),$

$c \equiv trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))$

provided $\neg\ (b \land c)$ *due to definition of TIOTS_TRANS_LABEL]*

$$
\begin{array}{l}
(\exists\ trans2 : \bigcup(getTDDelays(groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)))) \bullet \\
\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
(\exists\ parts : TRANSREL\_PART \mid \\
parts = groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)) \bullet \\
\quad \exists\ part : parts \setminus getTDDelays(parts) \bullet \exists\ trans2 : part \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{array}
$$

$\Leftrightarrow$ *[predicate calculus]*

$$
\begin{array}{l}
(\exists\, parts : TRANSREL\_PART \mid \\
parts = groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)) \bullet \\
\quad \exists\, trA : TRANSREL \mid trA = \bigcup getTDDelays(parts) \bullet \exists\, trans2 : trA \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
(\exists\, parts : TRANSREL\_PART \mid \\
parts = groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)) \bullet \\
\quad \exists\, part : parts \setminus getTDDelays(parts) \bullet \exists\, trans2 : part \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{array}
$$

$\Leftrightarrow$ *[predicate calculus]*

$$
\begin{array}{l}
\exists\, parts : TRANSREL\_PART \mid \\
parts = groupNTDDelays(getTransitions(d.TR, \operatorname{ran} del)) \bullet \\
\quad (\exists\, trA : TRANSREL \mid trA = \bigcup getTDDelays(parts) \bullet \exists\, trans2 : trA \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\quad (\exists\, part : parts \setminus getTDDelays(parts) \bullet \exists\, trans2 : part \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{array}
$$

$\Rightarrow$ *[property of groupNTDDelays]*

$$
\begin{array}{l}
\exists\, parts : TRANSREL\_PART \mid \bigcup parts = getTransitions(d.TR, \operatorname{ran} del) \bullet \\
\quad (\exists\, trA : TRANSREL \mid trA = \bigcup getTDDelays(parts) \bullet \exists\, trans2 : trA \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\quad (\exists\, part : parts \setminus getTDDelays(parts) \bullet \exists\, trans2 : part \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{array}
$$

$\Leftrightarrow$ *[definition of getTransitions]*

$$
\begin{array}{l}
\exists\, parts : TRANSREL\_PART \mid \bigcup parts = \{\, transA : d.TR \mid transA.2 \in \mathrm{ran}\; del \,\} \bullet \\
\quad (\exists\, trA : TRANSREL \mid trA = \bigcup getTDDelays(parts) \bullet \exists\, trans2 : trA \bullet \\
\quad\quad trans.2 = in(genAction(currentValues(\mathrm{dom}\; d.I \lhd trans2.3))) \lor \\
\quad\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\; d.O \lhd trans2.3))) \lor \\
\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\quad (\exists\, part : parts \setminus getTDDelays(parts) \bullet \exists\, trans2 : part \bullet \\
\quad\quad trans.2 = in(genAction(currentValues(\mathrm{dom}\; d.I \lhd trans2.3))) \lor \\
\quad\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\; d.O \lhd trans2.3))) \lor \\
\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{array}
$$

$\Rightarrow$ *[property of sets]*

$$
\begin{array}{l}
\exists\, parts : TRANSREL\_PART \bullet \\
\quad (\forall\, part : parts \bullet \forall\, transA : part \bullet transA \in d.TR \land transA.2 \in \mathrm{ran}\; del) \land \\
\quad (\exists\, trA : TRANSREL \mid trA = \bigcup getTDDelays(parts) \bullet \exists\, trans2 : trA \bullet \\
\quad\quad trans.2 = in(genAction(currentValues(\mathrm{dom}\; d.I \lhd trans2.3))) \lor \\
\quad\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\; d.O \lhd trans2.3))) \lor \\
\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\quad (\exists\, part : parts \bullet \exists\, trans2 : part \bullet \\
\quad\quad trans.2 = in(genAction(currentValues(\mathrm{dom}\; d.I \lhd trans2.3))) \lor \\
\quad\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\; d.O \lhd trans2.3))) \lor \\
\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{array}
$$

$\Leftrightarrow$ *[definition of getTDDelays]*

$$
\begin{array}{l}
\exists\, parts : TRANSREL\_PART \bullet \\
\quad (\forall\, part : parts \bullet \forall\, transA : part \bullet transA \in d.TR \land transA.2 \in \mathrm{ran}\; del) \land \\
\quad (\exists\, trA : TRANSREL \mid \\
\quad trA = \bigcup \{\, part : parts \mid part \in finite\_partition \land \#part = 1 \,\} \bullet \\
\quad\quad \exists\, trans2 : trA \bullet \\
\quad\quad\quad trans.2 = in(genAction(currentValues(\mathrm{dom}\; d.I \lhd trans2.3))) \lor \\
\quad\quad\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\; d.O \lhd trans2.3))) \lor \\
\quad\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\quad (\exists\, part : parts \bullet \exists\, trans2 : part \bullet \\
\quad\quad trans.2 = in(genAction(currentValues(\mathrm{dom}\; d.I \lhd trans2.3))) \lor \\
\quad\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\; d.O \lhd trans2.3))) \lor \\
\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{array}
$$

$\Rightarrow$ *[property of sets]*

$$\begin{array}{l}
\exists\, parts : TRANSREL\_PART \bullet \\
\quad (\forall\, part : parts \bullet \forall\, transA : part \bullet transA \in d.TR \land transA.2 \in \operatorname{ran} del) \land \\
\quad (\exists\, trA : TRANSREL \bullet \\
\qquad (\forall\, transA : trA \bullet transA \in d.TR \land transA.2 \in \operatorname{ran} del) \land \\
\qquad \exists\, trans2 : trA \bullet \\
\qquad\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\quad (\exists\, part : parts \bullet \exists\, trans2 : part \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{array}$$

$\Rightarrow$ *[property of sets]*

$$\begin{array}{l}
\exists\, parts : TRANSREL\_PART \bullet \\
\quad (\forall\, part : parts \bullet \forall\, transA : part \bullet transA \in d.TR \land transA.2 \in \operatorname{ran} del) \land \\
\quad (\exists\, trA : TRANSREL \bullet \\
\qquad (\forall\, transA : trA \bullet transA \in d.TR \land transA.2 \in \operatorname{ran} del) \land \\
\qquad \exists\, trans2 : d.TR \bullet trans2.2 \in \operatorname{ran} del \land \\
\qquad\quad (trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))) \lor \\
\quad (\exists\, trans2 : d.TR \bullet trans2.2 \in \operatorname{ran} del \land \\
\qquad (trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)))
\end{array}$$

$\Rightarrow$ *[predicate calculus]*

$$\begin{array}{l}
(\exists\, trans2 : d.TR \bullet trans2.2 \in \operatorname{ran} del \land \\
\quad (trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))) \lor \\
(\exists\, trans2 : d.TR \bullet trans2.2 \in \operatorname{ran} del \land \\
\quad (trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3))) \lor \\
\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))) \lor \\
\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)))
\end{array}$$

$\Leftrightarrow$ *[propositional calculus]*

$$\begin{aligned}
&\exists\, trans2 : d.TR \bullet trans2.2 \in \mathrm{ran}\ del \wedge \\
&\quad (trans.2 = in(genAction(currentValues(\mathrm{dom}\ d.I \lhd trans2.3))) \vee \\
&\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\ d.O \lhd trans2.3))) \vee \\
&\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{aligned}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{aligned}
&(\exists\, trans2 : d.TR \bullet trans2.2 \in \mathrm{ran}\ del \wedge \\
&\quad trans.2 = in(genAction(currentValues(\mathrm{dom}\ d.I \lhd trans2.3)))) \vee \\
&(\exists\, trans2 : d.TR \bullet trans2.2 \in \mathrm{ran}\ del \wedge \\
&\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\ d.O \lhd trans2.3)))) \vee \\
&(\exists\, trans2 : d.TR \bullet trans2.2 \in \mathrm{ran}\ del \wedge \\
&\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1))
\end{aligned}$$

$\Leftrightarrow$ *[property of functions]*

$$\begin{aligned}
&(trans.2 \in \mathrm{ran}\ in \wedge \exists\, trans2 : d.TR \mid trans2.2 \in \mathrm{ran}\ del \bullet \\
&\quad trans.2 = in(genAction(currentValues(\mathrm{dom}\ d.I \lhd trans2.3)))) \vee \\
&(trans.2 \in \mathrm{ran}\ tiots\_del \wedge \exists\, trans2 : d.TR \mid trans2.2 \in \mathrm{ran}\ del \bullet \\
&\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \vee \\
&(trans.2 \in \mathrm{ran}\ out \wedge \exists\, trans2 : d.TR \mid trans2.2 \in \mathrm{ran}\ del \bullet \\
&\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\ d.O \lhd trans2.3))))
\end{aligned}$$

■

B.6.1.7 Mapping transitions

Lemma 7.

$$
\begin{array}{l}
\forall t : TIOTS \bullet \forall d : e\_DFRS \bullet \\
\quad (t = fromDFRStoTIOTS(d) \Rightarrow \\
\quad\quad \forall trans : t.T \bullet \\
\quad\quad\quad (trans.2 \in \operatorname{ran} in \land \exists trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\quad\quad\quad\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3)))) \lor \\
\quad\quad\quad (trans.2 \in \operatorname{ran} tiots\_del \land \exists trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\quad\quad\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\quad\quad\quad (trans.2 \in \operatorname{ran} out \land \\
\quad\quad\quad \exists trans2 : d.TR \mid (trans2.2 \in \operatorname{ran} fun \lor trans2.2 \in \operatorname{ran} del) \bullet \\
\quad\quad\quad\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \lor \\
\quad\quad\quad (trans.2 \in \operatorname{ran} tau))
\end{array}
$$

proof

Let t and d range over values in the sets TIOTS and e_DFRS, respectively.

$t = fromDFRStoTIOTS(d)$

$\Rightarrow$ *[defintion of fromDFRStoTIOTS]*

$t.T = mapTransitionRelation(d.TR, d.I, d.O)$

$\Leftrightarrow$ *[definition of mapTransitionRelation]*

$$
\begin{array}{l}
\exists\ tr1, tr2 : TIOTS\_TRANSREL \bullet t.T = tr1 \cup tr2 \land \\
\quad tr1 = mapFunTransitions(getTransitions(d.TR, \operatorname{ran} fun), d.TR, d.O) \land \\
\quad tr2 = mapDelTransitions(getTransitions(d.TR, \operatorname{ran} del), d.TR, \\
\quad\quad \operatorname{ran} mapState, d.I, d.O)
\end{array}
$$

$\Leftrightarrow$ *[predicate calculus – one-point rule]*

$$
\begin{array}{l}
t.T = mapFunTransitions(getTransitions(d.TR, \operatorname{ran} fun), d.TR, d.O) \cup \\
\quad mapDelTransitions(getTransitions(d.TR, \operatorname{ran} del), d.TR, \operatorname{ran} mapState, d.I, d.O)
\end{array}
$$

$\Rightarrow$ *[property of sets]*

$$\begin{aligned}
&\forall\, trans : t.T \bullet \\
&\quad trans \in mapFunTransitions(getTransitions(d.TR, \operatorname{ran} fun), d.TR, d.O) \lor \\
&\quad trans \in mapDelTransitions(getTransitions(d.TR, \operatorname{ran} del), d.TR, \\
&\qquad \operatorname{ran} mapState, d.I, d.O)
\end{aligned}$$

$\Rightarrow$ *[Lemma 4]*

$$\begin{aligned}
&\forall\, trans : t.T \bullet \\
&\quad (trans.2 \in \operatorname{ran} out \land \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} fun \bullet \\
&\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \lor \\
&\quad (trans.2 \in \operatorname{ran} tau) \lor \\
&\quad trans \in mapDelTransitions(getTransitions(d.TR, \operatorname{ran} del), d.TR, \\
&\qquad \operatorname{ran} mapState, d.I, d.O)
\end{aligned}$$

$\Rightarrow$ *[Lemma 6]*

$$\begin{aligned}
&\forall\, trans : t.T \bullet \\
&\quad (trans.2 \in \operatorname{ran} out \land \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} fun \bullet \\
&\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \lor \\
&\quad (trans.2 \in \operatorname{ran} tau) \lor \\
&\quad (trans.2 \in \operatorname{ran} in \land \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
&\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3)))) \lor \\
&\quad (trans.2 \in \operatorname{ran} tiots\_del \land \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
&\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
&\quad (trans.2 \in \operatorname{ran} out \land \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
&\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))))
\end{aligned}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l}
\forall\ trans : t.T\ \bullet \\
\quad (trans.2 \in \operatorname{ran} in \land \exists\ trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del\ \bullet \\
\quad\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3)))) \lor \\
\quad (trans.2 \in \operatorname{ran} tiots\_del \land \exists\ trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del\ \bullet \\
\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\quad (trans.2 \in \operatorname{ran} out \land \\
\quad \exists\ trans2 : d.TR \mid (trans2.2 \in \operatorname{ran} fun \lor trans2.2 \in \operatorname{ran} del)\ \bullet \\
\quad\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \lor \\
\quad (trans.2 \in \operatorname{ran} tau)
\end{array}$$

■

B.6.1.8 Property of genAction

Lemma 8.

$$\begin{array}{l}
\forall t : TIOTS ; d : e\_DFRS; \\
f : (TIOTS\_ACTION \rightarrowtail TIOTS\_TRANS\_LABEL) ; g : (VNAME \twoheadrightarrow TYPE)\ \bullet \\
\quad (\forall a : A \bullet \exists trans : t.T \bullet a = (f^{\sim})(trans.2) \land \\
\quad\quad \exists trans2 : d.TR \bullet trans.2 = f(genAction(currentValues(\operatorname{dom} g \lhd trans2.3)))) \Rightarrow \\
\quad (\forall a : A \bullet \exists h : (NAME \rightarrow VALUE) \bullet a = genAction(h) \land \operatorname{dom} h \subseteq \operatorname{dom} g)
\end{array}$$

proof
Let t, d, f, and g range over values in the sets TIOTS, e_DFRS, (TIOTS_ACTION $\rightarrowtail$ TIOTS_TRANS_LABEL), and (VNAME $\twoheadrightarrow$ TYPE), respectively.

$$\begin{array}{l}
\forall\ a : A\ \bullet\ \exists\ trans : t.T\ \bullet\ a = (f^{\sim})(trans.2) \land \\
\quad \exists\ trans2 : d.TR\ \bullet\ trans.2 = f(genAction(currentValues(\operatorname{dom}\ g \lhd trans2.3)))
\end{array}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l}
\forall\ a : A\ \bullet\ \exists\ trans : t.T\ \bullet\ \exists\ trans2 : d.TR\ \bullet \\
\quad a = (f^{\sim})(trans.2) \land trans.2 = f(genAction(currentValues(\operatorname{dom}\ g \lhd trans2.3)))
\end{array}$$

$\Leftrightarrow$ *[predicate calculus – one-point rule]*

$$\begin{array}{l}
\forall\ a : A\ \bullet\ \exists\ trans2 : d.TR\ \bullet \\
\quad a = (f^{\sim})(f(genAction(currentValues(\operatorname{dom}\ g \lhd trans2.3))))
\end{array}$$

$\Leftrightarrow$ [property of injective functions – $x = (f^{\sim})(f(h(y))) \Leftrightarrow x = h(y)$]

$$\forall\, a : A \bullet \exists\, trans2 : d.TR \bullet$$
$$a = genAction(currentValues(\mathrm{dom}\; g \lhd trans2.3))$$

$\Leftrightarrow$ [predicate calculus – one-point rule]

$$\forall\, a : A \bullet \exists\, trans2 : d.TR;\; h : (NAME \pfun VALUE) \bullet a = genAction(h) \land$$
$$h = currentValues(\mathrm{dom}\; g \lhd trans2.3)$$

$\Leftrightarrow$ [definition of *currentValues*]

$$\forall\, a : A \bullet \exists\, trans2 : d.TR;\; h : (NAME \pfun VALUE) \bullet a = genAction(h) \land$$
$$h = \{\, n : NAME;\; v1, v2 : VALUE \mid (n, (v1, v2)) \in \mathrm{dom}\; g \lhd trans2.3 \bullet (n, v2) \,\}$$

$\Leftrightarrow$ [definition of $\lhd$]

$$\forall\, a : A \bullet \exists\, trans2 : d.TR;\; h : (NAME \pfun VALUE) \bullet a = genAction(h) \land$$
$$h = \{\, n : NAME;\; v1, v2 : VALUE \mid (n, (v1, v2)) \in$$
$$\{\, x : NAME;\; y1, y2 : VALUE \mid (x, (y1, y2)) \in trans2.3 \land$$
$$x \in \mathrm{dom}\; g \bullet (x, (y1, y2)) \,\}$$
$$\bullet (n, v2) \,\}$$

$\Leftrightarrow$ [definition of set comprehension]

$$\forall\, a : A \bullet \exists\, trans2 : d.TR;\; h : (NAME \pfun VALUE) \bullet a = genAction(h) \land$$
$$h = \{\, n : NAME;\; v1, v2 : VALUE \mid$$
$$(\exists\, x : NAME;\; y1, y2 : VALUE \mid (x, (y1, y2)) \in trans2.3 \land$$
$$x \in \mathrm{dom}\; g \land (n, (v1, v2)) = (x, (y1, y2)))$$
$$\bullet (n, v2) \,\}$$

$\Leftrightarrow$ [predicate calculus – one-point rule]

$$\forall\, a : A \bullet \exists\, trans2 : d.TR;\; h : (NAME \pfun VALUE) \bullet a = genAction(h) \land$$
$$h = \{\, n : NAME;\; v1, v2 : VALUE \mid$$
$$(n, (v1, v2)) \in trans2.3 \land n \in \mathrm{dom}\; g \bullet (n, v2) \,\}$$

$\Rightarrow$ *[property of set comprehension]*

$$\forall\, a : A \bullet \exists\, trans2 : d.TR;\ h : (NAME \twoheadrightarrow VALUE) \bullet a = genAction(h) \land \\ \forall\, e : h \bullet (\exists\, n : NAME;\ v1, v2 : VALUE \mid \\ (n,(v1,v2)) \in trans2.3 \land n \in \operatorname{dom} g \bullet e = (n, v2))$$

$\Rightarrow$ *[property of cartesian product – $e = (a,b) \Rightarrow e.1 = a$]*

$$\forall\, a : A \bullet \exists\, trans2 : d.TR;\ h : (NAME \twoheadrightarrow VALUE) \bullet a = genAction(h) \land \\ \forall\, e : h \bullet \exists\, n : NAME;\ v1, v2 : VALUE \mid \\ (n,(v1,v2)) \in trans2.3 \land n \in \operatorname{dom} g \bullet e.1 = n$$

$\Rightarrow$ *[property of sets – $(a \in X \land b = a) \Rightarrow (a \in X \land b \in X)$]*

$$\forall\, a : A \bullet \exists\, trans2 : d.TR;\ h : (NAME \twoheadrightarrow VALUE) \bullet a = genAction(h) \land \\ \forall\, e : h \bullet \exists\, n : NAME;\ v1, v2 : VALUE \mid \\ (n,(v1,v2)) \in trans2.3 \land n \in \operatorname{dom} g \bullet e.1 \in \operatorname{dom} g$$

$\Rightarrow$ *[propositional calculus]*

$$\forall\, a : A \bullet \exists\, trans2 : d.TR;\ h : (NAME \twoheadrightarrow VALUE) \bullet a = genAction(h) \land \\ \forall\, e : h \bullet \exists\, n : NAME;\ v1, v2 : VALUE \bullet e.1 \in \operatorname{dom} g$$

$\Leftrightarrow$ *[predicate calculus]*

$$\forall\, a : A \bullet \exists\, h : (NAME \twoheadrightarrow VALUE) \bullet a = genAction(h) \land \\ \forall\, e : h \bullet e.1 \in \operatorname{dom} g$$

$\Leftrightarrow$ *[definition of $\subseteq$]*

$$\forall\, a : A \bullet \exists\, h : (NAME \twoheadrightarrow VALUE) \bullet a = genAction(h) \land \operatorname{dom} h \subseteq \operatorname{dom} g$$

■

B.6.1.9 Proof of disjointness of $t.I$ and $t.O$

Lemma 9.

$$\forall\, t : TIOTS \bullet \forall\, d : e\_DFRS \bullet \\ (t = fromDFRStoTIOTS(d) \Rightarrow \mathsf{disjoint}\ \langle t.I, t.O \rangle)$$

proof

Let t and d range over values in the sets TIOTS and e_DFRS, respectively.

$t = fromDFRStoTIOTS(d)$

$\Rightarrow$ *[definition of fromDFRStoTIOTS]*

$t = fromDFRStoTIOTS(d) \land$
$t.I = getInputActions(t.T) \land t.O = getOutputActions(t.T)$

$\Leftrightarrow$ *[definition of getInputActions and getOutputActions]*

$t = fromDFRStoTIOTS(d) \land$
$t.I = \{ trans : t.T \mid trans.2 \in \mathrm{ran}\ in \bullet (in^{\sim})(trans.2) \} \land$
$t.O = \{ trans : t.T \mid trans.2 \in \mathrm{ran}\ out \bullet (out^{\sim})(trans.2) \}$

$\Rightarrow$ *[property of set comprehension]*

$t = fromDFRStoTIOTS(d) \land$
$(\forall i : t.I \bullet \exists trans : t.T \mid trans.2 \in \mathrm{ran}\ in \bullet i = (in^{\sim})(trans.2)) \land$
$(\forall o : t.O \bullet \exists trans : t.T \mid trans.2 \in \mathrm{ran}\ out \bullet o = (out^{\sim})(trans.2))$

$\Rightarrow$ *[Lemma 7]*

$(\forall i : t.I \bullet \exists trans : t.T \mid trans.2 \in \mathrm{ran}\ in \bullet i = (in^{\sim})(trans.2) \land$
$\quad ((trans.2 \in \mathrm{ran}\ in \land \exists trans2 : d.TR \mid trans2.2 \in \mathrm{ran}\ del \bullet$
$\quad\quad trans.2 = in(genAction(currentValues(\mathrm{dom}\ d.I \lhd trans2.3)))) \lor$
$\quad (trans.2 \in \mathrm{ran}\ tiots\_del \land \exists trans2 : d.TR \mid trans2.2 \in \mathrm{ran}\ del \bullet$
$\quad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor$
$\quad (trans.2 \in \mathrm{ran}\ out \land$
$\quad \exists trans2 : d.TR \mid (trans2.2 \in \mathrm{ran}\ fun \lor trans2.2 \in \mathrm{ran}\ del) \bullet$
$\quad\quad trans.2 = out(genAction(currentValues(\mathrm{dom}\ d.O \lhd trans2.3)))) \lor$
$\quad (trans.2 \in \mathrm{ran}\ tau))) \land$

$$\begin{array}{l}
(\forall\ o : t.O \bullet \exists\ trans : t.T \mid trans.2 \in \operatorname{ran} out \bullet o = (out^{\sim})(trans.2) \wedge \\
\quad ((trans.2 \in \operatorname{ran} in \wedge \exists\ trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3)))) \vee \\
\quad (trans.2 \in \operatorname{ran} tiots\_del \wedge \exists\ trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \vee \\
\quad (trans.2 \in \operatorname{ran} out \wedge \\
\quad \exists\ trans2 : d.TR \mid (trans2.2 \in \operatorname{ran} fun \vee trans2.2 \in \operatorname{ran} del) \bullet \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \vee \\
\quad (trans.2 \in \operatorname{ran} tau)))
\end{array}$$

$\Leftrightarrow$ [definition of *TIOTS_TRANS_LABEL* – disjoint $\langle$ ran *in*, ran *out*, ran *tiots_del*, ran *tau*$\rangle$]

$$\begin{array}{l}
(\forall\ i : t.I \bullet \exists\ trans : t.T \bullet i = (in^{\sim})(trans.2) \wedge \\
\quad trans.2 \in \operatorname{ran} in \wedge \exists\ trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3)))) \wedge \\
(\forall\ o : t.O \bullet \exists\ trans : t.T \bullet o = (out^{\sim})(trans.2) \wedge \\
\quad trans.2 \in \operatorname{ran} out \wedge \\
\quad \exists\ trans2 : d.TR \mid (trans2.2 \in \operatorname{ran} fun \vee trans2.2 \in \operatorname{ran} del) \bullet \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))))
\end{array}$$

$\Rightarrow$ [propositional calculus]

$$\begin{array}{l}
(\forall\ i : t.I \bullet \exists\ trans : t.T \bullet i = (in^{\sim})(trans.2) \wedge \\
\quad \exists\ trans2 : d.TR \bullet \\
\qquad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3)))) \wedge \\
(\forall\ o : t.O \bullet \exists\ trans : t.T \bullet o = (out^{\sim})(trans.2) \wedge \\
\quad \exists\ trans2 : d.TR \bullet \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))))
\end{array}$$

$\Rightarrow$ [Lemma 8, where $a \equiv i$, $A \equiv t.I$, $f \equiv in$, $g \equiv d.I$]

$$\begin{array}{l}
(\forall\ i : t.I \bullet \exists\ h : (NAME \nrightarrow VALUE) \bullet i = genAction(h) \wedge \operatorname{dom} h \subseteq \operatorname{dom} d.I) \wedge \\
(\forall\ o : t.O \bullet \exists\ trans : t.T \bullet o = (out^{\sim})(trans.2) \wedge \\
\quad \exists\ trans2 : d.TR \bullet \\
\qquad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3))))
\end{array}$$

$\Rightarrow$ *[Lemma 8, where $a \equiv o$, $A \equiv t.O$, $f \equiv out$, $g \equiv d.O$]*

$$(\forall\, i : t.I \bullet \exists\, h : (NAME \pfun VALUE) \bullet i = genAction(h) \land \operatorname{dom} h \subseteq \operatorname{dom} d.I) \land$$
$$(\forall\, o : t.O \bullet \exists\, h : (NAME \pfun VALUE) \bullet o = genAction(h) \land \operatorname{dom} h \subseteq \operatorname{dom} d.O)$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l} \forall\, i : t.I;\, o : t.O \bullet \exists\, h1, h2 : (NAME \pfun VALUE) \bullet \\ \quad \operatorname{dom} h1 \subseteq \operatorname{dom} d.I \land \operatorname{dom} h2 \subseteq \operatorname{dom} d.O \land \\ \quad i = genAction(h1) \land o = genAction(h2) \end{array}$$

$\Rightarrow$ *[definition of e_DFRS]*

$$\begin{array}{l} \forall\, i : t.I;\, o : t.O \bullet \exists\, h1, h2 : (NAME \pfun VALUE) \bullet \\ \quad \operatorname{dom} d.I \cap \operatorname{dom} d.O = \emptyset \land \\ \quad \operatorname{dom} h1 \subseteq \operatorname{dom} d.I \land \operatorname{dom} h2 \subseteq \operatorname{dom} d.O \land \\ \quad i = genAction(h1) \land o = genAction(h2) \end{array}$$

$\Rightarrow$ *[property of sets]*

$$\begin{array}{l} \forall\, i : t.I;\, o : t.O \bullet \exists\, h1, h2 : (NAME \pfun VALUE) \bullet \\ \quad h1 \neq h2 \land i = genAction(h1) \land o = genAction(h2) \end{array}$$

$\Rightarrow$ *[definition of genAction – injective function]*

$$\forall\, i : t.I;\, o : t.O \bullet i \neq o$$

$\Leftrightarrow$ *[definition of disjoint]*

$$\text{disjoint}\ \langle t.I, t.O \rangle$$

■

B.6.2 Time compatibility

B.6.2.1 Transformation: implies to equivalence

Lemma 10.

$((a \Rightarrow b) \wedge (c \Rightarrow d)) \Rightarrow ((a \Rightarrow b) \wedge (c \Leftrightarrow d))$
provided
$(a \vee c) \wedge \neg (b \wedge d)$

proof

$(a \Rightarrow b) \wedge (c \Rightarrow d)$

$\Leftrightarrow$ *[propositional calculus]*

$(a \Rightarrow b) \wedge (a \Rightarrow b) \wedge (c \Rightarrow d)$

$\Leftrightarrow$ *[propositional calculus]*

$(a \Rightarrow b) \wedge (\neg\, a \vee b) \wedge (c \Rightarrow d)$

$\Leftrightarrow$ *[proviso]*

$(a \Rightarrow b) \wedge ((\neg\, a \wedge (a \vee c)) \vee b) \wedge (c \Rightarrow d)$

$\Leftrightarrow$ *[propositional calculus]*

$(a \Rightarrow b) \wedge ((\neg\, a \wedge (\neg\, a \Rightarrow c)) \vee b) \wedge (c \Rightarrow d)$

$\Rightarrow$ *[propositional calculus]*

$(a \Rightarrow b) \wedge (c \vee b) \wedge (c \Rightarrow d)$

$\Leftrightarrow$ *[proviso]*

$(a \Rightarrow b) \wedge (c \vee (b \wedge \neg\,(b \wedge d))) \wedge (c \Rightarrow d)$

$\Leftrightarrow$ *[propositional calculus]*

$(a \Rightarrow b) \wedge (c \vee (b \wedge (\neg b \vee \neg d))) \wedge (c \Rightarrow d)$

$\Leftrightarrow$ *[propositional calculus]*

$(a \Rightarrow b) \wedge (c \vee (b \wedge (b \Rightarrow \neg d))) \wedge (c \Rightarrow d)$

$\Rightarrow$ *[propositional calculus]*

$(a \Rightarrow b) \wedge (c \vee \neg d) \wedge (c \Rightarrow d)$

$\Leftrightarrow$ *[propositional calculus]*

$(a \Rightarrow b) \wedge (d \Rightarrow c) \wedge (c \Rightarrow d)$

$\Leftrightarrow$ *[propositional calculus]*

$(a \Rightarrow b) \wedge (c \Leftrightarrow d)$

$\blacksquare$

B.6.2.2 Transformation: implies elimination

Lemma 11.

$(\forall x : X \mid Q(x) \bullet f(y) = k_1 \Rightarrow P_1(x)) \wedge (\forall x : X \mid Q(x) \bullet f(y) = k_2 \Rightarrow P_2(x))$
$\Rightarrow$
$(\forall x : X \mid Q(x) \bullet P_1(x)) \vee (\forall x : X \mid Q(x) \bullet P_2(x))$
provided
$(f(y) = k_1 \vee f(y) = k_2) \wedge \neg (\forall x : X \mid Q(x) \bullet P_1(x) \wedge P_2(x))$

proof

$(\forall x : X \mid Q(x) \bullet f(y) = k_1 \Rightarrow P_1(x)) \wedge$
$(\forall x : X \mid Q(x) \bullet f(y) = k_2 \Rightarrow P_2(x))$

$\Leftrightarrow$ *[propositional calculus]*

$$(\forall\, x : X \mid Q(x) \bullet \neg\,(f(y) = k_1) \lor P_1(x)) \land$$
$$(\forall\, x : X \mid Q(x) \bullet \neg\,(f(y) = k_2) \lor P_2(x))$$

$\Leftrightarrow$ *[predicate calculus]*

$$(\neg\,(f(y) = k_1) \lor (\forall\, x : X \mid Q(x) \bullet P_1(x))) \land$$
$$(\neg\,(f(y) = k_2) \lor (\forall\, x : X \mid Q(x) \bullet P_2(x)))$$

$\Leftrightarrow$ *[propositional calculus]*

$$(\neg\,(f(y) = k_1) \land \neg\,(f(y) = k_2)) \lor$$
$$(\neg\,(f(y) = k_1) \land (\forall\, x : X \mid Q(x) \bullet P_2(x))) \lor$$
$$(\neg\,(f(y) = k_2) \land (\forall\, x : X \mid Q(x) \bullet P_1(x))) \lor$$
$$((\forall\, x : X \mid Q(x) \bullet P_1(x)) \land (\forall\, x : X \mid Q(x) \bullet P_2(x)))$$

$\Leftrightarrow$ *[propositional calculus]*

$$(\neg\,(f(y) = k_1 \lor f(y) = k_2)) \lor$$
$$(\neg\,(f(y) = k_1) \land (\forall\, x : X \mid Q(x) \bullet P_2(x))) \lor$$
$$(\neg\,(f(y) = k_2) \land (\forall\, x : X \mid Q(x) \bullet P_1(x))) \lor$$
$$((\forall\, x : X \mid Q(x) \bullet P_1(x)) \land (\forall\, x : X \mid Q(x) \bullet P_2(x)))$$

$\Leftrightarrow$ *[proviso]*

$$(\neg\,(\mathit{true})) \lor$$
$$(\neg\,(f(y) = k_1) \land (\forall\, x : X \mid Q(x) \bullet P_2(x))) \lor$$
$$(\neg\,(f(y) = k_2) \land (\forall\, x : X \mid Q(x) \bullet P_1(x))) \lor$$
$$((\forall\, x : X \mid Q(x) \bullet P_1(x)) \land (\forall\, x : X \mid Q(x) \bullet P_2(x)))$$

$\Leftrightarrow$ *[propositional calculus]*

$$(\neg\,(f(y) = k_1) \land (\forall\, x : X \mid Q(x) \bullet P_2(x))) \lor$$
$$(\neg\,(f(y) = k_2) \land (\forall\, x : X \mid Q(x) \bullet P_1(x))) \lor$$
$$((\forall\, x : X \mid Q(x) \bullet P_1(x)) \land (\forall\, x : X \mid Q(x) \bullet P_2(x)))$$

$\Leftrightarrow$ *[predicate calculus]*

$(\neg (f(y) = k_1) \wedge (\forall x : X \mid Q(x) \bullet P_2(x))) \vee$
$(\neg (f(y) = k_2) \wedge (\forall x : X \mid Q(x) \bullet P_1(x))) \vee$
$(\forall x : X \mid Q(x) \bullet P_1(x) \wedge P_2(x))$

$\Leftrightarrow$ *[proviso]*

$(\neg (f(y) = k_1) \wedge (\forall x : X \mid Q(x) \bullet P_2(x))) \vee$
$(\neg (f(y) = k_2) \wedge (\forall x : X \mid Q(x) \bullet P_1(x))) \vee$
(false)

$\Leftrightarrow$ *[propositional calculus]*

$(\neg (f(y) = k_1) \wedge (\forall x : X \mid Q(x) \bullet P_2(x))) \vee$
$(\neg (f(y) = k_2) \wedge (\forall x : X \mid Q(x) \bullet P_1(x))) \vee$

$\Rightarrow$ *[propositional calculus]*

$(\forall x : X \mid Q(x) \bullet P_1(x)) \vee (\forall x : X \mid Q(x) \bullet P_2(x))$
■

B.6.2.3 Proof of time compatibility

Lemma 12.

$$\forall t : \mathit{TIOTS} \bullet \forall d : \mathit{e\_DFRS} \bullet$$
$$(t = \mathit{fromDFRStoTIOTS}(d) \Rightarrow t.D \in \mathit{tiots\_time\_compatible})$$

proof

Let t and d range over values in the sets TIOTS and e_DFRS, respectively.

$t = \mathit{fromDFRStoTIOTS}(d)$

$\Rightarrow$ [definition of *e_DFRS*]

$t = \mathit{fromDFRStoTIOTS}(d) \land$
$\forall\, \mathit{trans} : d.\mathit{TR} \bullet$
$\quad (\mathit{trans}.2,\ d.I,\ d.O,\ d.T,\ d.\mathit{gcvar}) \in \mathit{well\_typed\_transition}$

$\Rightarrow$ [definition of *well_typed_transition*]

$t = \mathit{fromDFRStoTIOTS}(d) \land$
$\forall\, \mathit{trans} : d.\mathit{TR} \mid \mathit{trans}.2 \in \operatorname{ran} \mathit{del} \bullet$
$\quad (\mathit{trans}.2,\ d.\mathit{gcvar}) \in \mathit{clock\_compatible\_transition}$

$\Rightarrow$ [definition of *clock_compatible_transition*]

$t = \mathit{fromDFRStoTIOTS}(d) \land$
$\forall\, \mathit{trans} : d.\mathit{TR} \mid \mathit{trans}.2 \in \operatorname{ran} \mathit{del} \bullet$
$\quad ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{discrete} \Rightarrow d.\mathit{gcvar}.2 = \mathit{nat}) \land$
$\quad ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{dense} \Rightarrow d.\mathit{gcvar}.2 = \mathit{ufloat})$

$\Rightarrow$ [Lemma 10, where $a \equiv ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{discrete}$, $b \equiv d.\mathit{gcvar}.2 = \mathit{nat}$, $c \equiv ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{dense}$, $d \equiv d.\mathit{gcvar}.2 = \mathit{ufloat}$, $(a \lor c)$ due to the definition of *DELAY*, $\neg\,(b \land d)$ due to the definition of *TYPE*]

$t = \mathit{fromDFRStoTIOTS}(d) \land$
$\forall\, \mathit{trans} : d.\mathit{TR} \mid \mathit{trans}.2 \in \operatorname{ran} \mathit{del} \bullet$
$\quad ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{discrete} \Rightarrow d.\mathit{gcvar}.2 = \mathit{nat}) \land$
$\quad ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{dense} \Leftrightarrow d.\mathit{gcvar}.2 = \mathit{ufloat})$

$\Rightarrow$ [Lemma 10, where $c \equiv ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{discrete}$, $d \equiv d.\mathit{gcvar}.2 = \mathit{nat}$, $a \equiv ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{dense}$, $b \equiv d.\mathit{gcvar}.2 = \mathit{ufloat}$, $(a \lor c)$ due to the definition of *DELAY*, $\neg\,(b \land d)$ due to the definition of *TYPE*]

$t = \mathit{fromDFRStoTIOTS}(d) \land$
$\forall\, \mathit{trans} : d.\mathit{TR} \mid \mathit{trans}.2 \in \operatorname{ran} \mathit{del} \bullet$
$\quad ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{discrete} \Leftrightarrow d.\mathit{gcvar}.2 = \mathit{nat}) \land$
$\quad ((\mathit{delayTransition}(\mathit{trans}.2)).1 \in \operatorname{ran} \mathit{dense} \Leftrightarrow d.\mathit{gcvar}.2 = \mathit{ufloat})$

$\Rightarrow$ *[propositional calculus]*

$$\begin{array}{l}
t = fromDFRStoTIOTS(d) \land \\
\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad (d.gcvar.2 = nat \Rightarrow (delayTransition(trans.2)).1 \in \operatorname{ran} discrete) \land \\
\quad (d.gcvar.2 = ufloat \Rightarrow (delayTransition(trans.2)).1 \in \operatorname{ran} dense)
\end{array}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l}
t = fromDFRStoTIOTS(d) \land \\
(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad d.gcvar.2 = nat \Rightarrow (delayTransition(trans.2)).1 \in \operatorname{ran} discrete) \land \\
(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad d.gcvar.2 = ufloat \Rightarrow (delayTransition(trans.2)).1 \in \operatorname{ran} dense)
\end{array}$$

$\Rightarrow$ *[Lemma 11, where* $f(y) \equiv d.gcvar.2$, $k_1 \equiv nat \equiv$, $k_2 \equiv ufloat$,
$x \equiv trans$, $X \equiv d.TR$, $Q(x) \equiv trans.2 \in \operatorname{ran} del$,
$P_1(x) \equiv (delayTransition(trans.2)).1 \in \operatorname{ran} discrete$,
$P_2(x) \equiv (delayTransition(entry.2)).1 \in \operatorname{ran} dense$,
$(f(y) = k_1 \lor f(y) = k_2)$ *due to the definition of TYPE*,
$\neg (\forall\, x : X \mid Q(x) \bullet P_1(x) \land P_2(x))$ *due to the definition of DELAY]*

$$\begin{array}{l}
t = fromDFRStoTIOTS(d) \land \\
((\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad (delayTransition(trans.2)).1 \in \operatorname{ran} discrete) \lor \\
\ (\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad (delayTransition(trans.2)).1 \in \operatorname{ran} dense))
\end{array}$$

$\Rightarrow$ *[definition of mapDelay,*
$\forall\, x : DELAY \bullet mapDelay(x) \in \operatorname{ran} tiots\_discrete \Leftrightarrow x \in \operatorname{ran} discrete]$

$$\begin{array}{l}
t = fromDFRStoTIOTS(d) \land \\
((\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete) \lor \\
\ (\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad (delayTransition(trans.2)).1 \in \operatorname{ran} dense))
\end{array}$$

$\Rightarrow$ [definition of *mapDelay*,
$\forall x : DELAY \bullet mapDelay(x) \in \text{ran } tiots\_dense \Leftrightarrow x \in \text{ran } dense$]

$$t = fromDFRStoTIOTS(d) \wedge$$
$$((\forall trans : d.TR \mid trans.2 \in \text{ran } del \bullet$$
$$mapDelay((delayTransition(trans.2)).1) \in \text{ran } tiots\_discrete) \vee$$
$$(\forall trans : d.TR \mid trans.2 \in \text{ran } del \bullet$$
$$mapDelay((delayTransition(trans.2)).1) \in \text{ran } tiots\_dense))$$

$\Rightarrow$ [definition of *fromDFRStoTIOTS*]

$$t.D = getDelays(t.T) \wedge$$
$$((\forall trans : d.TR \mid trans.2 \in \text{ran } del \bullet$$
$$mapDelay((delayTransition(trans.2)).1) \in \text{ran } tiots\_discrete) \vee$$
$$(\forall trans : d.TR \mid trans.2 \in \text{ran } del \bullet$$
$$mapDelay((delayTransition(trans.2)).1) \in \text{ran } tiots\_dense))$$

$\Leftrightarrow$ [definition of *getDelays*]

$$t.D = \{ trans : t.T \mid trans.2 \in \text{ran } tiots\_del \bullet (tiots\_del^{\sim})(trans.2) \} \wedge$$
$$((\forall trans : d.TR \mid trans.2 \in \text{ran } del \bullet$$
$$mapDelay((delayTransition(trans.2)).1) \in \text{ran } tiots\_discrete) \vee$$
$$(\forall trans : d.TR \mid trans.2 \in \text{ran } del \bullet$$
$$mapDelay((delayTransition(trans.2)).1) \in \text{ran } tiots\_dense))$$

$\Rightarrow$ [property of set comprehension]

$$(\forall dV : t.D \bullet$$
$$\exists trans : t.T \mid trans.2 \in \text{ran } tiots\_del \bullet dV = (tiots\_del^{\sim})(trans.2)) \wedge$$
$$((\forall trans : d.TR \mid trans.2 \in \text{ran } del \bullet$$
$$mapDelay((delayTransition(trans.2)).1) \in \text{ran } tiots\_discrete) \vee$$
$$(\forall trans : d.TR \mid trans.2 \in \text{ran } del \bullet$$
$$mapDelay((delayTransition(trans.2)).1) \in \text{ran } tiots\_dense))$$

$\Rightarrow$ *[Lemma 7]*

$$\begin{array}{l}
(\forall\, dV : t.D \bullet \\
\quad \exists\, trans : t.T \mid trans.2 \in \operatorname{ran} tiots\_del \bullet dV = (tiots\_del^{\sim})(trans.2) \land \\
\qquad ((trans.2 \in \operatorname{ran} in \land \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\qquad\quad trans.2 = in(genAction(currentValues(\operatorname{dom} d.I \lhd trans2.3)))) \lor \\
\qquad (trans.2 \in \operatorname{ran} tiots\_del \land \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\qquad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \lor \\
\qquad (trans.2 \in \operatorname{ran} out \land \\
\qquad \exists\, trans2 : d.TR \mid (trans2.2 \in \operatorname{ran} fun \lor trans2.2 \in \operatorname{ran} del) \bullet \\
\qquad\quad trans.2 = out(genAction(currentValues(\operatorname{dom} d.O \lhd trans2.3)))) \lor \\
\qquad (trans.2 \in \operatorname{ran} tau))) \land \\
((\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete) \lor \\
(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_dense))
\end{array}$$

$\Leftrightarrow$ *[definition of TIOTS_TRANS_LABEL –*
disjoint $\langle \operatorname{ran} in, \operatorname{ran} out, \operatorname{ran} tiots\_del, \operatorname{ran} tau\rangle$*]*

$$\begin{array}{l}
(\forall\, dV : t.D \bullet \\
\quad \exists\, trans : t.T \bullet dV = (tiots\_del^{\sim})(trans.2) \land trans.2 \in \operatorname{ran} tiots\_del \land \\
\qquad \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\qquad\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1)) \land \\
((\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete) \lor \\
(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_dense))
\end{array}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l}
(\forall\, dV : t.D \bullet \exists\, trans : t.T \bullet trans.2 \in \operatorname{ran} tiots\_del \land \\
\quad \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\qquad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1) \land \\
\qquad dV = (tiots\_del^{\sim})(trans.2)) \land \\
((\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete) \lor \\
(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_dense))
\end{array}$$

$\Rightarrow$ *[propositional calculus]*

$$\begin{aligned}
&(\forall\, dV : t.D \bullet \exists\, trans : t.T \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
&\quad trans.2 = tiots\_del(mapDelay(delayTransition(trans2.2)).1) \land \\
&\quad dV = (tiots\_del^{\sim})(trans.2)) \land \\
&((\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
&\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete) \lor \\
&(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
&\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_dense))
\end{aligned}$$

$\Rightarrow$ *[predicate calculus]*

$$\begin{aligned}
&(\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
&\quad dV = (tiots\_del^{\sim})(tiots\_del(mapDelay(delayTransition(trans2.2)).1))) \land \\
&((\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
&\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete) \lor \\
&(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
&\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_dense))
\end{aligned}$$

$\Leftrightarrow$ *[property of injective functions – $x = (f^{\sim})(f(g(y))) \Leftrightarrow x = g(y)$]*

$$\begin{aligned}
&(\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
&\quad dV = mapDelay(delayTransition(trans2.2)).1) \land \\
&((\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
&\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete) \lor \\
&(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
&\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_dense))
\end{aligned}$$

$\Leftrightarrow$ *[propositional calculus]*

$$\begin{aligned}
&((\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
&\quad dV = mapDelay(delayTransition(trans2.2)).1) \land \\
&(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
&\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete)) \lor \\
&((\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
&\quad dV = mapDelay(delayTransition(trans2.2)).1) \land \\
&(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
&\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_dense))
\end{aligned}$$

$\Leftrightarrow$ *[predicate calculus]*

$$\begin{array}{l}
((\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\quad dV = mapDelay(delayTransition(trans2.2)).1 \land \\
\quad trans2.2 \in \operatorname{ran} del \Rightarrow \\
\qquad mapDelay((delayTransition(trans2.2)).1) \in \operatorname{ran} tiots\_discrete) \land \\
\;(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete)) \lor \\
((\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\quad dV = mapDelay(delayTransition(trans2.2)).1 \land \\
\quad trans2.2 \in \operatorname{ran} del \Rightarrow \\
\qquad mapDelay((delayTransition(trans2.2)).1) \in \operatorname{ran} tiots\_dense) \land \\
\;(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_dense))
\end{array}$$

$\Leftrightarrow$ *[propositional calculus]*

$$\begin{array}{l}
((\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\quad dV = mapDelay(delayTransition(trans2.2)).1 \land \\
\quad mapDelay((delayTransition(trans2.2)).1) \in \operatorname{ran} tiots\_discrete) \land \\
\;(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_discrete)) \lor \\
((\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\quad dV = mapDelay(delayTransition(trans2.2)).1 \land \\
\quad mapDelay((delayTransition(trans2.2)).1) \in \operatorname{ran} tiots\_dense) \land \\
\;(\forall\, trans : d.TR \mid trans.2 \in \operatorname{ran} del \bullet \\
\quad mapDelay((delayTransition(trans.2)).1) \in \operatorname{ran} tiots\_dense))
\end{array}$$

$\Rightarrow$ *[propositional calculus]*

$$\begin{array}{l}
(\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\quad dV = mapDelay(delayTransition(trans2.2)).1 \land \\
\quad mapDelay((delayTransition(trans2.2)).1) \in \operatorname{ran} tiots\_discrete) \lor \\
(\forall\, dV : t.D \bullet \exists\, trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet \\
\quad dV = mapDelay(delayTransition(trans2.2)).1 \land \\
\quad mapDelay((delayTransition(trans2.2)).1) \in \operatorname{ran} tiots\_dense)
\end{array}$$

$\Rightarrow$ *[property of sets $a = b \wedge b \in c \Rightarrow a \in c$]*

$(\forall\ dV : t.D \bullet \exists\ trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet dV \in \operatorname{ran} tiots\_discrete) \vee$
$(\forall\ dV : t.D \bullet \exists\ trans2 : d.TR \mid trans2.2 \in \operatorname{ran} del \bullet dV \in \operatorname{ran} tiots\_dense)$

$\Rightarrow$ *[propositional calculus]*

$(\forall\ dV : t.D \bullet \exists\ trans2 : d.TR \bullet dV \in \operatorname{ran} tiots\_discrete) \vee$
$(\forall\ dV : t.D \bullet \exists\ trans2 : d.TR \bullet dV \in \operatorname{ran} tiots\_dense)$

$\Leftrightarrow$ *[predicate calculus]*

$(\forall\ dV : t.D \bullet dV \in \operatorname{ran} tiots\_discrete) \vee (\forall\ dV : t.D \bullet dV \in \operatorname{ran} tiots\_dense)$

$\Leftrightarrow$ *[definition of tiots_time_compatible]*

$t.D \in tiots\_time\_compatible$

■

B.6.3 Property of the initial state

Lemma 13.

$$\forall t : TIOTS \bullet \forall d : e\_DFRS \bullet$$
$$(t = fromDFRStoTIOTS(d) \Rightarrow t.q_0 \in t.Q)$$

proof

Let t and d range over values in the sets TIOTS and e_DFRS, respectively.

$t = fromDFRStoTIOTS(d)$

$\Rightarrow$ *[definition of fromDFRStoTIOTS]*

$t = fromDFRStoTIOTS(d) \wedge$
$t.Q = getStates(t.T) \cup \{t.q_0\}$

$\Rightarrow$ *[propositional calculus]*

$t.Q \;=\; getStates(t.T) \;\cup\; \{t.q_0\}$

$\Rightarrow$ *[property of sets]*

$t.q_0 \;\in\; t.Q$

■

B.6.4 Pertinence of states

Lemma 14.

$$\begin{aligned}&\forall\, t : TIOTS \bullet \forall\, d : e\_DFRS \bullet\\ &\qquad (t = fromDFRStoTIOTS(d) \Rightarrow \forall\, entry : t.T \bullet \{entry.1, entry.3\} \subseteq t.Q)\end{aligned}$$

proof

Let t and d range over values in the sets TIOTS and e_DFRS, respectively.

$t \;=\; fromDFRStoTIOTS(d)$

$\Rightarrow$ *[definition of fromDFRStoTIOTS]*

$t \;=\; fromDFRStoTIOTS(d) \;\wedge$
$t.Q \;=\; getStates(t.T) \;\cup\; \{t.q_0\}$

$\Rightarrow$ *[propositional calculus]*

$t.Q \;=\; getStates(t.T) \;\cup\; \{t.q_0\}$

$\Rightarrow$ *[property of sets]*

$getStates(t.T) \;\subseteq\; t.Q$

$\Leftrightarrow$ *[predicate calculus – one-point rule]*

$\exists\, s : \; TIOTS\_STATE\_SET \;\bullet\; s \;=\; getStates(t.T) \;\wedge\; s \;\subseteq\; t.Q$

$\Leftrightarrow$ *[definition of getStates]*

$\exists\, s : \mathit{TIOTS\_STATE\_SET} \bullet s \subseteq t.Q \wedge s = \bigcup\{\, trans : t.T \bullet \{trans.1,\, trans.3\}\,\}$

$\Rightarrow$ *[property of set comprehension and distributed union]*

$\exists\, s : \mathit{TIOTS\_STATE\_SET} \bullet s \subseteq t.Q \wedge \forall\, entry : t.T \bullet entry.1 \in s \wedge entry.3 \in s$

$\Leftrightarrow$ *[property of sets]*

$\exists\, s : \mathit{TIOTS\_STATE\_SET} \bullet s \subseteq t.Q \wedge$
$\quad \forall\, entry : t.T \bullet entry.1 \in s \wedge entry.3 \in s \wedge entry.1 \in t.Q \wedge entry.3 \in t.Q$

$\Rightarrow$ *[propositional calculus]*

$\exists\, s : \mathit{TIOTS\_STATE\_SET} \bullet \forall\, entry : t.T \bullet entry.1 \in t.Q \wedge entry.3 \in t.Q$

$\Leftrightarrow$ *[predicate calculus]*

$\forall\, entry : t.T \bullet entry.1 \in t.Q \wedge entry.3 \in t.Q$

$\Leftrightarrow$ *[definition of $\subseteq$]*

$\forall\, entry : t.T \bullet \{entry.1,\, entry.3\,\} \subseteq t.Q$

■

B.6.5 Well typed transitions

Lemma 15.

$$\begin{array}{l} \forall t : \mathit{TIOTS} \bullet \forall d : e\_\mathit{DFRS} \bullet \\ \quad (t = \mathit{fromDFRStoTIOTS}(d) \Rightarrow \\ \qquad \forall entry : t.T \bullet (entry.2, t.I, t.O, t.D) \in \mathit{well\_typed\_tiots\_transition}) \end{array}$$

proof

Let t and d range over values in the sets TIOTS and e_DFRS, respectively.

$t = \mathit{fromDFRStoTIOTS}(d)$

$\Rightarrow$ *[defintion of fromDFRStoTIOTS]*

$$t.I \; = \; getInputActions(t.T) \; \wedge$$
$$t.O \; = \; getOutputActions(t.T) \; \wedge$$
$$t.D \; = \; getDelays(t.T)$$

$\Leftrightarrow$ *[definition of getInputActions, getOutputActions, getDelays]*

$$t.I \; = \; \{ \, trans : \; t.T \; \mid \; trans.2 \; \in \; \mathrm{ran}\; in \; \bullet \; (in^{\sim})(trans.2) \, \} \; \wedge$$
$$t.O \; = \; \{ \, trans : \; t.T \; \mid \; trans.2 \; \in \; \mathrm{ran}\; out \; \bullet \; (out^{\sim})(trans.2) \, \} \; \wedge$$
$$t.D \; = \; \{ \, trans : \; t.T \; \mid \; trans.2 \; \in \; \mathrm{ran}\; tiots\_del \; \bullet \; (tiots\_del^{\sim})(trans.2) \, \}$$

$\Rightarrow$ *[property of set comprehension]*

$$(\forall \; trans : \; t.T \; \mid \; trans.2 \; \in \; \mathrm{ran}\; in \; \bullet \; (in^{\sim})(trans.2) \; \in \; t.I) \; \wedge$$
$$(\forall \; trans : \; t.T \; \mid \; trans.2 \; \in \; \mathrm{ran}\; out \; \bullet \; (out^{\sim})(trans.2) \; \in \; t.O) \; \wedge$$
$$(\forall \; trans : \; t.T \; \mid \; trans.2 \; \in \; \mathrm{ran}\; tiots\_del \; \bullet \; (tiots\_del^{\sim})(trans.2) \; \in \; t.D)$$

$\Leftrightarrow$ *[predicate calculus]*

$$\forall \; entry : \; t.T \; \bullet$$
$$\quad (entry.2 \; \in \; \mathrm{ran}\; in \; \Rightarrow \; (in^{\sim})(entry.2) \; \in \; t.I) \; \wedge$$
$$\quad (entry.2 \; \in \; \mathrm{ran}\; out \; \Rightarrow \; (out^{\sim})(entry.2) \; \in \; t.O) \; \wedge$$
$$\quad (entry.2 \; \in \; \mathrm{ran}\; tiots\_del \; \Rightarrow \; (tiots\_del^{\sim})(entry.2) \; \in \; t.D)$$

$\Leftrightarrow$ *[definition of well_typed_tiots_transition]*

$$\forall \; entry : \; t.T \; \bullet \; (entry.2, \, t.I, \, t.O, \, t.D) \; \in \; well\_typed\_tiots\_transition$$

■

B.6.6 Proof of soundness

Theorem 3.4.1. *Soundness of fromDFRStoTIOTS*

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

proof

$$\forall \; d : \; e\_DFRS \; \bullet \; fromDFRStoTIOTS(d) \; \in \; TIOTS$$

$\Leftrightarrow$ *[predicate calculus – one-point rule]*

$$\forall\, d : e\_DFRS \bullet \exists\, t : TIOTS \bullet t = fromDFRStoTIOTS(d)$$

$\Leftrightarrow$ *[definition of TIOTS]*

$$\begin{array}{l}
\forall\, d : e\_DFRS \bullet \exists\, t : TIOTS \mid t = fromDFRStoTIOTS(d) \bullet \\
\quad \mathsf{disjoint}\ \langle\, t.I, t.O\, \rangle \wedge \\
\quad t.D \in tiots\_time\_compatible \wedge \\
\quad t.q_0 \in t.Q \wedge \\
\quad \forall\, entry : t.T \bullet \{entry.1, entry.3\} \subseteq t.Q \wedge \\
\qquad (entry.2, t.I, t.O, t.D) \in well\_typed\_tiots\_transition
\end{array}$$

$\Leftrightarrow$ *[predicate calculus – one-point rule]*

$$\begin{array}{l}
\forall\, d : e\_DFRS \bullet \forall\, t : TIOTS \bullet t = fromDFRStoTIOTS(d) \Rightarrow \\
\quad (\mathsf{disjoint}\ \langle\, t.I, t.O\, \rangle \wedge \\
\quad t.D \in tiots\_time\_compatible \wedge \\
\quad t.q_0 \in t.Q \wedge \\
\quad \forall\, entry : t.T \bullet \{entry.1, entry.3\} \subseteq t.Q \wedge \\
\qquad (entry.2, t.I, t.O, t.D) \in well\_typed\_tiots\_transition)
\end{array}$$

$\Leftrightarrow$ *[propositional calculus]*

$$\begin{array}{l}
\forall\, d : e\_DFRS \bullet \forall\, t : TIOTS \bullet \\
\quad (t = fromDFRStoTIOTS(d) \Rightarrow \mathsf{disjoint}\ \langle\, t.I, t.O\, \rangle) \wedge \\
\quad (t = fromDFRStoTIOTS(d) \Rightarrow t.D \in tiots\_time\_compatible) \wedge \\
\quad (t = fromDFRStoTIOTS(d) \Rightarrow t.q_0 \in t.Q) \wedge \\
\quad (t = fromDFRStoTIOTS(d) \Rightarrow \\
\qquad \forall\, entry : t.T \bullet \{entry.1, entry.3\} \subseteq t.Q \wedge \\
\qquad\quad (entry.2, t.I, t.O, t.D) \in well\_typed\_tiots\_transition)
\end{array}$$

$\Leftrightarrow$ [predicate calculus]

$$\begin{array}{l}\forall\, t : TIOTS \bullet \forall\, d : e\_DFRS \bullet \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow \mathsf{disjoint}\,\langle\, t.I,\, t.O\,\rangle) \land \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow t.D \in tiots\_time\_compatible) \land \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow t.q_0 \in t.Q) \land \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow \\ \qquad \forall\, entry : t.T \bullet \{entry.1,\, entry.3\} \subseteq t.Q \land \\ \qquad\quad (entry.2,\, t.I,\, t.O,\, t.D) \in well\_typed\_tiots\_transition)\end{array}$$

$\Leftrightarrow$ [Lemma 9]

$$\begin{array}{l}\forall\, t : TIOTS \bullet \forall\, d : e\_DFRS \bullet \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow t.D \in tiots\_time\_compatible) \land \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow t.q_0 \in t.Q) \land \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow \\ \qquad \forall\, entry : t.T \bullet \{entry.1,\, entry.3\} \subseteq t.Q \land \\ \qquad\quad (entry.2,\, t.I,\, t.O,\, t.D) \in well\_typed\_tiots\_transition)\end{array}$$

$\Leftrightarrow$ [Lemma 12]

$$\begin{array}{l}\forall\, t : TIOTS \bullet \forall\, d : e\_DFRS \bullet \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow t.q_0 \in t.Q) \land \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow \\ \qquad \forall\, entry : t.T \bullet \{entry.1,\, entry.3\} \subseteq t.Q \land \\ \qquad\quad (entry.2,\, t.I,\, t.O,\, t.D) \in well\_typed\_tiots\_transition)\end{array}$$

$\Leftrightarrow$ [Lemma 13]

$$\begin{array}{l}\forall\, t : TIOTS \bullet \forall\, d : e\_DFRS \bullet \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow \\ \qquad \forall\, entry : t.T \bullet \{entry.1,\, entry.3\} \subseteq t.Q \land \\ \qquad\quad (entry.2,\, t.I,\, t.O,\, t.D) \in well\_typed\_tiots\_transition)\end{array}$$

$\Leftrightarrow$ [Lemma 14]

$$\begin{array}{l}\forall\, t : TIOTS \bullet \forall\, d : e\_DFRS \bullet \\ \quad (t = fromDFRStoTIOTS(d) \Rightarrow \\ \qquad \forall\, entry : t.T \bullet (entry.2,\, t.I,\, t.O,\, t.D) \in well\_typed\_tiots\_transition)\end{array}$$

$\Leftrightarrow$ *[Lemma 15]*

true

■

C

CSP-based testing theory

C.1 CSP$_M$ specification for the vending machine

Code C.1: CSP$_M$ specification (vending machine)

```
the_system_mode_values = {0, 1, 2, 3}
the_coffee_machine_output_values = {0, 1}

range(I_the_system_mode) =
  {I_the_system_mode.0, I_the_system_mode.1,
   I_the_system_mode.2, I_the_system_mode.3}

range(I_the_coffee_machine_output) =
  {I_the_coffee_machine_output.0, I_the_coffee_machine_output.1}

range(B) = {B.false, B.true}

tag(I_the_system_mode._) = I_the_system_mode
tag(I_the_coffee_machine_output._) = I_the_coffee_machine_output
tag(B._) = B

datatype TYPE = I_the_system_mode.the_system_mode_values |
  I_the_coffee_machine_output.the_coffee_machine_output_values |
  B.Bool

datatype VAR = the_coffee_request_button | the_coin_sensor |
  the_system_mode | the_coffee_machine_output |
  old_the_coffee_request_button | old_the_coin_sensor |
  funTrans | the_request_timer |
  eta1 | eta2 | eta3 | eta4 | gc

initialBinding = {
  (the_coffee_request_button, B.false),
  (the_coin_sensor, B.false),
  (the_system_mode, I_the_system_mode.1),
  (the_coffee_machine_output, I_the_coffee_machine_output.0),
  (old_the_coffee_request_button , B.false),
  (old_the_coin_sensor , B.false)
  (funTrans , B.false),
  (the_request_timer , B.false),
  (eta1 , B.false),
  (eta2 , B.false),
  (eta3 , B.false),
  (eta4 , B.false)
}

channel get, set: VAR.TYPE
MCELL(var,val) = get!var!val -> MCELL(var,val)
        [] set!var?val' : range(tag(val)) -> MCELL(var,val')
MEMORY(binding) = ||| (var,val) : binding @ MCELL(var,val)

SYSTEM_MEMORY = MEMORY(initialBinding)

channel maxDelay: VAR.TYPE.VAR.TYPE.VAR.TYPE.VAR.TYPE
channel delay: VAR.TYPE.VAR.TYPE.VAR.TYPE.VAR.TYPE

channel input: VAR.TYPE.VAR.TYPE
channel output: VAR.TYPE.VAR.TYPE

channel c_the_coffee_request_button:Bool
channel c_the_coin_sensor:Bool

INPUTS =
  c_the_coffee_request_button?newV_the_coffee_request_button ->
  get!the_coffee_request_button?B.v_the_coffee_request_button ->
  set!old_the_coffee_request_button!B.v_the_coffee_request_button ->
  set!the_coffee_request_button!B.newV_the_coffee_request_button ->
  c_the_coin_sensor?newV_the_coin_sensor ->
  get!the_coin_sensor?B.v_the_coin_sensor ->
  set!old_the_coin_sensor!B.v_the_coin_sensor ->
  set!the_coin_sensor!B.newV_the_coin_sensor ->
  input.the_coffee_request_button.B.newV_the_coffee_request_button
       .the_coin_sensor.B.newV_the_coin_sensor -> SKIP

channel REQ003, REQ005, REQ001, REQ002, REQ004

channel reset : {the_request_timer}

SYSTEM_BEHAVIOUR(v_old_the_coffee_request_button ,
  v_the_coffee_request_button , v_old_the_coin_sensor ,
  v_the_coin_sensor , v_the_system_mode , v_eta1 ,
  v_the_request_timer , v_eta2 , v_the_coffee_machine_output ,
  v_eta3 , v_eta4) =
    (((((not((v_old_the_coffee_request_button == true)) and
       (v_the_coffee_request_button == true)) and
       (v_old_the_coin_sensor == false)) and
       (v_the_coin_sensor == false)) and
       (v_the_system_mode == 0)) and v_eta1 and
       (v_the_request_timer != true or v_the_system_mode != 2) &
        set!funTrans!B.true -> reset.the_request_timer ->
        set!the_request_timer!B.true ->
        set!the_system_mode!I_the_system_mode.2 -> REQ003 -> SKIP)
      []
      ((v_the_system_mode == 2) and v_eta2 and
       (v_the_system_mode != 1 or v_the_coffee_machine_output != 0) &
        set!funTrans!B.true ->
        set!the_system_mode!I_the_system_mode.1 ->
        set!the_coffee_machine_output!I_the_coffee_machine_output.0 ->
        REQ005 -> SKIP)
      []
      (((not((v_old_the_coin_sensor == true)) and
       (v_the_coin_sensor == true)) and (v_the_system_mode == 1)) and
       (v_the_request_timer != true or v_the_system_mode != 0) &
        set!funTrans!B.true -> reset.the_request_timer ->
        set!the_request_timer!B.true ->
        set!the_system_mode!I_the_system_mode.0 -> REQ001 -> SKIP)
      []
      (((((not((v_old_the_coffee_request_button == true)) and
       (v_the_coffee_request_button == true)) and
       (v_old_the_coin_sensor == false)) and
       (v_the_coin_sensor == false)) and
       (v_the_system_mode == 0)) and v_eta3 and
       (v_the_request_timer != true or v_the_system_mode != 3) &
        set!funTrans!B.true -> reset.the_request_timer ->
        set!the_request_timer!B.true ->
        set!the_system_mode!I_the_system_mode.3 -> REQ002 -> SKIP)
      []
      ((v_the_system_mode == 3) and v_eta4 and
       (v_the_system_mode != 1 or v_the_coffee_machine_output != 1) &
        set!funTrans!B.true ->
        set!the_system_mode!I_the_system_mode.1 ->
        set!the_coffee_machine_output!I_the_coffee_machine_output.1 ->
        REQ004 -> SKIP)
      []
      (not(((((not((v_old_the_coffee_request_button == true)) and
       (v_the_coffee_request_button == true)) and
       (v_old_the_coin_sensor == false)) and
       (v_the_coin_sensor == false)) and
       (v_the_system_mode == 0)) and v_eta1 and
       (v_the_request_timer != true or v_the_system_mode != 2)) and
       not((v_the_system_mode == 2) and v_eta2 and
       (v_the_system_mode != 1 or v_the_coffee_machine_output != 0)) and
       not(((not((v_old_the_coin_sensor == true)) and
       (v_the_coin_sensor == true)) and (v_the_system_mode == 1)) and
       (v_the_request_timer != true or v_the_system_mode != 0)) and
       not(((((not((v_old_the_coffee_request_button == true)) and
       (v_the_coffee_request_button == true)) and
       (v_old_the_coin_sensor == false)) and
       (v_the_coin_sensor == false)) and
       (v_the_system_mode == 0)) and v_eta3 and
       (v_the_request_timer != true or v_the_system_mode != 3)) and
       not((v_the_system_mode == 3) and v_eta4 and
       (v_the_system_mode != 1 or v_the_coffee_machine_output != 1)) &
        SKIP)

 DELAY(delayChannel) =
    set!eta4!B.false ->
    set!eta3!B.false ->
    set!eta2!B.false ->
    set!eta1!B.false ->
    get!old_the_coffee_request_button?
      B.v_old_the_coffee_request_button ->
    get!the_coffee_request_button?
      B.v_the_coffee_request_button ->
    get!old_the_coin_sensor?B.v_old_the_coin_sensor ->
    get!the_coin_sensor?B.v_the_coin_sensor ->
    get!the_system_mode?I_the_system_mode.v_the_system_mode ->
    (
      (((((not((v_old_the_coffee_request_button == true)) and
       (v_the_coffee_request_button == true)) and
       (v_old_the_coin_sensor == false)) and
       (v_the_coin_sensor == false)) and (v_the_system_mode == 0)) &
        set!eta1!B.true -> SKIP)
      []
      ((v_the_system_mode == 2) & set!eta2!B.true -> SKIP)
      []
      (((((not((v_old_the_coffee_request_button == true)) and
       (v_the_coffee_request_button == true)) and
       (v_old_the_coin_sensor == false)) and
       (v_the_coin_sensor == false)) and (v_the_system_mode == 0)) &
        set!eta3!B.true -> SKIP)
      []
      ((v_the_system_mode == 3) & set!eta4!B.true -> SKIP)
      []
      (not(((((not((v_old_the_coffee_request_button == true)) and
       (v_the_coffee_request_button == true)) and
       (v_old_the_coin_sensor == false)) and
       (v_the_coin_sensor == false)) and
       (v_the_system_mode == 0))) and not((v_the_system_mode == 2)) and
       not(((((not((v_old_the_coffee_request_button == true)) and
       (v_the_coffee_request_button == true)) and
       (v_old_the_coin_sensor == false)) and
       (v_the_coin_sensor == false)) and (v_the_system_mode == 0))) and
       not((v_the_system_mode == 3)) &
        SKIP)
   )
   ; get!eta1?v_eta1 -> get!eta2?v_eta2 -> get!eta3?v_eta3 ->
     get!eta4?v_eta4 ->
     delayChannel.eta1.v_eta1.eta2.v_eta2.eta3.v_eta3.eta4.v_eta4 ->
     SKIP

 FUN_TRANS =
   get!eta1?B.v_eta1 ->
   get!eta2?B.v_eta2 ->
   get!eta3?B.v_eta3 ->
   get!eta4?B.v_eta4 ->
   get!old_the_coffee_request_button?
     B.v_old_the_coffee_request_button ->
   get!the_coffee_request_button?
     B.v_the_coffee_request_button ->
   get!old_the_coin_sensor?B.v_old_the_coin_sensor ->
   get!the_coin_sensor?B.v_the_coin_sensor ->
   get!the_system_mode?I_the_system_mode.v_the_system_mode ->
   get!eta1?B.v_eta1 ->
   get!the_request_timer?B.v_the_request_timer ->
   get!eta2?B.v_eta2 ->
   get!the_coffee_machine_output?
     I_the_coffee_machine_output.v_the_coffee_machine_output ->
   get!eta3?B.v_eta3 ->
   get!eta4?B.v_eta4 ->
   SYSTEM_BEHAVIOUR(v_old_the_coffee_request_button,
     v_the_coffee_request_button, v_old_the_coin_sensor,
     v_the_coin_sensor, v_the_system_mode, v_eta1,
     v_the_request_timer, v_eta2,
     v_the_coffee_machine_output, v_eta3, v_eta4)

 FUN = set!funTrans!B.false ->
   FUN_TRANS
   ; get!funTrans?B.engaged ->
   if engaged then FUN else OUTPUTS

 OUTPUTS =
   get!the_system_mode?v_the_system_mode ->
   get!the_coffee_machine_output?v_the_coffee_machine_output ->
   output.the_system_mode.v_the_system_mode
          .the_coffee_machine_output.v_the_coffee_machine_output ->
   SKIP

```

```
channel stableState, delayTransition

SPECIFICATION =
   set!the_request_timer!B.false ->
  FUN ; stableState ->
   delayTransition -> DELAY(maxDelay) ; INPUTS ; DELAY(delay)
   ; SPECIFICATION

SYSTEM = SPECIFICATION [| {|get,set|} |] SYSTEM_MEMORY
```

[Source: author]