Formal Modelling of Environment Restrictions from Natural-Language Requirements

Tainã Santos[1], Gustavo Carvalho[2(✉)], and Augusto Sampaio[2]

[1] Universidade de Pernambuco - Escola Politécnica de Pernambuco, Recife 50720-001, Brazil
tms@ecomp.poli.br

[2] Universidade Federal de Pernambuco - Centro de Informática, Recife 50740-560, Brazil
{ghpc,acas}@cin.ufpe.br

Abstract. When creating system models, further to system behaviour one should take into account properties of the environment in order to achieve more meaningful models. Here, we extend a strategy that formalises data-flow reactive systems as CSP processes to take into account environment restrictions. Initially, these restrictions are written in natural language. Afterwards, with the aid of case-grammar theory, they are formalised by deriving LTL formulae automatically. Finally, these formulae are used to prune infeasible scenarios from the CSP-based system specification, in the light of the environment restrictions. Considering examples from the literature, and from the aerospace (Embraer) and the automotive (Mercedes) industry, we show the efficacy of our proposal in terms of state space reduction, up to 61% in some cases.

Keywords: Natural language · Environment restrictions
Case grammar · Linear temporal logic
Communicating Sequential Processes

1 Introduction

A central element when applying formal methods is capturing the system behaviour precisely, which is typically modelled using some formal notation. Besides modelling the system behaviour, it is also relevant to consider its environment. Although some interactions are possible when only considering the system model, they might not be feasible in practice due to characteristics of the environment. For example, considering a control system operating the car turn lights, in the presence of a turn indicator lever, the control system might not capture a direct change from left flashing to right flashing, since the lever cannot change directly between its extreme positions; it must reside for some moment in the neutral position.

© Springer Nature Switzerland AG 2018
T. Massoni and M. R. Mousavi (Eds.): SBMF 2018, LNCS 11254, pp. 252–270, 2018.
https://doi.org/10.1007/978-3-030-03044-5_16

Here, we consider as environment the collection of entities that interact with the system being modelled. In the aforementioned example, the turn indicator lever would be part of the environment that interacts with the control system operating the car turn lights. Therefore, in order to develop more meaningful models, it is also recommended to take into consideration properties of the environment that, for instance, restrict how the user interacts with the system. In this way, unrealistic interactions are not considered by the models, which tends to reduce the overall system state space. This is more widely beneficial for model checking, simulation, testing and the final system implementation and deployment. For example, when applying model-based testing strategies, infeasible test cases, which cannot be performed due to environment restrictions, are not derived from models.

In this work, we define a controlled natural language (CNL) for specifying restrictions on how a system interacts with its environment. There is a trade-off concerning the adoption of a CNL for requirements specification: one can use a low-constrained CNL to enforce general writing styles, but, typically, formal analysis is not possible by automatic means; on the other extreme, one can adopt a highly-constrained CNL that enables automatic reasoning at the expense of writing naturalness. We seek a compromise between these two extremes: our CNL enforces enough structure to allow for automatic processing of environment restrictions, but aiming at not losing naturalness.

After specifying the restrictions adhering to our CNL, we derive LTL formulae to formalise these restrictions. These formulae are then used to prune, from the specification model, defined using the process algebra CSP (Communicating Sequential Processes) [16], infeasible scenarios in the light of the restrictions. We propose two approaches for restricting CSP models: the first one is based on filtering the inputs, by checking, via a monitor process, which ones satisfy the environment restrictions; the second one involves syntactically modifying the specification so that only valid inputs are selected, but this is done by the process itself, rather than by another process like in the first approach.

Formal modelling of environments is addressed, for instance, in [9,14], where a model is created to capture how the test environment interacts with the system. In our work, the model of the environment restrictions is automatically derived from natural-language descriptions, which are formalised by LTL formulae. Previous works, such as [12], also define ways of generating LTL formulae from natural language, but we differ from them since our formulae are defined over variables and values (not event-based), which enables an easier and more natural way of writing expressions (e.g., one can write $x > 10$, instead of writing $x_{11} \vee x_{12} \vee x_{13}$ to denote the events representing all values x can have that are greater than 10; here, assuming that the greatest possible value of x is 13).

The strategy for modelling environment restrictions presented here is part of a broader research effort for generating test cases from natural-language requirements: the NAT2TEST strategy. In [4], we describe how models of data-flow reactive systems (DFRSs) are automatically derived from controlled natural-language specifications of system requirements. Afterwards, different formal

notations can be used to represent models of DFRSs, such as the process algebra CSP, allowing the exploitation of different techniques and tools. In [3] we describe tool support for this strategy: the NAT2TEST tool. A comprehensive explanation of this strategy is presented in [2].

Our strategy for modelling environment restrictions was integrated into the NAT2TEST$_{\text{CSP}}$, a version of the NAT2TEST strategy that uses CSP, and, considering examples from the literature, and from the aerospace (Embraer) and the automotive (Mercedes) industry, we show the efficacy of our proposal in terms of state space reduction (up to 61% in some cases). Therefore, the main contributions of this work are the following:

- A CNL for describing environment restrictions;
- A strategy for formalising environment restrictions as LTL formulae;
- Two approaches for imposing environment restrictions on CSP models;
- Integration of this work into the NAT2TEST$_{\text{CSP}}$ strategy;
- Empirical analyses concerning examples from the literature and the industry.

This paper is organised as follows. Section 2 briefly introduces background material: linear temporal logic, the process algebra CSP, and modelling data-flow reactive systems as CSP processes. Section 3 presents our CNL for specifying environment restrictions, and explains how LTL formulae are automatically derived from specifications in CNL. Section 4 details the two approaches for imposing environment restrictions on CSP models of the system behaviour. Section 5 gives empirical evidence on the efficacy of our proposal. Finally, Sect. 6 presents our conclusions, and addresses related and future work.

2 Preliminaries

In this section we present an overview of the related background: linear temporal logic (Sect. 2.1), besides the process algebra CSP (Sect. 2.2), which is used to represent the behaviour of data-flow reactive systems (Sect. 2.3).

2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) [15] is a logic for reasoning about linear-time temporal propositions. Given an alphabet Σ of elementary propositions (denoted by lower-case letters), the syntax of LTL is given by the following grammar:

$$\phi ::= \mathit{false} \mid \mathit{true} \mid a \mid \neg\, \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \bigcirc\, \phi \mid \phi\, \mathcal{U}\, \phi \mid \phi\, \mathcal{R}\, \phi$$

Classically, $\bigcirc$ is the next operator (ϕ holds in the next state), $\mathcal{U}$ is the until operator ($\phi\, \mathcal{U}\, \psi$ means that for every execution of the system the formula ψ must eventually become true and the formula ϕ must be true until, not necessarily including, the first point at which ψ becomes true), and $\mathcal{R}$ is the release operator (the dual of $\mathcal{U}$). Two other derived operators are the eventually and the always operators: $\Diamond\phi \equiv \mathit{true}\ \mathcal{U}\ \phi$ and $\Box\phi \equiv \neg\ \Diamond\neg\ \phi$, respectively. The formula $\Diamond\phi$

means that, for every execution of the system, ϕ must hold for some state in the future, whereas $\Box\phi$ means that, for every execution, ϕ holds for all states. In this work, we also consider the weak-until operator: $\phi \mathcal{W} \psi \equiv (\phi \mathcal{U} \psi) \vee \Box\phi$, where ψ is not required to occur.

A practical application of LTL is to formalise properties of systems. However, as discussed in [5], it is not always straightforward to define a formula that correctly captures the intended behaviour. In order to make this task easier, a repository[1] was developed to collect patterns that commonly occur in the specification of concurrent and reactive systems. These patterns also have an application scope. Here, our restrictions fit the *absence* and the *universality* patterns, considering the *global*, the *after* and the *after-until* application scopes.

As noted in [11], when considering LTL formulae in the context of CSP specifications, we need to assume an adapted interpretation of the classical LTL operators, since LTL is usually defined for state-based models while the operational semantics of CSP is defined in terms of labelled transition systems (labels are associated to transitions and not states; moreover, some transitions are labelled by the invisible action τ). We follow [11] in this respect.

2.2 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 (traces semantics), a process behaviour is described by the set of sequences of events it can perform. To define a process as a sequence of events, we use the prefix operator ($P = ev \rightarrow Q$), where ev is an event, and P and Q are processes.

The 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 parallel composition, CSP allows a composition with ($\|$) or without ($|||$) synchronisation between the composed processes. CSP processes synchronise between themselves by means of events. For instance, $P \underset{X}{\|} Q$ requires synchronisation on the events in X.

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. It is also possible to interpret these symbols ($!$ and $?$) as a process sending or receiving a value through a channel, respectively. Another CSP operator used in this work is hiding ($\backslash$): it encapsulates events within a process and, thus, makes them internal (represented as τ). CSP also has a functional language for manipulating local data.

[1] http://patterns.projects.cs.ksu.edu/.

From a CSP specification written in its machine-readable version called CSP_M, the FDR tool[2] [8] 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*).

2.3 DFRSs as CSP Processes

The NAT2TEST strategy generates test cases fully automatically from natural-language requirements [3]. The data-flow reactive system (DFRS) model serves as an intermediate formal notation from which it is possible to generate models in several formal target notations, such as CSP. As detailed in [4], any DFRS can be encoded as a Timed Input-Output Transition System (TIOTS), a labelled transition system extended with time, which is widely used to characterise conformance relations for timed reactive systems. However, being more abstract, a DFRS comprises a more concise representation of timed requirements.

Here, we are interested on the CSP-based specialisation of the NAT2TEST strategy ($\text{NAT2TEST}_\text{CSP}$), since it provides us with a sound testing theory. Test generation is mechanised in terms of a high-level strategy by reusing successful techniques and tools: refinement checking (FDR) and SMT solving (Z3[3]). More information is available in [2]. Nevertheless, our results on formal modelling of environment restrictions can also be applied to other strategies, taking as starting point the LTL formulae automatically derived from the textual descriptions.

In what follows, we present a concise explanation of DFRS models, and how they are encoded as CSP processes. A DFRS model represents 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. A DFRS can also have internal timers, which can be used to trigger time-based behaviour.

In the CSP notation, the system behaviour is denoted by the process S, which is defined as $SYSTEM$, hiding all of its internal events (only events related to input, output and time behaviour are visible).

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

The process $SYSTEM\_MEMORY$ is defined to allow the parallel components of the system to communicate via shared memory (i.e., global variables, which are not directly supported by CSP). The process $SPECIFICATION$ interacts with the memory reading and writing values via the channels *get* and *set*, respectively; $\{|\ c\ |\}$ represents all values that can be communicated over the channel c (e.g., $\{|\ get, set\ |\}$ denotes all events communicated over the channels *get* and *set*).

[2] https://www.cs.ox.ac.uk/projects/fdr/.
[3] https://github.com/Z3Prover/z3.

A DFRS model has delay and function transitions. The former occur when the system is in a stable state (no system reaction is enabled, and time might evolve), whereas the latter occur when the state is not stable (system reacts to input stimuli). The process *SPECIFICATION* captures this behaviour.

$$SPECIFICATION = \\ ... \rightarrow FUN \ ; \ ... \rightarrow INPUTS \ ; \ \ DELAY \ ; \ SPECIFICATION$$

The first events (not shown) are related to a symbolic encoding of time in CSP, which enables the representation of discrete and continuous time using the standard CSP notation. Since explaining the details of this codification is outside the scope of this paper, we refer to [2] for further details.

After performing these first events, this process behaves as *FUN*, which performs function transitions until the system reaches a stable state, when an output event is performed over the channel *output*. Afterwards, the system performs a delay transition. Basically, time evolves (represented by the *DELAY* process) and new inputs are received (process *INPUTS*) over the channel *input*. In the CSP definition, *INPUTS* takes place before *DELAY* as a consequence of our symbolic time representation – see [2] for more details. Then the process recurses. Therefore, when we analyse a trace of S we observe an alternating sequence of time, input, and output-related events; representing time elapsing, system stimuli and system reaction, respectively.

3 Environment Restrictions

In this section we define a CNL that is convenient to capture restrictions on the environment (Sect. 3.1) and, with the support of case-grammar theory [7] (Sect. 3.2), we devise an automatic translation into LTL formulae (Sect. 3.3). To illustrate our ideas, we consider an adaptation of the vending machine (VM) presented in [9]. We also refer to a Mercedes' turn indicator system (TIS) to illustrate some specific features (explained on demand).

Initially, the VM is in the *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 depending on the time elapsed since the coin insertion. After producing coffee, the system goes back to the *idle* state.

3.1 A CNL for Environment Restrictions

An environment restriction can be seen as the description of an interaction between the environment and the system that is not allowed to happen. It describes input scenarios that are not feasible in practice. The grammar of our CNL for specifying environment restrictions (EnvReq-CNL) is given in Table 1.

The EnvReq-CNL allows for the specification of restrictions that fit the *absence* and *universality* property patterns, considering *global*, *after* and *after-until* application scopes (see [5] for more details on LTL property patterns). The

Table 1. The EnvReq-CNL grammar

```
TestEnvRestriction    ::= (NEVER | ALWAYS) Scope?
                          (StatementClause | ImplicationClause)
Scope                 ::= AFTER AndCondition, (AND UNTIL AndCondition,)?
AndCondition          ::= ...
StatementClause       ::= AndCondition
ImplicationClause     ::= ConditionalClause COMMA THEN ConsequenceClause
ConditionalClause     ::= CONJ AndCondition
ConsequenceClause     ::= RestrictionOrClause
                          | COLON RestrictionOrClause
                            (COMMA AND RestrictionAndClause)+
RestrictionAndClause  ::= RestrictionOrClause
                          | RestrictionAndClause
                            COMMA AND RestrictionOrClause
RestrictionOrClause   ::= RestrictionClause
                          | RestrictionOrClause OR RestrictionClause
RestrictionClause     ::= NounPhrase VerbPhraseRestriction
NounPhrase            ::= ...
VerbPhraseRestriction ::= VerbRestriction VerbComplement
VerbRestriction       ::= (CNOT | CONLY) VBASE
VerbComplement        ::= ...
```

terminal symbols *NEVER* and *ALWAYS* are mapped to the words "*It is never the case that*" (*absence* pattern) and "*It is always the case that*" (*universality pattern*). After these words, one can specify the application scope (*global* is the default one), followed by the restriction as a statement (*StatementClause*) or as an implication (*ImplicationClause*) clause.

A statement comprises clauses according to a conjunctive normal form (CNF): this structure is ensured by the symbol *AndCondition*. An implication clause is composed by a conditional clause, whose structure is also a CNF preceded by a conjunction, followed by a consequence clause. The consequence clause (also a CNF) describes something that shall be performed (*CONLY*) or cannot be performed (*CNOT*) by the environment. Therefore, this grammar allows for the specification of restrictions in one of the following four templates.

T1 — It is always the case that S, C.
T2 — It is never the case that S, C.
T3 — It is always the case that S, when C_1 then C_2.
T4 — It is never the case that S, when C_1 then C_2.

The symbol S denotes a scope, and if absent it means the global one. The symbol C denotes conditions describing restrictions on the environment. As it can be seen, T2 is the dual of T1; and T3/T4 can be rewritten as T1/T2, respectively, using classical transformations ($C_1 \Rightarrow C_2 \equiv \neg\ C_1 \vee C_2$). Nevertheless, we permit different writing styles aiming at flexibility.

To illustrate our CNL for environment restrictions, consider the VM example. Suppose that a coin can only be inserted when the system is in the idle

state: when it is waiting for a coffee request or it is producing a weak (strong) coffee, some mechanical device blocks the hole where the coin should be inserted. The following sentence describes this restriction in accordance to the grammar previously presented. This sentence adheres to T3, considering a global scope.

– VM-RST001: It is always the case that when the system mode is not idle, then the coin sensor cannot be true.

Similarly, suppose that the coffee request button can only be pressed when the system is expecting such an input from the user (the system mode is *choice*). This restriction can be described as follows (template T2 and global scope).

– VM-RST002: It is never the case that the coffee request button is pressed, and the system mode is not choice.

The restrictions can also refer to the previous value of input and output variables. To give a concrete example, consider the following restriction related to the turn indicator system (TIS) of Mercedes vehicles (made available by Daimler; more information in Sect. 5). It is not possible to move the turn indicator lever from the left position directly to the right position. It is necessary to move the lever to the neutral position first. The following sentence (TIS-RST001) specifies this restriction according to the grammar of EnvReq-CNL.

– It is always the case that when the turn indicator lever was in the left position, then the turn indicator lever cannot change to the right position.

The analysis whether the sentences adhere to the EnvReq-CNL is performed by the CNL-Parser, which is part of the NAT2TEST tool [3]. To integrate our work to the NAT2TEST strategy, we modify its CNL (i.e., SysReq-CNL) in order to allow for the specification of both system requirements and environment restrictions. This is achieved by updating the rewriting rule of the start symbol (*Sentence*) as follows. Here, *Sentence* is the start symbol of the NAT2TEST context-free grammar (SysReq-CNL) for specifying system requirements, which can now be rewritten as a system requirement (the non-terminal symbol *SysRequirement*), but also as a restriction on the test environment (the non-terminal symbol *TestEnvRestriction* – see Table 1). More details about the SysReq-CNL are available in [2].

```
Sentence ::= SysRequirement | TestEnvRestriction
```

Before generating the corresponding LTL formulae, we automatically extract requirement frames from the syntax trees of the restrictions. This additional step is performed to decouple the generation of LTL formulae from the structure of the CNL, besides making easier the LTL generation step.

3.2 From Syntax Trees to Requirement Frames

The case-grammar theory [7] is a linguistic theory that can be used to provide semantics to natural-language requirements. In this theory, a sentence is analysed

in terms of the thematic roles (TR) played by each word, or group of words in the sentence. The verb is the main element of the sentence, and it determines its possible semantic relations with the other words, that is, the role that each word plays with respect to the action or state described by the verb.

The verb's associated TRs are aggregated into a structure named as case frame (CF). Each verb in a requirement (describing an environment restriction) gives rise to a different CF. All derived CFs are joined afterwards to compose a *Requirement Frame* (RF). Additionally, a RF also has information about the application scope of the restriction. In this work, we consider five thematic roles: the condition action (CAC – the verb related to the condition), the condition patient (CPT – entity who is referred by the condition verb), the condition modifier (CMD – a modification applied to the condition verb), and the condition from/to value (CFV, CTV – values associated to the condition patient). For instance, Table 2 shows the RF obtained from VM-RST001.

Table 2. Requirement frame of VM-RST001

Scope: global			
Condition 1: main verb (CAC): is			
CPT:	the system mode	CFV:	–
CMD:	not	CTV:	idle
Restriction 1: main verb (CAC): be			
CPT:	the coin sensor	CFV:	–
CMD:	cannot	CTV:	true

In order to infer the requirement frame of a given restriction we apply inference rules, which map parts of the CNL structure to thematic roles. The description of these inference rules is outside the scope of this paper.

3.3 From Requirement Frames to LTL

After identifying the requirement frames, we formalise the environment restrictions by generating LTL formulae. First, we identify the core formula (π), which is derived from conditions C or C_1 and C_2 (see Table 3). The symbols ϕ and ψ refer to conditions described by C (or C_1) and C_2, respectively.

Afterwards, we conclude the generation of the LTL formula by considering the application scope (see the correspondence in what follows). The symbol γ refers to the conditions associated to the *after* application scope, and ω to the conditions of the *until* clause (if present). The symbol π denotes the core formula, previously identified.

- Global scope: $\Box(\pi)$
- After scope: $\Box(\gamma \Rightarrow \Box(\pi))$
- After-until scope: $\Box(\gamma \wedge \neg\, \omega \Rightarrow \Box(\pi\ \mathcal{W}\ \omega))$

Table 3. Mapping writing templates to LTL: core formula

Template	Text	Core formula (π)
T1	It is always the case that S, C	ϕ
T2	It is never the case that S, C	$\neg\ \phi$
T3	It is always the case that S, when C_1 then C_2	$\phi \Rightarrow \psi$
T4	It is never the case that S, when C_1 then C_2	$\neg\ (\phi \Rightarrow \psi)$

If the application scope is *global*, the formula is preceded by a single $\Box$ operator. When considering an *after* scope, the restriction applies globally only after γ holds. Similarly, regarding the *after-until* scope, the restriction applies globally after γ holds, but until ω holds, which might never occur.

After identifying the general outline of our formulae, we use the thematic roles to generate γ and ω from S, and ϕ and ψ from C (or C_1) and C_2, respectively. The generation of boolean expressions from thematic roles is similar to the one described in [4]. The condition patients (CPT) turn into variables, while their values are extracted from the roles condition from/to value (CFV and CTV). The verbs (CAC) and modifiers (CMD) are used to determine the associated boolean operators. Algorithm 1 summarises the process for generating LTL formulae from requirement frames.

Algorithm 1. *generateLTLFormulae*

input : *reqFrames*
output : *ltlFormulae*

for *reqFrame* $\in$ *reqFrames* **do**
  $\gamma, \omega, \phi, \psi \leftarrow$ *generateBooleanExpressions*(*reqFrame*);
  $\pi \leftarrow$ *mapWritingTemplateToLTL*(ϕ, ψ, *reqFrame*);
  if *identifyScope*(*reqFrame*) = *global* **then**
    lftlFormulae.add($\Box(\pi)$);
  else if *identifyScope*(*reqFrame*) = *after* **then**
    lftlFormulae.add($\Box(\gamma \Rightarrow \Box(\pi))$);
  else
    lftlFormulae.add($\Box(\gamma \wedge \neg\ \omega \Rightarrow \Box(\pi\ \mathcal{W}\ \omega))$);

For instance, considering the roles presented in Table 2 for *Condition 1*, we have that *the_system_mode* (CPT) turns out to be a variable whose value is (CAC) not (CMD) equal to *idle* (CTV) (i.e., *the_system_mode* $\neq$ 1). We note that the value 1 is used to represent the value *idle*. When performing these translations, our tool automatically represents string values as enumeration values. Concerning *Restriction 1*, *the_coin_sensor* is another variable (CPT) whose value cannot (CMD) be (CAC) true (CTV), i.e., *the_coin_sensor* $\neq$ *true*. Filling these expressions into the LTL formula associated to T3 (the template used

in VM-RST001), and considering its global scope, we have the following LTL formula: $\Box(the\_system\_mode \neq 1 \Rightarrow the\_coin\_sensor \neq true)$.

The LTL formulae derived for the other environment restrictions previously presented (VM-RST002 and TIS-RST001) are the following, respectively.

$$\Box(\neg\ (the\_coffee\_request\_button = true \wedge the\_system\_mode \neq 0))$$
$$\Box(old\_the\_turn\_indicator\_lever = 1 \Rightarrow the\_turn\_indicator\_lever \neq 2)$$

In the first formula, being pressed is represented as *true* (an optimisation automatically performed when the possible string values are s and $\neg\ s$ – the former is treated as *true*, and the latter as *false*). Concerning the system mode, the value 0 represents the *choice* state.

Regarding the turn indicator system, the positions of the turn indicator lever (*the neutral position*, *the left position*, and *the right position*) are represented by the values 0, 1, and 2, respectively. It is also important to note that the *old_* prefix is used to refer to the previous value of a variable.

The next step of our strategy is to impose the environment restrictions (represented as LTL formulae) to the CSP specification of the system. We do not translate from the environment restrictions (in natural language) directly to CSP to make our strategy extensible to other situations when the system behaviour is not being modelled as CSP processes. In such situations, the effort to apply our strategy would be to define a translation between LTL formulae and the adopted formalism to represent the system behaviour.

4 Imposing Restrictions

After obtaining the LTL formulae from the natural-language descriptions of the environment restrictions, the next step is to consider them to constrain the CSP model of the system, which is automatically derived from the system requirements by the NAT2TEST$_{\text{CSP}}$ strategy; therefore, we emphasise that the CSP model of the system is also generated from a controlled natural-language specification of the system behaviour (more details in [2]). In the following sections we propose two different approaches for enforcing the test environment restrictions.

As already mentioned, the first one (Sect. 4.1) imposes the restrictions by filtering the inputs that obey the environment restrictions; this is captured by a monitor process. The effect of pruning is achieved by composing the original system model in CSP in parallel with this monitor. In this way, the original CSP system specification is totally preserved. Differently, the second approach (Sect. 4.2) modifies the original CSP model so that only valid inputs are selected. In addition to reducing the system state space, this approach has the additional advantage of producing a simpler CSP model that requires less time to compile. On the other hand, it is not compositional.

4.1 Approach 1: Monitoring Input Generation

In this approach, a monitor process deadlocks (prohibits the system process to advance) under undesired scenarios. Considering the VM example, part of this

monitoring is performed by the *CHECK_RST* process (shown below); it deadlocks (*STOP*) when at least one of the restrictions is violated. The expressions in the if-clause are derived from the corresponding LTL formulae (see requirements VM-RST001 and VM-RST002 in Sect. 3.

$$
\begin{aligned}
&CHECK\_RST(the\_coffee\_request\_button,\\
&\ the\_coin\_sensor, the\_system\_mode) =\\
&\quad \text{if } (\text{not}(the\_system\_mode \mathrel{!}= 1) \text{ or } the\_coin\_sensor \mathrel{!}= \text{true}) \text{ and}\\
&\qquad \text{not}(the\_coffee\_request\_button == \text{true} \text{ and } the\_system\_mode \mathrel{!}= 0)\\
&\quad \text{then } ... \text{ else } STOP
\end{aligned}
$$

Now, we present a detailed explanation of how this monitor process is created for any system. The monitor process (*MONITOR*) interacts synchronously with the system (S – see Sect. 2.3) over the channels *input* and *output*.

$$
S' = S \underset{\{|input,output|\}}{\parallel} ((MONITOR(...) \underset{\{|get,set|\}}{\parallel} MONITOR\_MEMORY) \setminus \{...\})
$$

The process *MONITOR* receives as parameters the initial value ($init\_in\_val_i$ and $init\_out\_val_k$) of the system variables (inputs — in_var_i, and outputs — out_var_k). Then, it synchronises on the *output* event to record the first output values (out_val_k). It is necessary to keep track of the current and the previous value of variables since the restrictions might refer to the old value (see Sect. 3.3). Afterwards, *MONITOR* behaves as *MONITOR_LOOP*.

$$
\begin{aligned}
&MONITOR(init\_in\_val_1, ..., init\_in\_val_n,\\
&\ init\_out\_val_1, ..., init\_out\_val_m) =\\
&\quad output.out\_var_1?out\_val_1 ... out\_var_m?out\_val_m \rightarrow\\
&\quad MONITOR\_LOOP(init\_in\_val_1, ..., init\_in\_val_n,\\
&\qquad init\_in\_val_1, ..., init\_in\_val_n, init\_out\_val_1, ...,\\
&\qquad init\_out\_val_m, out\_val_1, ..., out\_val_m)
\end{aligned}
$$

The process *MONITOR_LOOP* has the following cyclic behaviour. First, it reads the input values that can be generated (synchronising over *input*). Then, it checks the conditions related to application scopes.

The auxiliary variables $gamma_i$ and $omega_i$ are used to keep track of whether γ_i and ω_i (for a given i-th restriction) hold in the current state. It is necessary to perform basic syntactic translations to adhere to the CSP_M syntax (e.g., $\neg$ becomes $not(...)$). Therefore, Γ_i denotes γ_i in CSP_M (similarly to Ω_i). We note that $gamma_i$ is only reset to false if the corresponding *until* condition is satisfied.

$$
\begin{aligned}
&MONITOR\_LOOP(old\_in\_val_1, ..., old\_in\_val_n, in\_val_1, ..., in\_val_n,\\
&\ old\_out\_val_1, ..., out\_out\_val_m, out\_val_1, ..., out\_val_m) =\\
&\quad input.in\_var_1?in\_val'_1 ... in\_var_n?in\_val'_n \rightarrow\\
&\quad (\text{if } \Gamma_1 \text{ then } set!gamma_1!true \rightarrow SKIP \text{ else } SKIP)\ ;\ ...
\end{aligned}
$$

$$
\begin{array}{l}
(\text{if } \Omega_1 \text{ then } set!omega_1!true \rightarrow set!gamma_1!false \rightarrow SKIP \\
\qquad \text{else } set!omega_1!false \rightarrow SKIP) \ ; \ ... \\
CHECK\_RST(in\_val_1, ..., in\_val_n, in\_val'_1, ..., in\_val'_n, \\
\qquad old\_out\_val1, ..., old\_out\_val_m, out\_val_1, ..., out\_val_m)
\end{array}
$$

After setting the value of these auxiliary variables, it checks whether each scenario is valid according to the restrictions (auxiliary process *CHECK_RST*). Being valid means that all environment restrictions (Π_i as the CSP_M version of π_i) are satisfied, if within their application scopes. Since the CSP_M syntax does not support $a \Rightarrow b$, the implications are represented as "not(a) or b". If the application scope is *global*, the values ($v\_$) of $gamma_i$ and $omega_i$ are not considered. If the application scope is *after*, the value of $omega_i$ is not considered.

$$
\begin{array}{l}
CHECK\_RST(old\_in\_val_1, ..., old\_in\_val_n, in\_val_1, ..., in\_val_n, \\
\quad old\_out\_val_1, ..., out\_out\_val_m, out\_val_1, ..., out\_val_m) = \\
\qquad get!gamma_1?v\_gamma_1 \rightarrow ... \rightarrow get!gamma_l?v\_gamma_l \rightarrow ... \\
\qquad get!omega_1?v\_omega_1 \rightarrow ... \rightarrow get!omega_l?v\_omega_l \rightarrow ... \\
\qquad \text{if } (\text{not}(v\_gamma_1 \text{ and not}(v\_omega_1)) \text{ or } \Pi_1) \text{ and ... and} \\
\qquad\quad (\text{not}(v\_gamma_l \text{ and not}(v\_omega_l)) \text{ or } \Pi_l) \text{ then} \\
\qquad\quad output.out\_var_1?out\_val'_1 ... out\_var_m?out\_val'_m \rightarrow \\
\qquad\quad MONITOR\_LOOP(old\_in\_val_1, ..., old\_in\_val_n, \\
\qquad\qquad in\_val_1, ..., in\_val_n, out\_val_1, ..., out\_val_m, \\
\qquad\qquad out\_val'_1, ..., out\_val'_m) \\
\qquad \text{else } STOP
\end{array}
$$

If all restrictions are satisfied, the if-condition evaluates to true, and the monitor process allows for system responses (synchronisation over *output*) before behaving as *MONITOR_LOOP* again (passing as argument the updated value of variables). However, if this condition is not true, then the monitor process deadlocks. As a consequence, it makes the system process (S) to deadlock as well, since it can only perform input/output events if the monitor process agrees (synchronises) on them.

Deadlock is a desired effect here, since it prohibits the system to advance under undesired scenarios; some traces will not have an output after the inputs that violate the environment restrictions. These traces will not be considered when generating test cases, since we only take into account traces where for each input one can observe the expected system reaction (output).

Although this approach is compositional (it does not require modifications on S) and reduces the final model ($S \ || \ MONITOR$) state space, it does not simplify the original model of the system (S) to consider only valid inputs; consequently, it does not reduce the final model compilation time, which is a relevant aspect when using FDR. The underlying reason is the way FDR deals with the parallel composition. In Sect. 4.3 we discuss in more detail the importance of optimising the compilation of CSP models when using FDR. Nevertheless, this approach might be useful if compositionality is mandatory, when modifying the system model is not possible; for instance, when performing black-box model-based testing.

4.2 Approach 2: Changing Input Generation

Our second approach also imposes the environment restrictions, and the resulting labelled-transition system (LTS) is created in less time. This approach is even simpler to encode than the previous one, but it requires the modification of the CSP process originally created for the system behaviour (S). The idea here is to modify the process *INPUTS* (see Sect. 2.3) to block (deadlock on) the undesired scenarios. Considering the VM example, it suffices to define a process $CHECK\_RST'$ (similar to the one defined in Sect. 4.1), and to compose it sequentially with *INPUTS*: defining a new process $INPUTS' = INPUTS \; ; \; CHECK\_RST'$.

In details, let $CHECK\_RST'$ be the following CSP process. After reading the current and previous values of the system variables, along with the values of the auxiliary variables $gamma_i$ and $omega_i$, it checks whether the environment restrictions hold. If so, the process finishes successfully ($SKIP$). Otherwise, it deadlocks ($STOP$).

$$
\begin{aligned}
&CHECK\_RST' = \\
&\quad get!old\_in\_var_1?old\_in\_val_1 \rightarrow ... \\
&\qquad \rightarrow get!old\_in\_var_n?old\_in\_val_n \rightarrow \\
&\quad get!in\_var_1?in\_val_1 \rightarrow ... \rightarrow get!in\_var_n?in\_val_n \rightarrow \\
&\quad get!old\_out\_var_1?old\_out\_val_1 \rightarrow ... \\
&\qquad \rightarrow get!old\_out\_var_m?old\_out\_val_m \rightarrow \\
&\quad get!out\_var_1?out\_val_1 \rightarrow ... \rightarrow get!out\_var_m?out\_val_m \rightarrow \\
&\quad get!gamma_1?v\_gamma_1 \rightarrow ... \rightarrow get!gamma_l?v\_gamma_l \rightarrow ... \\
&\quad get!omega_1?v\_omega_1 \rightarrow ... \rightarrow get!omega_l?v\_omega_l \rightarrow ... \\
&\quad \text{if } (\text{not}(v\_gamma_1 \text{ and not}(v\_omega_1)) \text{ or } \Pi_1) \text{ and ... and} \\
&\qquad (\text{not}(v\_gamma_l \text{ and not}(v\_omega_l)) \text{ or } \Pi_l) \text{ then } SKIP \text{ else } STOP
\end{aligned}
$$

Now, we update the original *SPECIFICATION* process considering a new process for generating inputs ($INPUTS'$). After generating inputs, the scenarios that are not feasible in practice are pruned from the resulting LTS, since $CHECK\_RST'$ deadlocks. S'' denotes the process created using this second approach. This approach yields a faster compilation time, since the environment restrictions are imposed during the creation of the LTS of S''.

$$
\begin{aligned}
&INPUTS' = INPUTS \; ; \; CHECK\_RST' \\
&SPECIFICATION' = \\
&\quad ... \rightarrow FUN \; ; \; ... \rightarrow INPUTS' \; ; \; DELAY \; ; \; SPECIFICATION' \\
&SYSTEM' = SPECIFICATION' \underset{\{|get,set|\}}{\parallel} SYSTEM\_MEMORY \\
&S'' = SYSTEM' \setminus \{...\}
\end{aligned}
$$

It is important to note that our second approach is semantically equivalent to the first one in the CSP trace semantics (Theorem 1). Let $\mathbb{S}$ be the set of all CSP specifications of data-flow reactive systems, S be a given CSP specification, and $appr1$ and $appr2$ functions that yield a CSP specification considering the

environment restrictions of S (R_S) according to the first (Sect. 4.1) and the second (Sect. 4.2) approaches, previously described.

Theorem 1. $\forall S : \mathbb{S} \bullet appr1(S, R_S) \sqsubseteq_{\mathrm{T}} appr2(S, R_S) \wedge appr2(S, R_S) \sqsubseteq_{\mathrm{T}} appr1(S, R_S)$

In CSP, the traces refinement relation means trace inclusion. Therefore, if $P \sqsubseteq_{\mathrm{T}} Q \wedge Q \sqsubseteq_{\mathrm{T}} P$ holds, it means that both processes have the same set of traces (i.e., they are equivalent in this semantic model). The proof of Theorem 1 relies on the fact that both approaches create a deadlock on situations where the restrictions are not satisfied. The difference between them is that the first one creates the deadlock via parallel synchronisation, whereas the second one uses the primitive process $STOP$. □

Another important theoretical result of our work is described by Theorem 2.

Theorem 2. $\forall S : \mathbb{S} \bullet S \sqsubseteq_{\mathrm{T}} appr1(S, R_S)$

The CSP specification yielded by $appr1$ (or $appr2$ — see Theorem 1) might deadlock on some (or none) of the traces of S, where undesired input scenarios occur, but it does not produce new traces (new events are not performed). Therefore, the traces of $appr1(S, R_S)$ are a subset of the traces of S. □

Differently, $appr1(S, R_S) \sqsubseteq_{\mathrm{T}} S$ does not hold in general, since we expect the left-hand side process to have less traces than S due to the imposed restrictions.

4.3 Relevance of Compilation Optimisation

Consider the following CSP specification.

$$
\begin{aligned}
&\text{channel } input, output : \{0..20000\}\\
&A(v) = \text{ if } v >= 0 \text{ then } output.v \rightarrow A(v-1) \text{ else } STOP\\
&P = input?v \rightarrow A(v)\\
&Q = input?v \rightarrow \text{if } v == 2 \text{ then } A(v) \text{ else } STOP\\
&R = P \underset{\{|input, output|\}}{\parallel} Q\\
&P' = input?v \rightarrow \text{ if } v == 2 \text{ then } A(v) \text{ else } STOP\\
&R' = P'
\end{aligned}
$$

A is an auxiliary process that performs the event $output.i$, with i varying from v to 0. P receives an input value v and then behaves as $A(v)$. Q behaves similarly to the process $MONITOR$: it synchronises on all communications over the input and the output channels, and restricts P to behave as $A(2)$ (suppose that 2 is the only feasible input).

The process R' is equivalent to R (both processes have the same set of traces), but it is defined differently. It follows our second approach (detailed in Sect. 4.2) to restrict the behaviour of P, which involves modifying the definition of P (i.e., defining a new process P'). Considering the channels $input$ and $output$ ranging

from 0 to 20,000, more than 60s is necessary to create the LTS of R, whereas the LTS of R' is created within 20s[4].

When constructing (compilation phase) the LTS of R, FDR first expands the LTS of P and Q, and then constructs the resulting LTS via bisimulation. Therefore, although the resulting LTS has less states (reduction of the state space), the time required to construct this LTS tends to be the same or even greater. In other words, in general, the approach described in Sect. 4.1 does not represent performance gains with respect to compilation time, but only regarding analysis time (when the resulting LTS model has already been created).

5 Empirical Analyses

Our evaluation considers examples from four different domains: (i) the vending machine (VM) discussed in Sect. 3; (ii) the control system for safety injection in a nuclear power plant (NPP) presented in [10]; (iii) a priority command function (PC) provided by Embraer[5]; and (iv) part of the turn indicator system (TIS) of Mercedes vehicles[6].

In order to provide an argument to the efficacy of our proposal, we measured the achieved reduction in terms of number of states and transitions. We only consider the approach described in Sect. 4.2, since the other one (Sect. 4.1) does not improve the model compilation time, and, thus, it is does not scale for complex examples such as the TIS (exceeds available RAM memory).

Threats to external validity (the ability to generalise our conclusions) apply to our analyses, since we do not consider a large set of examples. Despite that, the results give some evidence about the efficacy of our proposal. Table 4 summarises our findings; S is the original system model, whereas S'' is the system model constrained by the test environment restrictions (as described in Sect. 4.2).

Table 4. Metrics of the empirical analyses

	VM	NPP	PC	TIS
#restrictions of S''	2	3	1	2
#states of S	4,652	14,681	5,592	215,470
#states of S''	1,814	12,261	2,728	189,644
Reduction (states)	**61.01%**	**16.48%**	**51.22%**	**11.99%**
#transitions of S	4,761	15,617	6,137	228,141
#transitions of S''	1,841	12,975	2,949	200,339
Reduction (trans.)	**61.33%**	**16.92%**	**51.95%**	**12.19%**

A significant reduction in the number of states/transitions was achieved for the VM (61.01%/61.33%) and for the PC (51.22%/51.95%) examples, whereas

[4] Considering an i7-5500U @ 2.40 GHz × 4, 8 GB of RAM, with Ubuntu 16.04 LTS.
[5] http://www.embraer.com/en-us/pages/home.aspx.
[6] http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/index_e.html.

it was smaller for the NPP (16.48%/16.92%) and the TIS (11.99%/12.19%). The reduction was smaller for the NPP and TIS examples, since the infeasible scenarios specified by the environment restrictions are less common than the ones considered in the other two examples (VM and PC). Nevertheless, as said before, besides the benefit of reducing the state space, there is a more general benefit of developing more meaningful models, since infeasible scenarios can be ignored by analysis via model checking, simulation and the final implementation.

6 Conclusion

This paper presents a strategy for modelling environment restrictions formally in order to develop more meaningful models of the system behaviour, besides taking advantage of them to reduce the input space of models. The proposed approach integrates different techniques and notations (natural-language processing, linear temporal logic, CSP, and model checking). The restrictions are formalised as LTL formulae, which are automatically generated with the aid of a controlled natural language. Then, these formulae are used to impose the restrictions to a CSP model of the system.

The contribution of this work integrates with the NAT2TEST strategy, which provides means for generating test cases from natural-language requirements, ruling out infeasible test scenarios. The efficacy of our proposal is illustrated considering examples from the literature, and from the aerospace (Embraer) and the automotive (Mercedes) industry. Despite the integration with NAT2TEST, our results can also be applied to other contexts, taking as starting point the LTL formulae automatically derived. For instance, it is possible to take into account these LTL formulae to perform classical model checking [11].

Generating temporal logic formulae from natural-language specifications is not a new research topic. In [6] an action-based branching temporal logic (ACTL) is used to formalise requirements in order to support verification of specification properties. More recently, reference [17] presents another strategy for formal consistency checking of natural-language requirements via the generation of LTL formulae. In [12], similarly to our work, the authors use case-grammar theory to support the generation of LTL formulae. A common aspect between these works and ours is the definition of an underlying structure (via templates or CNL) for writing requirements. However, differently from them, our LTL formulae are defined over variables and values, and not over events.

Formal modelling of the environment has already been addressed too. For instance, in [9] a conformance relation (i $rtioco_e$ s) is proposed to relate implementation (i) and specification (s) models in the light of an environment model (e); all models are defined as timed input-output transition systems. In the RT-Tester tool [14], the system behaviour and the test environment are both modelled as state machines. In [13], considering programmable controllers, the authors propose a strategy for reducing the set of test cases by modelling the plant behaviour, additionally to the system behaviour, as finite state machines. Differently from our work, the user needs to manually and formally model the

environment. Here, the formal model of the environment restrictions is automatically generated from high-level descriptions in natural language. However, these other works can model arbitrary properties, which is not our case.

As future work, we intend to: (1) extend our CNL to allow the specification of other types of restrictions, (2) investigate the use of valency grammar [1] in contrast to the case-grammar theory, and (3) conduct further empirical analyses.

Acknowledgements. This work is partially supported by INES (www.ines.org.br), CNPq grant 465614 /2014-0 and FACEPE grants APQ-0399-1.03/17 and PRONEX APQ/0388-1.03 /14. It is also partially supported by the CIn-UFPE and Motorola cooperation project, as well as by the CNPq grants 303022/2012-4 and 132332/2015-9.

References

1. Allerton, D.J.: Valency grammar. In: Brown, K. (ed.) The Encyclopedia of Language and Linguistics, pp. 301–314. Elsevier Science Ltd. (2006)
2. Carvalho, G.: NAT2TEST: generating test cases from natural language requirements based on CSP. Ph.D. thesis, Centro de Informática, UFPE, Brazil (2016)
3. Carvalho, G., Barros, F., Carvalho, A., Cavalcanti, A., Mota, A., Sampaio, A.: NAT2TEST tool: from natural language requirements to test cases based on CSP. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 283–290. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_20
4. Carvalho, G., Cavalcanti, A., Sampaio, A.: Modelling timed reactive systems from natural-language requirements. Form. Asp. Comput. **28**(5), 725–765 (2016)
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 411–420. ACM, New York (1999)
6. Fantechi, A., Gnesi, S., Ristori, G., Carenini, M., Vanocchi, M., Moreschini, P.: Assisting requirement formalization by means of natural language translation. Form. Methods Syst. Des. **4**(3), 243–263 (1994)
7. Fillmore, C.J.: The case for case. In: Bach, E., Harms, R.T. (eds.) Universals in Linguistic Theory, pp. 1–88. Holt, Rinehart, and Winston, New York (1968)
8. Gibson-Robinson, T., et al.: FDR: from theory to industrial application. In: Gibson-Robinson, T., Hopcroft, P., Lazić, R. (eds.) Concurrency, Security, and Puzzles. LNCS, vol. 10160, pp. 65–87. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51046-0_4
9. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using Uppaal. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 79–94. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31848-4_6
10. Leonard, E., Heitmeyer, C.: Program synthesis from formal requirements specifications using APTS. High. Order Symbol. Comput. **16**, 63–92 (2003)
11. Leuschel, M., Currie, A., Massart, T.: How to make FDR spin LTL model checking of CSP by refinement. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 99–118. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45251-6_6
12. Lignos, C., Raman, V., Finucane, C., Marcus, M., Kress-Gazit, H.: Provably correct reactive control from natural language. Auton. Robot. **38**(1), 89–105 (2015)

13. Ma, C., Provost, J.: A model-based testing framework with reduced set of test cases for programmable controllers. In: Proceedings of the IEEE Conference on Automation Science and Engineering, pp. 944–949. IEEE (2017)
14. Peleska, J., Vorobev, E., Lapschies, F., Zahlten, C.: Automated model-based testing with RT-tester. Universität Bremen, Technical report (2011)
15. Pnueli, A.: The temporal semantics of concurrent programs. Theor. Comput. Sci. **13**(1), 45–60 (1981)
16. Roscoe, A.W.: Understanding Concurrent Systems. Springer, London (2010). https://doi.org/10.1007/978-1-84882-258-0
17. Yan, R., Cheng, C.H., Chai, Y.: Formal consistency checking over specifications in natural languages. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, pp. 1677–1682. EDA Consortium (2015)