Information Engineering Department
Institute of Communication, Information and Perception Technologies
University of Pisa, Scuola Superiore Sant'Anna

From natural language requirements to Simulation Monitors: Synthesis through code generation

Master Degree in Embedded Computing Systems

Supervisors:
Prof. Marco Di Natale
Prof. Giorgio C. Buttazzo

External Supervisor:
Dr. Stylianos Basagiannis

Author:
Marco Emanuele Celia

University of Pisa July 2017

Acknowledgements

Marco Celia
Cork, July 2017

Abstract

The increasing level of complexity of modern embedded systems concurs to enhance the gap between the textual representation of High-Level System Requirements and the testing units in charge of their verification. Such gap is worsened by the possible presence of errors and omissions in the natural language requirement and causes engineers to inject into the verification units features derived from their personal interpretation.

The goal of this Master thesis is to provide a framework consisting of a requirement (loose) syntax, a requirements editor, a parser, and an automatic synthesis tool, which helps engineers writing high level requirements in a structured natural language with a contract-based paradigm, thus reducing, as much as possible, inconsistency and common errors, and directly generating verification monitors for a target platform.

To restrict the domain and the framework complexity and make the problem affordable, the first version of the tool is restricted to the controls domain and assumes the use of Simulink as a modelling and simulation tool. To ease the generation process, a control verification Simulink library has been implemented.

Table of contents

List of figures **vii**

1 Introduction **1**

1.1 Development Models . . . 1
1.2 Requirements Fundamentals . . . 2
1.2.1 Definition . . . 2
1.2.2 Guidelines . . . 3
1.2.3 Requirement Specification languages . . . 4
1.2.4 Readability versus Formality . . . 5
1.2.5 Requirements Specialization . . . 5
1.2.6 High-Level Requirements . . . 6
1.3 Verification tools . . . 7
1.3.1 Property checking . . . 7
1.3.2 Automatic monitor generation . . . 8
1.4 Proposed Tool . . . 8
1.4.1 Premise . . . 8
1.4.2 Custom Editor . . . 9

2 MDB and Hybrid Systems **11**

2.1 Why models . . . 11
2.2 Model Based Design . . . 12
2.2.1 Basis elements . . . 12
2.2.2 Automatic Code Generation . . . 13
2.3 Hybrid Systems . . . 14
2.3.1 State Machines . . . 14
2.3.2 Time-Based Model . . . 15
2.3.3 LTI System . . . 16
2.3.4 Mixed Models . . . 17

2.3.5 Formal Model 17
2.3.6 Examples 20

3 Formal Verification 22

3.1 Temporal Logic 22
3.1.1 Linear Temporal Logic 22
3.1.2 Computation Tree Logic 24
3.1.3 Property Specification Language 27
3.1.4 Signal Temporal Logic 28
3.1.5 Summary 30
3.2 Patterns Identification 30
3.2.1 Property Specification Patterns 30
3.2.2 Domain Patterns 33
3.3 The Contract Paradigm 36

4 The Framework 38

4.1 From Requirements to Monitors 38
4.1.1 Syntax Definition 39
4.2 Support from Modeling Environment 41
4.2.1 Signal Exporter 41
4.2.2 STL operators library 43
4.2.3 System Performance Control Library 46
4.3 Design Patterns 52
4.3.1 Factory 52
4.3.2 Visitor 55
4.4 Eclipse Editor 55
4.4.1 Data Dictionary Importer 55
4.4.2 Context Aware Syntax Helper 55
4.4.3 Abstract Syntax Tree Generation 55
4.5 Model population 55

List of figures

1.1 Waterfall Model 1
1.2 V-Model 2
1.3 Requirement specialization 6
1.4 Off-line, Passive On-line and Active On-line monitors from [?] 7
1.5 Framework 9

2.1 Modeling 11
2.2 Elements of MBD 13
2.3 Hybrid System Structure 18
2.4 Hybrid System: Tick Generator 20
2.5 Tick Generator Internals 20
2.6 Hybrid System: Bouncing Ball 21

3.1 CTL formulas 26
3.2 Property Scopes 31
3.3 Pattern Hierarchy 32
3.4 Rise Time 34
3.5 Overshoot 35
3.6 Settling Time 36

4.1 Requirement Framework 38
4.2 STL Simulink 43
4.3 Test of STL Always-Eventually 44
4.4 Until Simulation 45
4.5 Timed Until Implementation 45
4.6 System Performance Control Library 46
4.7 Input References 47
4.8 Detector’s masks 48
4.9 Detectors’ time behavior 48

4.10 Ramp Detector Implementation 49
4.11 Overshoot Blocks' Masks 50
4.12 Overshoots' Time response 50
4.13 Overshoots' Implementations 51

Chapter 1

Introduction

1.1 Development Models

Every industrial product has its own *life-cycle* which starts when comes out the need of a new product and continues with the requirements identification, design, development and verification processes. Furthermore several other activities regarding the product maintenance could be as well part of the cycle. There are several approaches describing the sequentiality of product life cycle tasks, such as the so called *Waterfall model* (Fig.1.1), in which a phase starts after the completion of the previous one and phases' deliverables follows an unidirectional flows like in a pipeline.

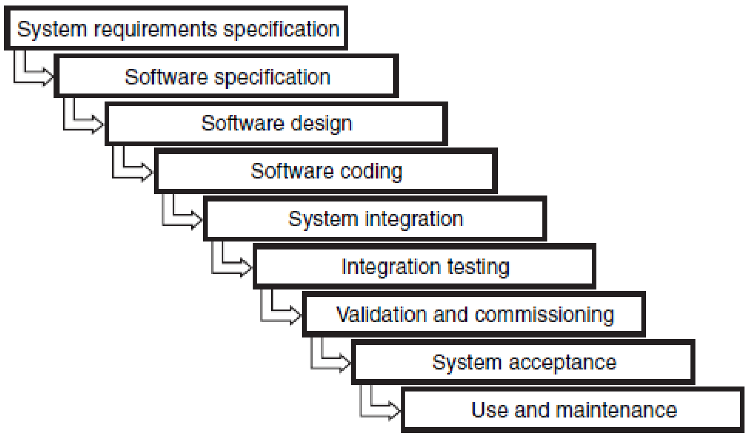


Fig. 1.1 Waterfall Model

A example of the waterfall model we can examine the military standard MIL-STD-2167A, which establish a uniform development model applicable during the whole system life-cycle.

As second alternative the *V-model* (Fig.1.2) enforces the temporal correlation among developing and validation phases. Activities on the left hand side of the V are associated with models of the system, where each subsequent activity represents an enhancement of the model from the previous activity. Processes on the right hand side of the V correspond to experimental (testing) activities. Results of a test phase validate the structure of the correspondent developing phase.

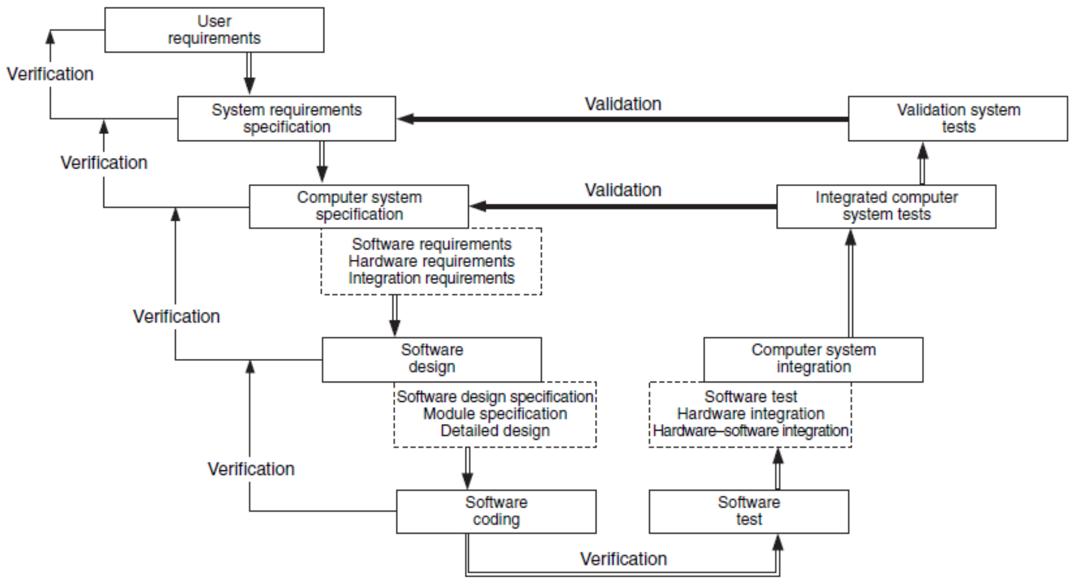


Fig. 1.2 V-Model

This work focuses on the first stage of a generic product development model, i.e. Requirement analysis, trying to reduce the gap also with its relative validation.

1.2 Requirements Fundamentals

1.2.1 Definition

"...Requirements definition is a careful assessment of the needs that a system is to fulfill. It must say why a system is needed, based on current and foreseen conditions, which may be internal operations or an external market. It must say what system features will serve and satisfy this context. And it must say h ow the system is to be constructed..." — Doug Ross [Ross77a]

Regardless of the adopted development model, the first, and critical, step to approach a product development is the System Requirement specification. It is much more than a functional specification, it is the groundwork for all the future stages and hence it has to catch everything is needed in terms of performance, behaviors and constraints to be satisfied.

Being the requirement analysis the fundamental step that drives the design and implementation phases it is also worth noting how errors at this stage, *late* errors, are much more costly in terms of time to be fixed than *early* errors. The DoD Software Technology Plan [**?**] states that "early defect fixes are typically two orders of magnitude cheaper than late defect fixes, and the early requirements and design defects typically leave more serious operational consequences."

1.2.2 Guidelines

The result of the system requirements specification process is an unambiguous and complete specification document. It should help:

a) System customers to accurately describe what they wish to obtain;

b) System suppliers to understand exactly what the customer wants;

Furthermore to the stakeholders a good system requirements specification (SRS) should provide several specific benefits, such as the following:

- *Establish the basis for agreement between the customers and the suppliers on what the software product is to do*. The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs

- *Reduce the development effort.* The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

- *Provide a baseline for validation and verification.* Organizations can develop their validation and verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.

Attempts to reduce as much as possible the inconsistencies coming from the plain text descriptions has been proposed in several standards. The IEEE 830-1998 [**?**] offers a metric to produce and evaluate a good system requirements specification in terms of

a) Correctness

b) Unambiguity

c) Completeness

d) Consistency

e) Verifiability

f) Modifiability

g) Traceability

while the MIL-STD-498 [**?**] was a United States military standard whose purpose was to establish uniform requirements for systems development and documentation. Those standards rather than automatic tools are sort of guidelines to achieve a good specification for the system.

1.2.3 Requirement Specification languages

As said above, the requirement document quite often serves as a "contract" between designers and the costumers, as such it has to be somehow understandable also from those people having a non technical background, so quite often they are given with many pages of natural language descriptions, sometimes flanked with manually derived diagrams that represent the structure of the design. Due to its nature, the natural language specification of a property could be very far, and sometimes totally unrelated, from a formal definition. For instance natural language could be lacking, redundant or misleading, even worse the requirement definition is generally the result of an empirical process subjected to ambiguity, subjectivity and imprecisions.

The literature provides several methods aiming to avoid bad-formed requirements, all of them are based on the addition of a certain degree of formality in the syntax used to define the property referring a specific requirement. The most general classification of the specification languages is into *formal*, *semi-formal* and *informal*.

A language is formal if its syntax and semantic are defined in a rigorous mathematical way. A formally expressed requirement provides an high level of verifiability, furthermore the process of translating natural into formal language requires the analysts to improve their comprehension of the requirement semantic. In fact, in order for the translation to be successful they have to verify, and eventually eliminate, the presence of ambiguities also from the natural language definition.

Semi-formal languages are usually expressed in a graphical manner, aimed to provide a non-ambiguous description through a generally easier syntax that has not a direct mapping to a mathematical semantic. The *structured natural language*, which will be of interest in below discussions, could be in turn part of this category. In particular it could be view as a plain text which has some constraint in the precedence of sentences participants.

Informal languages are those that have neither a rigorous syntax or rigorous semantic. In particular the syntax is derived from the relative grammatic and the semantic is that rich that makes impossible its formalization. Despite all this, they are the most commonly used, and, hence, a good specification always relies on the experience of designers.

1.2.4 Readability versus Formality

The more we increase the syntax formality and the more the number of possible inconsistencies is reduced. As drawback requirement specification starts taking a non-easy readable shape and becomes very distant from its natural language expression. There are several reasons why this aspect represents an obstacle to the usability of these techniques. First, a requirements document could not be anymore used as sort of contract among stakeholders since there are no guarantees that one of the parts is able to understand the meaning of the formula. Further, their use implies engineer to have a certain level of expertise in typing formal requirements. Although the usability does not constitute the strength of this approach the reason why formal languages are still of interest is due to their capability of totally avoid formulas ambiguity.

1.2.5 Requirements Specialization

A direct consequence of the use of formal languages is the possibility to perform automatic verification techniques out of the requirements, but, as previously said, rarely the specifications are formally written, implying, hence, several manual translation stages from natural language.

This work bridges the gap between formal and informal representation of requirements. In order to do so the effort has been spread along an ideal line starting from the natural up to the formal language description. The first step consists of defining a syntax which allows the presence of free-text, limited to certain points of the sentence, and, at the same time, preserves a well defined structure. Once the requirements are structured is then possible to apply some processing technique able to recognize every *common pattern*, i.e. all those properties having a direct correlation, in terms of temporality or property semantic, with a specific formal language pattern. Finally the formal properties are used to generate automatically verification monitors.

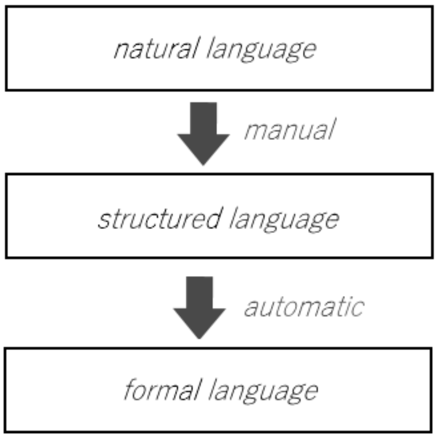


Fig. 1.3 Requirement specialization

1.2.6 High-Level Requirements

Rarely system requirements are written in a formal language, the non-rigorous nature of plain text constitutes the main obstacle to automatic generation of monitors out of the specification. Writing requirements in a semi-formal shape rather than a plain text will make them suitable to language processing techniques which allow to get their formal specification. As enforcement of this concept, in certain specific application domain requirements usually refer to properties having a well-known semantic. Typically for such cases also plain text requirements tend to have a fixed structure. This scenario commonly recurs when the specification is given from an high level of abstraction of the systems and, hence, does not regard constraints on the development. Performance requirements in a control system are an example. In fact they have an invariant textual form compared, for instance, to system's hardware structure or control algorithms.

1.3 Verification tools

1.3.1 Property checking

In general, the problem of determining whether a property ϕ is satisfied from system S could be faced in many ways depending on the set of environmental conditions in which system and property are defined. Being not too much formal we can define a system S as an entity that correlates inputs and output signals according to a precise law, while the *model* of the system as a set of possible *states* in which system can lie, each state x_i ranging over the domain X_i. A system evolves over the time domain and the function $\beta : T \rightarrow X$, associating a state to a precise time instant, is called *behavior* of the systems. Within this context a property ϕ defines a subset of the systems' state and a property *monitor* is that entity that ensures whether a certain behavior *satisfies* the property, i.e. whether $L_\phi \in X$.

The procedure leading to the verification of a property, or formula, consists of associating a function $\Omega_\phi : X \rightarrow B$ which, given the set of all possible behaviors returns a boolean value meaning whether the property is satisfied or not. Behaviors are examined one at time assuming the presence of a simulation unit charged to produce them. A monitor is hence another unit which cooperates with the system simulator mainly in three ways (Fig.1.4).

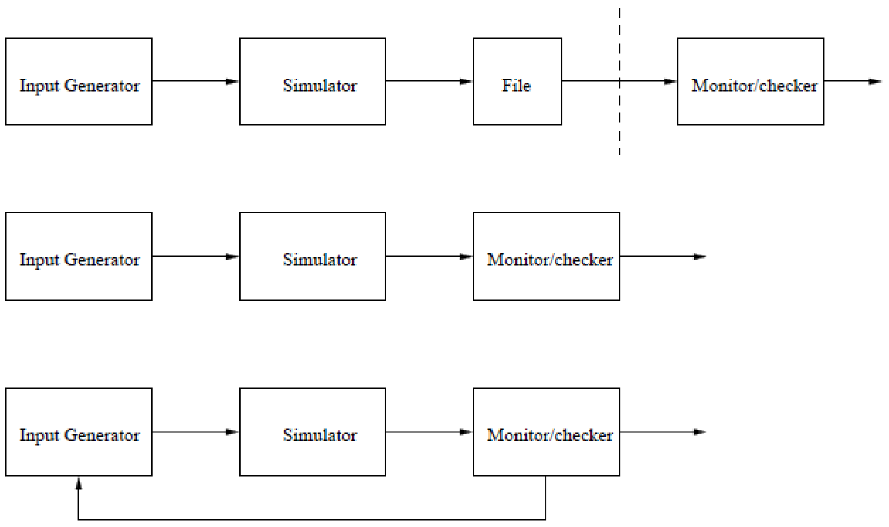


Fig. 1.4 Off-line, Passive On-line and Active On-line monitors from [**?**]

The *off-line* monitoring expects monitors to check formulas against the simulation traces. Since monitors are no forced to be causal over the execution time this technique constitutes the easiest way to conduce model checking. In the *on-line* methods the monitor is running in parallel with the simulator, we refer to *active* on-line if the monitor is also affecting, through a feed-back loop, the simulation inputs, and *passive* on-line otherwise. In general on-line monitors have the advantage of being able to assert satisfaction or violation of a property while the simulation is running, this prevents either high time consuming simulation to be terminated and dangerous behavior to occur after the violation of the property. The cost we pay for an on-line monitoring is that formulas must be causal, i.e. it is not possible to check a property at certain time t as function of states occurring later than t. Since this work focuses mostly on dynamic controls system domain monitors that are better suited to fit it are of the type passive on-line.

1.3.2 Automatic monitor generation

Under the assumption that requirements are written in formal language, i.e. absence of inconsistencies, the generation of verification monitors for simulation can be performed in an automatic way.

Code generation procedures are always targeted to a specific platform, which in turn varies as function of the application domain. For what regards dynamical systems typical targets are simulation frameworks such as Simulink by Matworks[**?**] and LabVIEW[**?**]. Together with the target platform usually code generation starts from a model of the entities we want generate for.

Tools such as Acceleo[**?**] are based on the prior definition of the mode of the system architecture, allowing language independent code generation from any kind of EMF compatible meta-model, such as UML, UML 2.0 and SysML. Also Simulink with the Embedded Coder offers the possibility to customize the C/C++ code generated from any Simulink model.

Another chance is to not rely on the presence of a system model and generate code implementing verification monitors directly from statements expressed in some formal language. The work in [**?**] starts from properties expressed in the STL formal language[**?**] and produces Simulink monitors, to ease the auto-generation procedure it makes use of a library implementing the STL temporal operators.

1.4 Proposed Tool

1.4.1 Premise

This work aims to show the feasibility of a tool that starts from semi-formal system specification and ends up with verification monitors to be plugged into a simulation model of the system. The number of particular cases to deal with it is huge and the scalability of the approach is a key factor that needs time to be proved. However, due to the impact that such a tool could have in the design and verification processes, the problem has been sized in order to be manageable at least with the purpose of producing a proof of concept prototype.

For the above reasons we decide to use, as a case study, the high-level requirements for dynamic control systems. Such requirements has been restructured in the *contract-base* paradigm, which, basically, expect the formulation of the property as a composition of two main sections, *assumption* and *assertion*, were the first logically implies the second.

Further the formalization of the requirements has been performed using the STL formal language, which, unlike many others temporal logics derived from LTL, adds some domain specific features that will be better described in sec.3.1.4. Finally, as target platform, Simulink has been selected since it is one of the most used simulation tools for such application domain. A graphical representation of the whole framework together with the smart editor, which will be discussed in next section, is presented in figure 1.5.

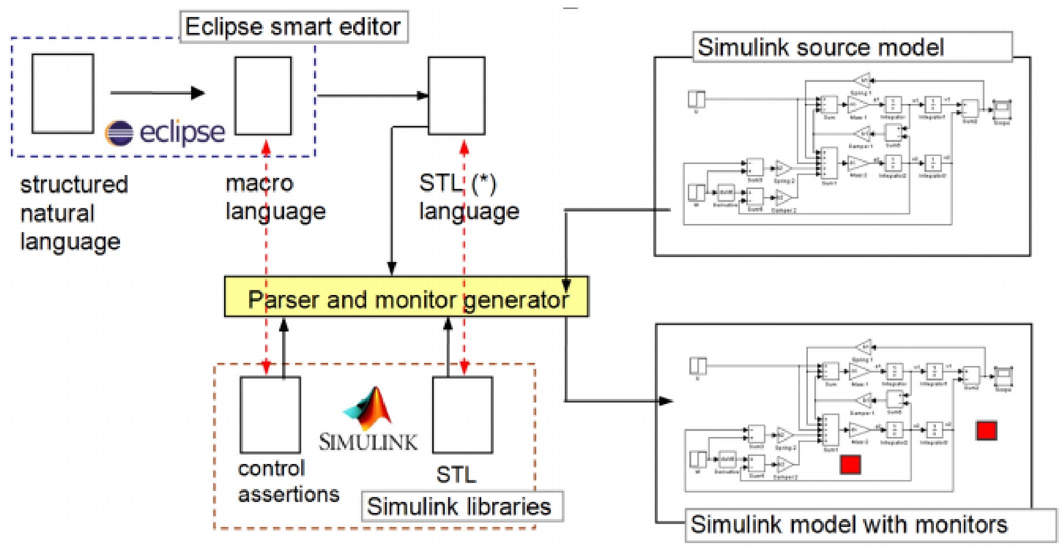


Fig. 1.5 Framework

1.4.2 Custom Editor

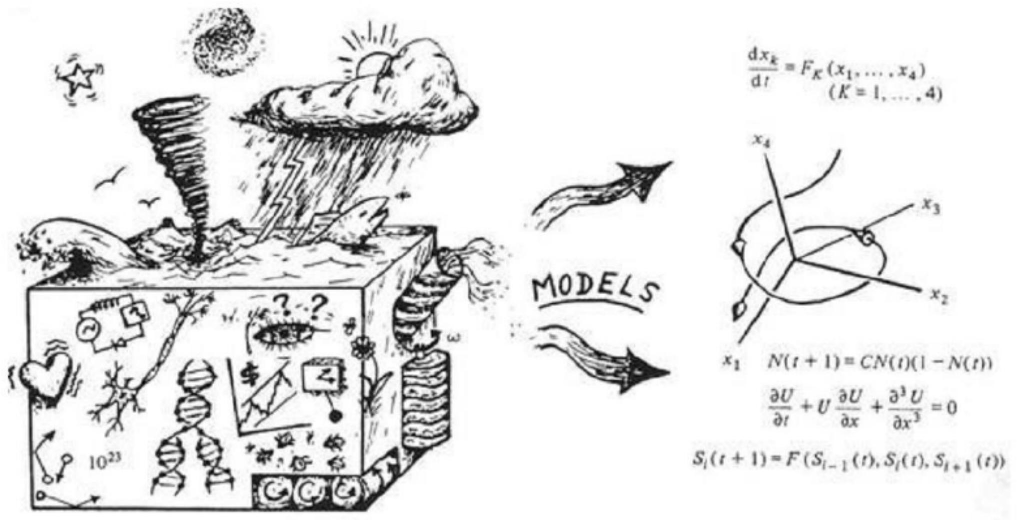
The smart editor is meant as the tool helping users to write semi-formal requirements. It has been implemented as an Eclipse Editor plug-in, from which inherits many features common to most commercial editors such as *context-aware* text completion and syntax error checking. The editor comes with it is own syntax which even if for the time being is deeply application domain oriented preserves a modular an easily extensible software structure.

Some more specific features are the capability to import a data dictionary which is ideally synchronized with the target model and generate platform dependent monitors.

Chapter 2

MDB and Hybrid Systems

2.1 Why models



MODELS
$\frac{dx_k}{dt} = F_K(x_1, \dots, x_4)$
$(K = 1, \dots, 4)$
x_4
x_3
x_2
x_1
$N(t+1) = CN(t)(1 - N(t))$
$\frac{\partial U}{\partial t} + U\frac{\partial U}{\partial x} + \frac{\partial^3 U}{\partial x^3} = 0$
$S_i(t+1) = F(S_{i-1}(t), S_i(t), S_{i+1}(t))$
10^{23}

Fig. 2.1 Modeling

We can define a model as a simplification of the reality (Fig.2.1). Models can be of any nature, *mathematical models* are, typical, sets of equations, *physical models* could be mock-ups aiming to provide a final picture of what the system will look like and so on.

Usually a model is the result of a process affected of different metrics. In a very generic perspective the first is the determination of *model type*, namely which of all the possible approaches, leading to a correct result, to model our system (mathematical, graphical, textual and so on) is the most appropriate one. In practical cases this step translates in choosing the most appropriate modeling tool.

The second is the determination of the *level of abstraction* of our model, since any system can be view as a composition of several layers one may be more interested in representing features of some of them. However, a good model has always to be connected enough to reality, indeed the risk to oversimplify may cause the model to lose meaningful information.

The previous metrics could be in turn influenced by the *model purpose*. One possible scenario is when industries and organizations wants to shorten as much as possible the development time, the most natural way to accomplish that is to reuse as much as possible what they already have implemented. Reuse always implies knowledge, therefore models can be used to provide graphical and more intuitive documentations for the software or system. This is the case of UML [**?**] and SysML[**?**], the first is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system, while the latter was defined as extension of the first and provide support for the specification, analysis, design, verification, and validation of systems that include hardware, software, data, personnel, procedures, and facilities.

On the other hand there could be many cases in which for a certain application domain, aerospace for instance, fully test the final product before using it's an high costly process. More, sometimes reproducing all the possible testing cases could be either too difficult or too dangerous, just think to nuclear applications. Hence, for all of these reasons, systems models could be used to simulate behavior, perform tests and verify compliance against the design. The methodology considering the model as the primary artifact for simulating the system behavior and verify properties on the behavior is called *Model Based Design*.

2.2 Model Based Design

2.2.1 Basis elements

In a traditional development, usually a system engineer defines the overall system specification and presents it as a design document to software engineers, who will have the task of implementing those ideas into a fully working solution. However, the main problem with this approach is the fact that in most cases the ideas presented by the system engineer via the specification document may widely differ to the implemented software. Even the most detailed and diligently prepared type of documentation may not always guarantee that the design document generated by the system engineers would be fully understood and interpreted correctly by the implementing software programmers. The Model Based Design (MBD) approach enables behavioral modeling based on a mathematical formalism and executable semantics, and it's the reference approach for the analysis of the system, its verication by simulation, the documentation of the design and the automatic generation of a code implementation (Fig. 2.2).

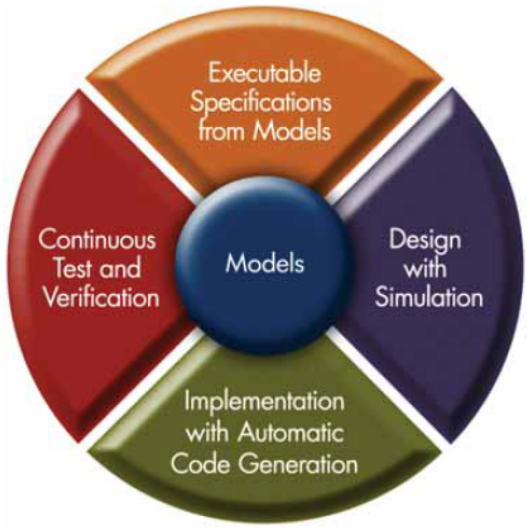

Executable Specifications from Models
Continuous Test and Verification
Models
Design with Simulation
Implementation with Automatic Code Generation

Fig. 2.2 Elements of MBD

2.2.2 Automatic Code Generation

In classical development models the implementation phase follows the design, at this stage the developers will generate suitable software code in a selected language to implement the system. If the system definitions are not clear enough, the programmers would refer to system designer for clarification, but, despite these clarifications, there's still the possibility that the developed system may be different from the one that system designers had in mind. Even worse, it may contain coding errors that invalidates all the tests performed during the design. Although unlikely not even the case in which programmers job is perfect and there are no coding errors could make hand-coding the preferable approach, indeed just consider that there could be always the need to perform small changes inside the source code, such changes, whether not propagated in the models, rapidly will make the implementation divergent from the design.

MBD offers the capability to greatly improve the above approach. Within such methodology models can be used as the input to an automatic code generation tool. Usually MBD tools offers a framework at least for modeling and generate code from models, those framework commonly comprise a language offering the capability to explore model features and use them for producing target specific code.

2.3 Hybrid Systems

2.3.1 State Machines

Systems are functions that transform signals. A broad class of systems can be characterized using the concept of state and the idea that a system evolves through a sequence of changes in state, or state transitions. Such characterizations are called state-space models. A state-space model describes a system procedurally, giving a sequence of step-by-step operations for the evolution of a system. It shows how the input signal drives changes in state, and how the output signal is produced.

A description of a system as a function involves three entities: the set of input signals, the set of output signals, and the function itself,

$$F : InputSignals \rightarrow OutputSignals$$

For a state machine, the input and output signals have the form

$$EventFlow : \mathbb{N}_0 \rightarrow Symbol$$

where $\mathbb{N}_0 = \{0, 1, 2, ...\}$ and *Symbol* is an arbitrary set. The domain of these signals represents an ordering relation that does not necessarily corresponds to time, discrete or continuous. Ordering of the domain means that it is possible to say whether one event occurs before or after another one, but without the knowledge of how much time elapses between these events.

We can define *state machine* that entity that constructs the output signal one symbol at a time by observing the input signal one symbol at a time. Specifically

$$StateMachine = (States, Inputs, Outputs, update, initialState)$$

where $States$, $Inputs$, $Outputs$ are sets, $update$ is a function, and $initialState \in States$. The meaning of these names is:

States is the set space

Inputs is the input alphabet

Outputs is the outputs alphabet

initialState is the initial state $\in States$

update : $States \times Inputs \to States \times Outputs$ is the update function

The update function makes possible for the state machine to construct, step by step, the output signal by observing the input signal. In particular, if $x(n) \in States$ is the current state at step n, and $u(n) \in Inputs$ is the current input symbol, then current output symbol and the next state are given by the following

$$(x(n+1), y(n)) = update(x(n), u(n))$$

The above could be decomposed in two different functions, one for new state and one for output

$$x(n+1) = nextstate(x(n), u(n)) \tag{2.1}$$

$$y(n) = output(x(n), u(n)) \tag{2.2}$$

where

$$nextstate : States \times Inputs \to States$$
$$output : States \times Inputs \to Outputs$$

2.3.2 Time-Based Model

The previous section defines input and output signals of a state machine as a collection of *events*. These events are generated in a domain which, usually, does not have a relation with a time set but, rather, represents just an ordered set. If we require events to happen at time instants that are multiple of a certain time quantity δ we have that

$$t\_e_i - t\_e_j = \alpha \cdot \delta, \quad i > j, \alpha \in \mathbb{Z}_+ \quad \forall i, j \in \mathbb{N}_0$$

where t_e_i and t_e_j are the time instant in which occur events e_i and e_j. Under this assumption the state machine becomes a *time-based model*, i.e. it reacts at all times in a base T.

Another specialization of could be achieved by imposing, as further assumption, that state, input and output spaces be numeric sets

$$States = \mathbb{R}^N, \quad Inputs = \mathbb{R}^M, \quad Outputs = \mathbb{R}^K$$

Combining the previous two assumptions we can define a *discrete-time system*, and the index n in (2.1) and (2.2) is called *time index*. Finally, we require that

$$u(n) \in \mathbb{R}^M \text{ and } y(n) \in \mathbb{R}^K \quad \forall n \in T$$

Therefore we disallow the capability, proper of state machines, to handle special "*do nothing*" input called *absent*.

2.3.3 LTI System

In discrete-time systems the time base $T \equiv \mathbb{Z}_+$, a definition of continuous-time system could be achieved if we relax T to $\mathbb{R}_{0^+}$. This last category of systems shares many mathematical properties with its discrete-time version, but they can no longer be view as state machines since inputs, outputs, and state transitions do not occur at discrete instances.

Formally, a representation of a continuous time system is given, $\forall t \in \mathbb{R}_{0^+}$, by

$$\begin{aligned} \dot{x}(t) &= nextstate(x(t), u(t)) \\ y(t) &= output(x(t), u(t)) \end{aligned}$$

The system is defined as *linear* if functions $nextstate$ and $output$ are linear. It is also said *time invariant* if both do not change over time. As result a system holding these two properties is said Linear Time Invariant (LTI). An interesting property of those system is that they can be represented in a compact matrix-based form. The $nextstate$ function is a $N \times (N+M)$ matrix, while the $output$ is a $K \times (N+M)$ matrix. If we partition both in two sub-matrix, the first of each comprising the first N columns we get

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

where $A \in \mathbb{R}^{N\times N}, B \in \mathbb{R}^{N\times M}, C \in \mathbb{R}^{K\times N}$ and $D \in \mathbb{R}^{K\times M}$.

The major difference between discrete and continuous system is that the new state is represented with the derivative of the current state instead of a function of current input and state. The motivation is that the derivative, which represents the trend of a function, better depicts the continuous evolution of the system rather than express it as a value at the successive time index.

2.3.4 Mixed Models

Sections 2.3.1 and 2.3.2 provided definitions of state machines and time-based model, trying to qualify the latter by imposing some constraint on the, more general, model of the first. In 2.3.3 such link was definitively broken due to the relaxation of the time to a continuous set.

In general, the two different models are used as alternative views of the same system. Even for simpler cases of discrete-time systems, which, though different from the continuous time, keep a precise mapping relationship between time and discrete time, retrieve the state machine relative to the time model is not straightforward. The reason is that operations of state machines normally acts according to a logic clear and understandable in response to situations or events that arise from the outside, such as pressing of buttons, the attainment of steady state conditions and so on.

In order to get state machine models to coexist with time-based models, we need to interpret state transitions on the time line used for the time-based portion of the system, be it continuous time or discrete time. The resulting models are called hybrid systems. A hybrid system combines time-based signals with sequences of events. The time- based signals are of the form $x : T \to R$ where R is some range (such as $\mathbb{R}$ or $\mathbb{C}$), and T is the time domain, discrete or continuous. In 2.3.1 we defined the event signals as

$$u : \mathbb{N}_0 \to Symbols$$

for a hybrid system, however, these have to share a common time base with the time-based signals, so they must be in the form

$$u : T \to Symbols$$

Thus, events occur in time. Typically, for most $t \in T$, $u(t) = absent$, meaning that the state machine does not perform any action.

2.3.5 Formal Model

The general structure of an hybrid system is shown in figure 2.3.

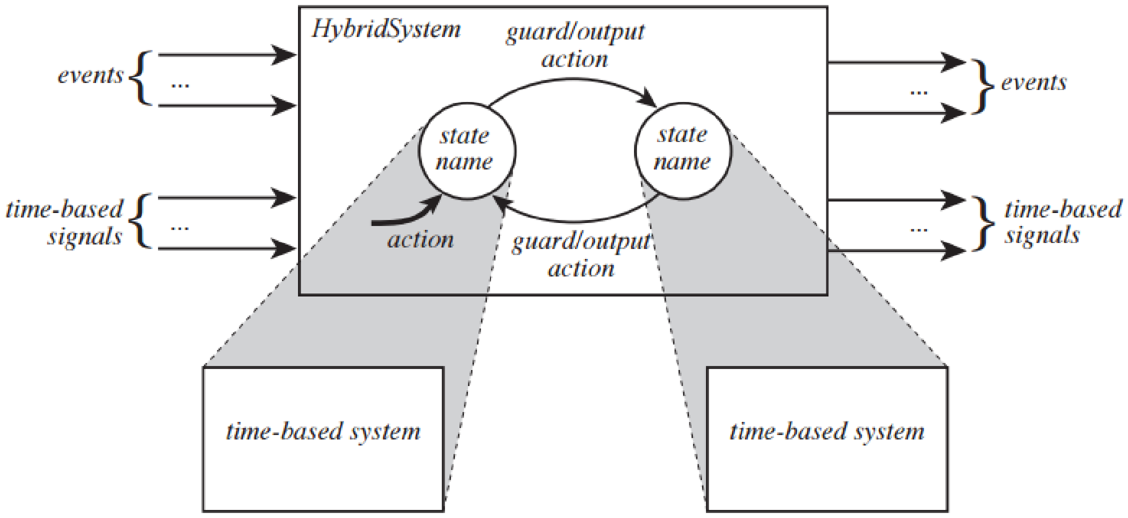


Fig. 2.3 Hybrid System Structure

As previously mentioned, inputs and outputs may include either asynchronous events and time-based signals. In addition, each state of the state machine is associated with a time-based system, which is usually named *refinement* of the state. The refinement represents the time-based behavior of the hybrid system while the internal machine is in the relative state. In order to prevent mess in the notation, those state referring to the state machine are usually called *modes* while with *states* are referred those of the refinements. Such notation strengthen the concept of a systems which alternates different time-based behaviors depending on the operating mode. Summarizing, for *state* of an hybrid system is meant the pair $(m(t), s(t))$ where s is the state of the time-based refinement system associated with mode m.

A formal model for an hybrid system could be defined similarly to the one in 2.3.1, indeed

$$HybridSystem = (States, Inputs, Outputs, TransitionStructure, initialState)$$

where $States, Inputs$ and $Outputs$ are sets, $initialState \in States$ is the initial state and $TransitionStructure$ is the evolving law of the system in time $t \in T$, from now on we assume $T \equiv \mathbb{R}_+$.

$States = Modes \times RefinementStates$ represents the state space containing pairs *mode*, *refinementState* as previously mentioned. The set of possible modes is finite, while no constraint are on the refinement state set.

$Inputs = InEvents \times TimeInputs$ corresponds to the set of input symbols. $InEvents$ is an alphabetic set in which each symbol stands for the description of the event, it also includes the special "do nothing" input *absent*. $TimeInputs$, instead, contains all the inputs to which the refinement reacts. A complete input to the hybrid system is hence the functions pair $u : \mathbb{R}_+ \to InEvents$ and $x : \mathbb{R}_+ \to TimeInputs$. It's useful to recall that, apart for a limited number of cases, the function relative to the events for the most of the time is set to the value *absent*.

$Outputs = OutEvents \times TimeOutputs$ is the set of output symbols. In strong analogy with what already done for the $Inputs$ set, we can identify $OutEvents$ as the alphabetic set containing those symbol representing the description of the event, and $TimeOutputs$ as the one containing all possible outputs for the refinement. The entire output of the hybrid system is then the functions pair $v : \mathbb{R}_+ \to OutEvents$ and $y : \mathbb{R}_+ \to TimeOutputs$. Even in this case for the most of the time the function $v(t)$ is equal to $absent$.

The $TransitionStructure$ determines how a mode transition occurs and how the refinement state changes over time. The evolution of an hybrid system occurs with the alternation of two phases, one associated to the transitioning and the other to the keeping of the current mode. In general, for state machines the transition between states is regulated by the verification of a particular conditions named *guards*. Such conditions consists of an event,internal or external, occurrence. For hybrid system the concept of guard is slightly extended, in particular, given the system in a mode m with relative refinement in the state s, the guard leading to a destination mode d has the form

$$G_{m,d} = U_{m,d} \times X_{m,d} \times S_{m,d} \subset InEvents \times TimeInputs \times RefinementStates$$

commonly, associated to a mode transition there is an output event $v_{m,d}$ and a function *action*, $A_{m,d} : RefinementStates \to RefinementStates$, which updates the value of the refinement state. The transition from mode m to mode d occurs at time t if the triple $(u(t), x(t), s(t)) \in G_{m,d}$. If it is the case, after the transition (which is considered instantaneous) the system starts operating in mode d with refinement state equal to $s(t+) = A_{m,d}(s(t))$, producing the output event $v_{m,d}$. In the case no guard is satisfied at time t the systems evolves in the current mode. The refinement state $s(t)$ and the time-based output $y(t)$ are then determined by the time-based input signal $x(t)$ according to equations governing the refinement dynamics (see Sec.2.3.3). The alternation of the two phases discussed above can be represented as $\bigcup_{i=0}^{\infty} (t_i, t_{i+1}]$, during each interval the active phase is the one who maintain constant the operation mode, while the transition phase take place at the rightmost time instant of every interval.

2.3.6 Examples

Timed Automata The simplest continuous-time hybrid systems that can be analyzed is a timed finite state machine measuring passage of time, it has a very simple refinement dynamics that can be modeled with a first-order differential equation,

$$\dot{s}(t) = 1, \quad \forall t \in T_m$$

where $T_m \subset T$ is the subset of time during which the hybrid system is in mode m. Figure 2.4 shows a timed automata which generates a tick at time intervals alternating between 1 and 2 seconds. Although simple such example shows all the basic features of an hybrid system, namely the alternate evolution between time-count phase and tick-phase (Fig.2.5a), output event signal most of the time at absent value (Fig.2.5c) and refinement output, equal to the state in this case, as a signal resulting from a differential equation (Fig.2.5b).

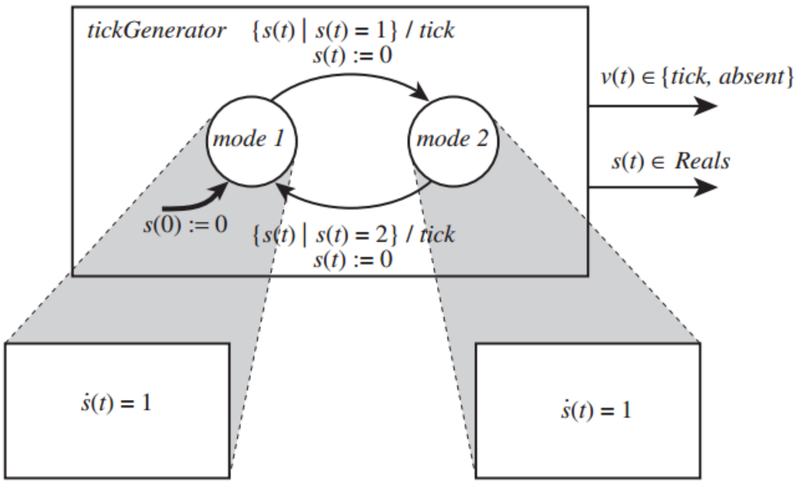
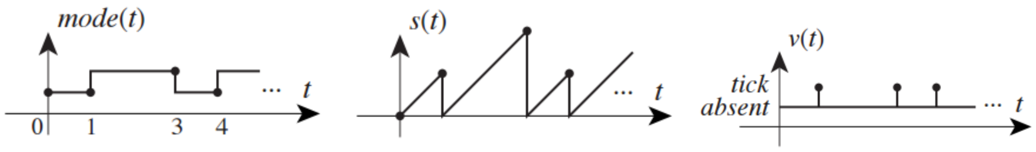


Fig. 2.4 Hybrid System: Tick Generator

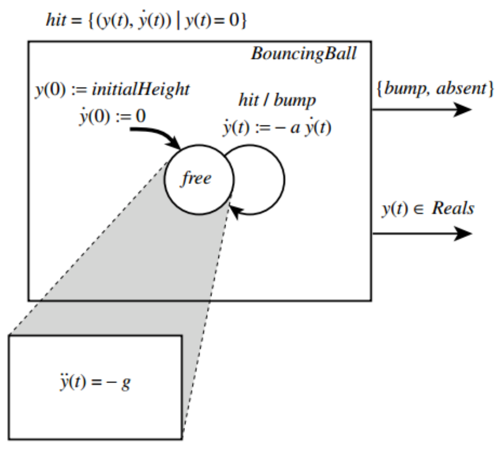


(a) Mode (b) Refinement Out (c) Out Event

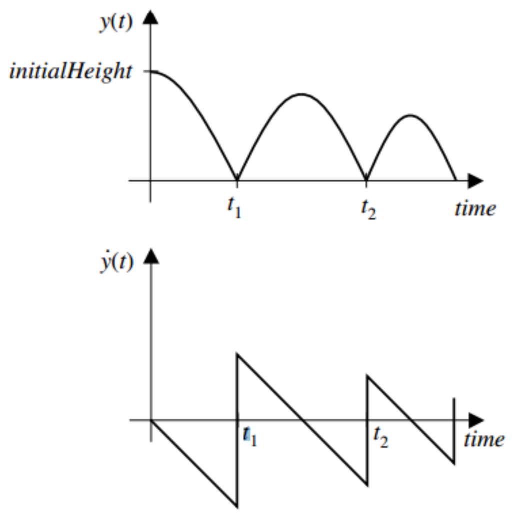
Fig. 2.5 Tick Generator Internals

Bouncing Ball An more interesting example is an hybrid system describing the dynamics of a bouncing ball. In this case the adoption of a mixed model greatly simplifies the representation of such a system, since, being it non-linear, finding a representation only through differential equation is not easy.

At time $t = 0$ the ball is dropped from a certain height $y(0) = h$, it freely falls, with constant velocity $\dot{y}(t) < 0\,m/s$, up to hit the ground, say at time $t1$. In that instant the *bounce* events occurs and, under the assumption of inelastic collision, the ball bounces back with an inverted velocity $-a\dot{y}(t1)$, where $a \in (0,1)$. After the ball reaches a certain height it falls back repeatedly. The model of the bouncing ball is illustrated in Figure 2.6a



(a) hybrid system



(b) height and velocity

Fig. 2.6 Hybrid System: Bouncing Ball

The *hit* condition is raised whenever the height of the ball is equal to 0, namely whenever it reaches the ground. In all other time instants in order to get position and velocity of the ball is enough to double integrate the acceleration (Fig.2.6b).

Chapter 3

Formal Verification

3.1 Temporal Logic

Temporal logic is a rigorous formalism for specifying behaviors of continuous and discrete systems [**?** **?**] provides simple constructs to describe the order in which different "events" in the system should happen. The basic infrastructure of a formal language provides syntactical operators to handle property verification during time. In literature plenty of formalisms have been proposed, each one specializing either time or application specific features. First part of this chapter aims to analyze some of them starting from their common ancestor, providing also a general comparison.

3.1.1 Linear Temporal Logic

In the Linear Temporal Logic (LTL) time is implicitly represented as an enumerated sequence of reaction steps occurring in a discrete time space. Such temporal logics was developed to check properties in (typically hardware) systems with boolean, discrete-time signals.

LTL with *future* and *past* is defined using the following syntax:

$$\varphi := \quad p \quad | \quad \neg\varphi \quad | \quad \varphi \vee \varphi \quad | \quad \bigcirc\varphi \quad | \quad \ominus\varphi \quad | \quad \varphi\,\mathcal{U}\,\varphi | \quad \varphi\,\mathcal{S}\,\varphi$$

where p belongs to a set $P = \{p_1, \dots, p_n\}$ of propositions indicating values of the corresponding state variable. LTL is interpreted over n-dimensional Boolean ω-sequences of the form $\omega : \mathbb{N} \to \mathbb{B}^n$. We use $\omega[t]$ to denote the value of a sequence ω at position t and abuse p to denote the projection of w on variable p. The basic future temporal operators are *next* ($\bigcirc$), which specifies what should hold in the next step and *until* ($\mathcal{U}$), which requires φ_1 to hold until φ_2 becomes true, without bounding the temporal distance to this becoming. From these basic LTL operators one can derive other standard Boolean operators as well as temporal operators such as *eventually* ($\Diamond$) and *always* ($\Box$):

$$\Diamond\varphi = True\ \mathcal{U}\ \varphi \quad \text{and} \quad \Box\varphi = \neg\Diamond\neg\varphi$$

Analogously, by means the past temporal operators, is possible to define *eventually in the past*($\diamondminus$) and *always in the past*($\boxminus$)

$$\diamondminus\varphi = True\ \mathcal{S}\ \varphi \quad \text{and} \quad \boxminus\varphi = \neg\diamondminus\neg\varphi$$

The satisfaction relation $(\omega,t) \models \varphi$ indicating that a sequence ω satisfies φ starting from position t, is defined inductively as follows:

$$\begin{array}{rll}
p & (\omega,t) \models \varphi & \leftrightarrow\ p[t] = 1 \\
not\ \varphi & (\omega,t) \models \neg\varphi & \leftrightarrow\ (\omega,t) \not\models \varphi \\
\varphi_1\ or\ \varphi_2 & (\omega,t) \models \varphi_1 \vee \varphi_2 & \leftrightarrow\ (\omega,t) \models \varphi_1\ or\ (\omega,t) \models \varphi_2 \\
next\ \varphi & (\omega,t) \models \bigcirc\varphi & \leftrightarrow\ (\omega,t+1) \models \varphi \\
previously\ \varphi & (\omega,t) \models \ominus\varphi & \leftrightarrow\ (\omega,t-1) \models \varphi \\
\varphi_1\ until\ \varphi_2 & (\omega,t) \models \varphi_1\mathcal{U}\varphi_2 & \leftrightarrow\ \exists\, t' \in [t,\infty)\ (\omega,t') \models \varphi_1\ and\ \forall\, t'' \in [t,t'),(\omega,t'') \models \varphi_2 \\
\varphi_1\ since\ \varphi_2 & (\omega,t) \models \varphi_1\mathcal{S}\varphi_2 & \leftrightarrow\ \exists\, t' \in [0,t]\ (\omega,t') \models \varphi_1\ and\ \forall\, t'' \in (t',t],(\omega,t'') \models \varphi_2 \\
\\
eventually\ \varphi & (\omega,t) \models \Diamond\varphi & \leftrightarrow\ \exists\, t' \geq t\ (\omega,t') \models \varphi \\
always\ \varphi & (\omega,t) \models \Box\varphi & \leftrightarrow\ \forall\, t' \geq t\ (\omega,t') \models \varphi
\end{array} \tag{3.1}$$

A major difficulty in checking properties expressed in future LTL is due to the *non-causal* definition of the satisfaction relation. In other words, the satisfiability of φ at time t may depend on the value of ω at some future time instant $t' \geq t$. Even worse, some temporal operators refer to future time instants in a *quantified* manner, for example, requiring some p to hold in all future time instants. The satisfiability of such a property may sometime be determined only at infinity, that is, "after" we can be sure that no instance of $\neg p$ is observed.

The causality problem is not present anymore when the recursion goes backward in time, meaning that the satisfaction of a past formula φ by a sequence ω at position t is determined only according to the values of ω at the interval $[0,t]$. However the futuristic specification keeps a style more natural for humans, for this reason it has been adopted also by industrial specification languages such as PSL (3.1.3).

Evaluation over incomplete behavior Monitors do not exploit features of the model representing the system S, but rather observe sequences, i.e. model's outputs, as they are produced during simulation. Although the LTL semantic is defined over *complete infinite sequences*, that is for all possible behaviors, is not possible to observe infinite sequence in finite time. Hence, trying to extend the LTL semantic to *incomplete behaviors* represents the hardest challenge in monitoring.

At the end of the observation of a certain sequence ω we can assert one of the following with respect to the property φ

1. ω *satisfies* φ. Such situation happens, for example, when $\varphi = \Diamond p$ and p occurs in ω.

2. ω *not satisfies* φ. For example, when $\varphi = \Box\neg p$ and p occurs in ω.

3. ω *is undecided.* For example, when $\varphi = \Diamond p$ and p has not yet occurred in ω.

The category "undecided" can be further refined in order to distinguish, for instance, the "not yet violated" ($\varphi = \Box\neg p$) category from the "not yet satisfied" one ($\varphi = \Diamond p$).
Even for those cases in which the satisfiability of a property cannot be determined from the observed behavior we would like to give an answer at the end of the sequence. In order to achieve such objective we may consider some LTL sub-classes. One is the bounded-LTL. It provides only *next* as temporal operator, which, in turn, can be used to derive others such as bounded-always and bounded-eventually

$$\Box_{[0,r]}\varphi = \bigwedge_{i=0}^{r-1} \bigcirc^i \varphi \quad \text{and} \quad \Diamond_{[0,r]}\varphi = \bigvee_{i=0}^{r-1} \bigcirc^i \varphi$$

where $\bigcirc^i$ is a shorthand for $\bigcirc(\bigcirc(\cdots\bigcirc)p\ldots))$. Within this context of timed formulas a global property that has not been violated during the formula's lifetime is considered satisfied. Analogously a property expressing an eventuality not observed during the formula's lifetime is considered violated.

3.1.2 Computation Tree Logic

LTL formulas define properties referring individual executions, or simulations, that is, operators are provided for describing events along a single computation path.

The Computation Tree Logic (CTL) [**?**] extends this concept by expressing formulas with respect to many executions, or simulations, at once. In this sense it belongs to the family of *branching time logic*, in which operators quantify over the paths that are possible from a given state. The main feature provided by logic such as CTL is the possibility to add temporal connectives to the usual the usual atomic propositional logic formulas. The temporal connectives are expressions about paths into the future that the state of the system can follow.

The occurrence of a CTL formula is specified by means the pair *path*, *temporal* quantifiers. The first regards the possible execution paths that the system can follow starting from the current one. The second, instead, relates the occurrences within each single path.

It's clear how LTL logic represents a subset of CTL, in fact an LTL formula can be view as a CTL formula without the specification of the branch quantifier. However, part of the literature uses identify with CTL the sub-logic related to path and with CTL* the one which unifies both path and temporal quantifiers. According that, LTL and CTL becomes two intersecting set, while CTL* is the superset containing both. Anyway, for the sake of simplicity, in what follows we will make no difference between CTL and CTL*, using the first in place of the second.

When evaluating a CTL formula the first member to be considered is the path quantifier. It can be

A meaning on *all* the path from the current state, read as "*inevitably*"

E meaning on *at least one* path from the current state, read as "*possibly*"

Once defined the occurrence among the paths, the second member to consider is the temporal quantifier, i.e.

X meaning the next state

G meaning all the future states, read as "*globally*".

F meaning in some future state

U meaning until

Combining the previous operators is possible to generate different types of formulas, suppose that the system is in some state S the formula

φ is true iff it is satisfied by the current state S

AX(φ) is true iff φ is true for every immediate successor of state S

AG(φ) is true iff φ is true for every successor to state S, including S. That is, φ is true for all states on all paths into the future from S, i.e. the subtree originating from S

AF(φ) is true iff on all paths into the future from S, there is a state where φ holds

A [φ_1 **U** φ_2] is true iff all paths starting in state S satisfy φ_1 until the reach a state in which φ_2 holds

EX(φ) is true iff φ is true for at least one immediate successor to state S

EG(φ) if true iff there is a path from S into the future for which φ holds for every state on the path, including S

EF(φ) is true iff there exists a path into the future from S on which there is a state where φ holds

E [φ_1 **U** φ_2] is true iff there exists a path starting in state S that satisfies φ_1 until reaching a state in which φ_2 holds

A graphical representation of the above formulas is shown in Figure 3.1

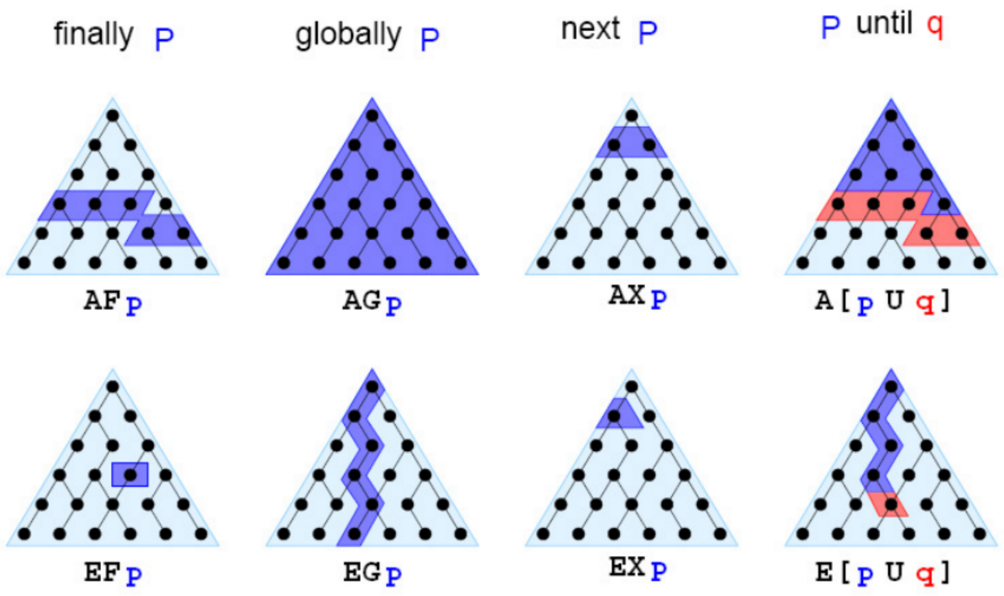


Fig. 3.1 CTL formulas

A subset Computation Tree Logic is supported for verification in UPPAAL, which is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata [**?**].

3.1.3 Property Specification Language

The Property Specification Language (PSL) [**?**] is a formal language for specification of electronic system behavior, compatible with multiple electronic system design languages. In 2005 it became an IEEE standard [**?**]. PSL was designed to be mathematically rigorous, with the result that a PSL specification is both precise and automatically verifiable. Thus, a hardware specification written in PSL is machine readable and can be used as input to verification tools.

Being a specification language whose targets is the hardware, signals in PSL are used as stream of bits having a fixed length, and , for the same reason, time is meant as a sequences of *clock-cycles*. The structure of PSL is based on four layers

– The *boolean layer* is composed of Boolean expressions. PSL interprets a high signal as true, and a low signal as false, independent of whether the signal is active-high or active-low. Operations of the Boolean layer are expressed through the supported hardware specification languages such as Verilog, VHDL and so on.

– The *temporal layer* consists of temporal properties which describe the relationships between Boolean expressions over time. The semantic of temporal operators is quite similar to the LTL ones.

– The *verification layer* consists of directives which describe how the temporal properties should be used by verification tools. For instance, is possible to define verification or assumption directives, which tells the tools to verify the value of some property or specify coverage criteria. The verification layer also provides a means to group PSL statements into *verification units*.

– The *modeling layer* provides a means to model behavior of design inputs, and to declare and give behavior to auxiliary signals and variables

While the Boolean layer consists of Boolean expressions that hold or do not hold at a given cycle, the temporal layer provides a way to describe relationships between Boolean expressions over time. A PSL assertion typically looks in only one direction – forwards from the first cycle. Thus, the simplest PSL assertion **assert a;** states that signal **a** should hold at the very first cycle.
The PSL temporal operators are the following

always (p**)** meaning that property p must *globally* hold

never (p) meaning that property p must *never* hold

next (p) will hold if its operand, p, holds at the next clock-cycle. Variations on the next operator allows to specify ranges of future cycles, i.e. a **next a[i:j]**(p) property holds if its operand holds in all of the cycles from the i^{th} next cycle through the j^{th} next cycle, inclusive.

p **until** q provides a way to move forward, meaning that property p must hold *until* q hold. In order to include the cycle in which q holds the **until_** is normally used.

p **before** q is dual to **until** and requires that its first operand happen strictly before its second. Even for this case in order to include the cycle in which q holds, namely is allowed an overlap between left and right sides, the **before_** is used.

eventually!(p) allows you to specify that p must occur in the future without saying exactly when.

A good feature of PSL is that it let user choose how the formulas are evaluated in cases of incomplete behavior. Such problem, already discussed in section 3.1.1, rises whenever the simulation, or execution, time is not enough to determine the satisfiability of a formula. PSL handles this providing the possibility to define the strength of an operator, therefore operators can be claimed as *strong* or *weak* depending on the addition of an exclamation mark (**!**) to their names. Formulas with strong operators, marked with (**!**), are considered satisfied if the simulation time is *enough* and the property holds, for instance **next![n]** indicate that at least n cycles are needed. Some operators, anyway ,constitutes an exception since they are only allowed in their weak (**always**, **never**) or strong (**eventually!**) versions. Moreover, PSL offers many other versions of **next** associated with occurrence of events rather than cycles, for further details refer [**?**].

3.1.4 Signal Temporal Logic

Signal Temporal Logic (STL) [**?**] is a temporal logic for specifying properties on *dense-time real-valued signals*. Such a logic is particularly useful in domains like *control systems*, where continuous variables are used to model the physical plant under control. The natural models for such systems are differential equations, for purely continuous systems, or hybrid systems when the dynamics is mixed and contains mode switching, saturation, etc. Before going into details of STL is useful to analyze it's direct ancestor MITL.

Metric Interval Temporal Logic The *real-time temporal logic* MITL [**?**] allows reasoning over Boolean signals over dense-time domains. Formally, signals are functions of the form $s : \mathbb{T} \to \mathbb{B}$, where the time domain $\mathbb{T}$ is the set of non-negative real numbers $\mathbb{R}_+$.

We consider the logic $\mathrm{MITL}_{[a,b]}$ as a fragment of MITL, such that all temporal modalities are restricted to intervals of the form $[a,b]$ with $0 \leq a < b$ and $a,b \in \mathbb{Q}+$. Similarly to 3.1.1, the use of bounded temporal properties is justified by the nature of monitoring where the behavior of a system is observed for a finite time interval. The basic formulas of $\mathrm{MITL}_{[a,b]}$ are defined by the grammar

$$\varphi := \quad p \quad | \quad \neg\varphi \quad | \quad \varphi \vee \varphi \quad | \quad \varphi_1 \, \mathcal{U}_{[a,b]} \, \varphi_2$$

From basic MITL[a,b] operators one can derive other standard Boolean and temporal operators, in particular the time-constrained eventually and always operators:

$$\Diamond_{[a,b]}\varphi = True \, \mathcal{U}_{[a,b]} \, \varphi$$
$$\Box_{[a,b]}\varphi = \neg\Diamond_{[a,b]}\neg\varphi$$

The semantic of the unbounded operators is equivalent to the one provided for LTL (3.1), while the introduction of the time window modifies *until*, *eventually* and *always* as follow

$$(s,t) \models \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 \leftrightarrow \exists\, t' \in [t+a, t+b] \; (s,t') \models \varphi_2 \; and \; \forall\, t'' \in [t,t'], (s,t'') \models \varphi_1$$
$$(s,t) \models \Diamond_{[a,b]}\varphi \leftrightarrow \exists\, t' \in [t+a, t+b], (s,t') \models \varphi$$
$$(s,t) \models \Box_{[a,b]}\varphi \leftrightarrow \forall\, t' \in [t+a, t+b], (s,t') \models \varphi$$

By using bounded operators we avoid the problems related to the ambiguity of $\models$ when applied to finite signals or sequences. Nevertheless, even for $\mathrm{MITL}_{[a,b]}$ certain signals are too short to determine satisfaction of the formula, for example the property $\Box_{[a,b]}\Diamond_{[c,d]}p$ cannot be evaluated on signals shorter than $b+d$.

We can now extend our semantic domain and logic to real-valued signals. While Boolean signals of finite variability admit a finite representation, this is typically not the case for real-valued signals which are often represented via sampling, that is a sequence of time stamped values of the form $(t, s[t])$. Although the semantics of the logic is defined in terms of the mathematical objects, for signals of the from $s : \mathbb{T} \to \mathbb{R}^n$ is not possible to ignore issues related to their effective representation based on the output of some numerical simulator. For this reason STL does not directly cope with continuous signal, but rather via a set of abstraction of the form $\mu : \mathbb{R}^n \to \mathbb{B}$. Typically μ partitions the continuous state-space according to the satisfaction of some inequality constraints on the real variables. As long as

$\mu(s[t])$ remains constant we do not really care about the exact value of $s[t]$. However, in order to evaluate formulas we need the sampling to be sufficiently dense so that all such transitions can be detected when they happen. From now on we assume that we deal with signals that are well-behaving with respect to every μ, that is, every change in $\mu(s)$ is detected in the sense that every point t such that $\mu(s[t]) \neq \lim_{t' \to t} \mu(s[t]')$ is included in the sampling.

We can define an STL formula as a $\text{MITL}_{[a,b]}$ formula over the atomic propositions $\mu_1(s(t)), \ldots, \mu_m(s(t))$, where each μ_i is a predicate of the form $\mu_i : \mathbb{R}^n \to \mathbb{B}$. The monitoring process for STL formula decomposes hence into two phases, in which the first is always the construction of a Boolean "filter" for every $\mu_i \in U = \{\mu_1, \ldots, \mu_m\}$, which transforms s into a Boolean signal $p_i = \mu_i(s)$.

3.1.5 Summary

3.2 Patterns Identification

The drawback to using temporal logics for property specification is their steep learning curve for industrial practitioners. Consequently, designers and developers will be less likely to use verification tools if they must devote large amounts of time to learning a specification language. Even with significant expertise, dealing with the complexity of such a specification could be daunting. As many other disciplines, like *software engineering*, complexity in this context is addressed through the definition and use of common *patterns*. Patterns are meant as a further levels of abstraction, parameterizable and often formalism-independent.

3.2.1 Property Specification Patterns

Property specification patterns [**?**] describe commonly observed requirements in a generalized manner. Observing the several formalisms introduced before is possible to notice how there are two basic parts of properties that commonly occur. The first tells when the property should hold, and the second tells what condition should be satisfied during this time. Precisely each formula consists of two pieces: a *scope* and a *pattern*. The scopes defines *when* a particular property should hold during the simulation, the pattern, instead, the condition that must be satisfied. The are five basic kinds of scopes that can be recognized,

Globally meaning the entire simulation

Before R meaning the entire execution up to the occurrence of **R**

After Q meaning the entire execution from the occurrence of **Q**

Between Q and R any piece of simulation from the occurrence of Q and the occurrence of **R**

After Q until R like above but holds even if **R** does not occur

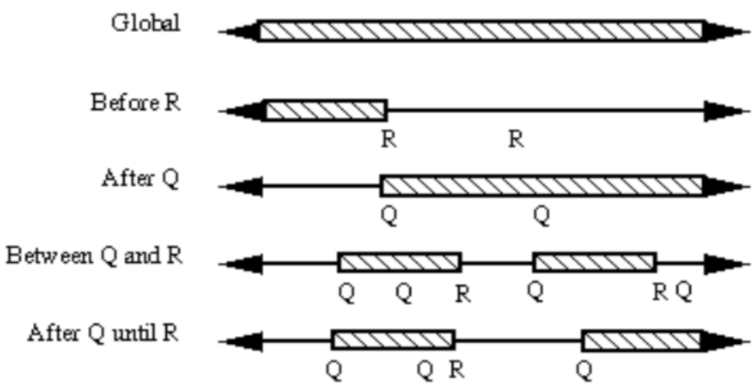


Fig. 3.2 Property Scopes

Figure 3.2 illustrates the portions of an execution that are designed by the different kind of scopes.

Patterns are organized in a hierarchical manner based on their semantic. A the first level they are divided by *occurrence* and *order*. Firsts are used to express requirements related to the existence (or lack of existence) of certain states/events during well-defined regions of system execution, while seconds are used to express requirements related to pairs of states/events during well-defined regions of system execution. For both categories regions are defined using *scopes*. In the class of Occurrence patterns we find

P **is false (Absence)** to describe a portion of a system's execution that is free of certain events or states.

P **is true (Universality)** to describe a portion of a system's execution which contains only states that have a desired property.

P **becomes true (Existence)** to describe a portion of a system's execution that contains an instance of certain events or states.

P **occur at most** *N* **times (Bounded Existence)** to describe a portion of a system's execution that contains at most a specified number of instances of a designated state transition or event.

while Order patterns are

S **precedes** *P* **(Precedence)** to describe relationships between a pair of events/states where the occurrence of the first is a necessary pre-condition for an occurrence of the second.

S **responds to** *P* **(Response)** to describe cause-effect relationships between a pair of events/states.

$Q_1 \dots Q_n$ **precedes** $P_1 \dots P_n$ **(Precedence Chain)** to describe a relationship between a sequence of events/states and a sequence of events/states. The occurrence of sequence $P_1 \dots P_n$ must be preceded by an occurrence of the the sequence $Q_1 \dots Q_n$.

$P_1 \dots P_n$ **responds** $Q_1 \dots Q_n$ **(Response chain)** to describe a relationship between stimulus events and a sequence of response events. The occurrence of the stimulus $P_1 \dots P_n$ must be followed by an occurrence of the sequence $Q_1 \dots Q_n$.

Clearly patterns are not unique in the sense that one can be achieved by combination of some others. The Absence, for instance, is dual to Existence, while Precedence is converse to Response. This redundancy does not represents an issue but rather extends the usability of those patterns even in formalisms that not provide complete support for all scope operators. Figure 3.3 provides an overview of patterns classification.

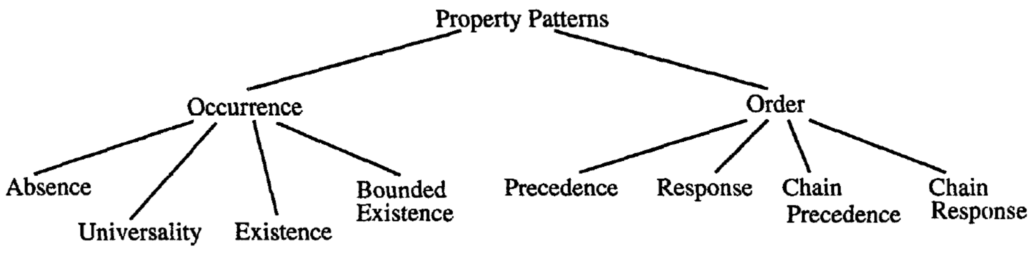


Fig. 3.3 Pattern Hierarchy

Being each pattern defined over a scope, the number of possible achievable combinations corresponds to the dimension of the set generated by the product of scope and pattern sets. Each combination could be mapped in a specific requirements, therefore it can be independently translated in a chosen formalism. For the sake of briefness we show the complete LTL an CTL formalization for just a pattern, and refer [**?**] for an exhaustive list.

Before describe the pattern lets define the *weak until* operator $\mathcal{W}$ which is used make less verbose the expression of some formulas.

$$\text{LTL:} \quad P \,\mathcal{W}\, Q = P \,\mathcal{U}\, (Q \mid \Box P)$$
$$\text{CTL:} \quad P \,\mathbf{W}\, Q = \neg\mathrm{E}\,(\neg Q \,\mathbf{U}\, (\neg P \,\&\, \neg Q))$$

Universality ***P* is true**

Scope	*LTL*
Globally	$\square\, P$
Before R	$A\, R \rightarrow (P\, \mathcal{U}\, R)$
After Q	$\square\, R \rightarrow (Q \rightarrow \square\, P)$
Between R and Q	$\square\, ((Q \,\&\, \neg R \,\&\, \Diamond\, R) \rightarrow (P\, \mathcal{U}\, R))$

Scope	*CTL*
Globally	$\mathbf{AG}(P)$
Before R	$\mathbf{A}((P \mid \mathbf{AG}(\neg R))\ \mathbf{W}\ R)$
After Q	$\mathbf{AG}(Q \rightarrow \mathbf{AG}(P))$
Between R and Q	$\mathbf{AG}(Q \,\&\, \neg R \rightarrow \mathbf{A}((P \mid \mathbf{AG}(\neg R))\ \mathbf{W}\ R$
After Q unitl R	$\mathbf{AG}(Q \,\&\, \neg R \rightarrow \mathbf{A}(P\ \mathbf{W}\ R))$

3.2.2 Domain Patterns

When dealing with complex systems quite often the requirements analysis is performed at different levels of abstraction. As result the complete specification is provided by many requirements documents, in which requirements of a certain level are grouped according to derivation from a *parent requirement* belonging to an higher level.

A typical scenario for control systems is that high-level requirements focus all those properties specific of that application domain. Examples are the so called *performance requirements*, which assess features of the system response over time. Since they refer properties having a well-known mathematical representation such kind of requirements in turn are prone to have a parameterizable definition. In other words they are suitable to become domain specific patterns.

Two typical performance requirements expressed in natural language are the following

1. *The Driver System (DRV) shall accelerate the motor from zero to x_1 rpm in less than t_1 sec, with an overshoot of less than x_2 rpm and a time to settle to within $\pm x_3$ rpm of x_4 rpm of less than t_2 ms with a system inertia less than or equal to S_{In}*
2. *The Driver System (DRV) shall decelerate the motor from x_1 to zero rpm in less than t_1 sec, with an undershoot of less than x_2 rpm and a time to settle to within $\pm x_3$ rpm of zero rpm of less than t_2 ms with a system inertia less than or equal to S_{In}*

An informal definition of the addressed requirements is the following

Rise(Fall)-time is the amount of time it takes for a signal to rise (fall) from an initial value to another value some distance from an expected steady state value in response to a step input (Fig.3.4)

Overshoot (Undershoot) is a quantity that defines how much a signal goes beyond an expected steady state value in response to a step input (Fig.3.5)

Settling-Time is the time it takes a signal to reach and remain within a band around a steady state value in response to a step input (Fig.3.6)

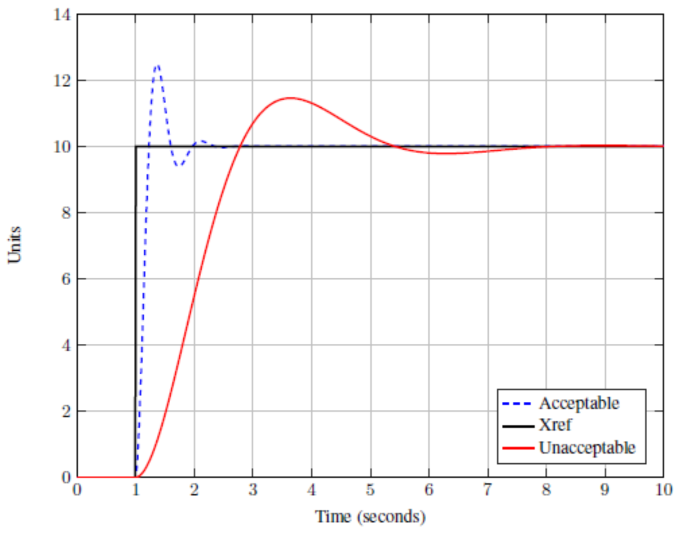


Fig. 3.4 Rise Time

In [**?**] J.Kapinski et al provided an STL formalization for above property. Since all of them are defined in response to an asynchronous step input they extended the STL language to provide the operator

$$step(x,a) \triangleq x(t+\varepsilon) - x(t) > a$$

where a is the step amplitude and ε corresponds to the smallest time variation following t. In the case the formula has to be verified by a discrete time monitors ε is equal to the sample time.

In STL, timed formulas can be nested such as, for example,

$$\Diamond_{[0,T]}(\mathbf{q} \wedge \Box_{[a,b]}\mathbf{p})$$

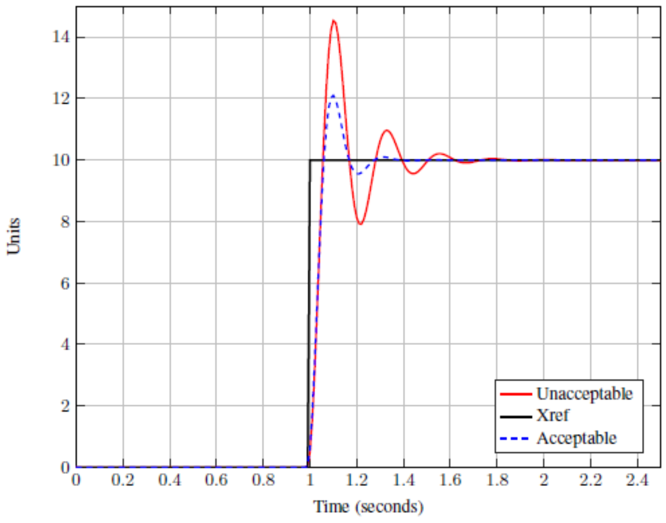


Fig. 3.5 Overshoot

The proposition **p** is nested one level deeper than proposition **q**. The meaning is that there has to be one time instant t in $[0,T]$ (the outer Eventually condition) such that **q** is satisfied in t and for all the system evolutions starting from time t, the condition **p** is verified at some time between $t+a$ and $t+b$. In a runtime monitor implementation, the evaluation of the global condition with **p** depends not only on the time range of its temporal operator, but also on the time t in which **q** is satisfied. If t_q is the time at which **q** is satisfied, the time range in which **p** is evaluated becomes $[a+t_q, b+t_q]$. The nested time interval $[a,b]$ is therefore not an absolute time, but is relative to the time instant identified by the outer clause.

The STL expression of the performance requirements comes in a more readable shape if formulas aim to prove the presence of the property instead of their absence:

$$\begin{array}{ll}
RiseTime(ref,x,a,t_1,x_1) & \Diamond_{[0,T]}(step(ref,a) \wedge \Box_{[0,t_1]}(x < x_1)) \\
FallTime(ref,x,a,t_1,x_1) & \Diamond_{[0,T]}(step(ref,a) \wedge \Box_{[0,t_1]}(x > x_1)) \\
Overshoot(ref,x,a,x_2) & \Diamond_{[0,T]}(step(ref,a) \wedge \Diamond(x - ref > x_2)) \\
Undershoot(ref,x,a,x_2) & \Diamond_{[0,T]}(step(ref,a) \wedge \Diamond(ref - x > x_2)) \\
SettlingTime(ref,x,a,t_2,x_3) & \Diamond_{[0,T]}(step(ref,a) \wedge \Diamond_{[t_2,T]}(|x - ref| > x_3))
\end{array}$$

where T is the simulation time, ref is the command to the system and x is its output. In **REF** each pattern is coupled with a natural language representation, doing so we create the connection from natural to formal language, avoiding, at least for this class of requirements ,ambiguities and errors.

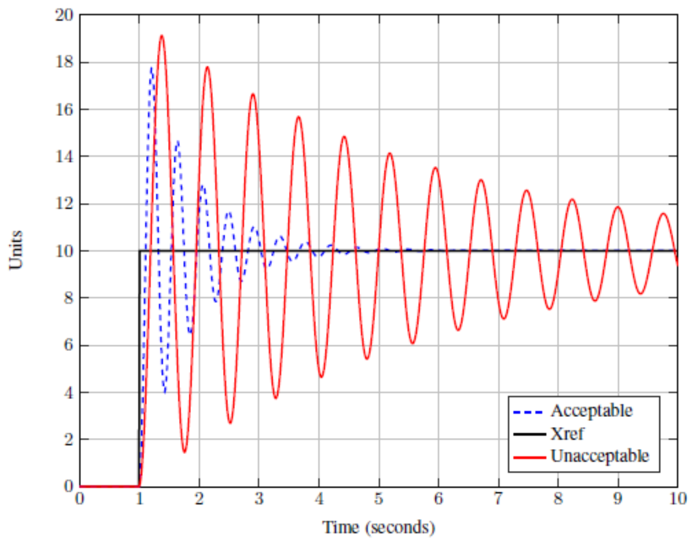


Fig. 3.6 Settling Time

3.3 The Contract Paradigm

The contract-based paradigm, founded on the use of contracts as formal requirements, allows distributed designers to develop different aspects and components of the overall system in a concurrent but controlled way. In a component-based model, a component is a hierarchical entity that represents a unit of design and components are interconnected and communicate through ports carrying discrete or event values. Implementations and requirements can be attached to components, where requirements are expressed as *contracts* [**? ?**]. A contract is an assertion on the behaviors of a component (the promise) subject to certain assumptions. An assertion represents a set of runs of the component and a run can be seen informally as a sequence of values of the component's variables and ports. Therefore a contract C is the pair of assertions

$$C = (A, G)$$

where A corresponds to the assumption, and G to the promise. The component's contract (A, G) corresponds to the requirement $A \rightarrow G$ or equivalently $G \cup \neg A$ on the implementation of the component. A component's implementation M satisfies the requirement if $M \subseteq G \cup \neg A$, in which case the implementation is said to satisfies the contract.

With reference to both requirements of 3.2.2, the assumption under which system performances have to be validated is specified in the subsentence *"...with a system inertia less than or equal to S_{In}"*. In fact it represents the environmental condition under which the system has to guarantee a specific behavior. By coupling the definition of domain patterns with the contract formalization those requirement can be restructured as follows

Req. ID	Formalization $C = (A,G)$
$R.01$	$(inertia \leq S_{In},\ \neg RiseTime(ref,x,a,t_1,x_1) \wedge \neg Overshoot(ref,x,a,x_2) \wedge \neg SettlingTime(ref,x,a,t_2,x_3)\ )$
$R.02$	$(inertia \leq S_{In},\ \neg FallTime(ref,x,a,t_1,x_1) \wedge \neg Undershoot(ref,x,a,x_2) \wedge \neg SettlingTime(ref,x,a,t_2,x_3)\ )$

Table 3.1 Formalized Performance Requirements

In complex models with multiple entities the interactions can be regulated by contracts of single components. Moreover, the system specification and validation can be conduced by combination of all components' assumptions and guarantees. Benveniste et al in [**?**] provide different modalities of combining requirements expressed as contracts.

It is common that each requirement reflect the interpretation of the customer needs related to a single aspect of the design and under some implicit or explicit hypothesis, that represents the precondition under which the behavior described in the requirement should be guaranteed. These *preconditions* do not represent the system's assumptions since they are not absolute constraints that the environment must satisfy. Instead, they model the enabling conditions under which the proposed behavior must be exposed by the system. As example, such preconditions are identified in performance requirement by the verification of a step input, since it acts as trigger for the evaluation of the effective promise related to the property. In [**?**] Mangeruca et al deeply analyze this difference, providing also a richer definition of the classical contract paradigm as the triple

$$C = [A,(P,Q)]$$

where A is the assumption, P is the precondition and Q is the guarantee. The newer semantics of the contract is then represented by the logical formula $A \rightarrow (P \rightarrow Q)$. For an in-depth exploration of such extended contracts please refer the reading.

Chapter 4

The Framework

4.1 From Requirements to Monitors

The components structure of the developed framework is shown in Fig. 4.1.

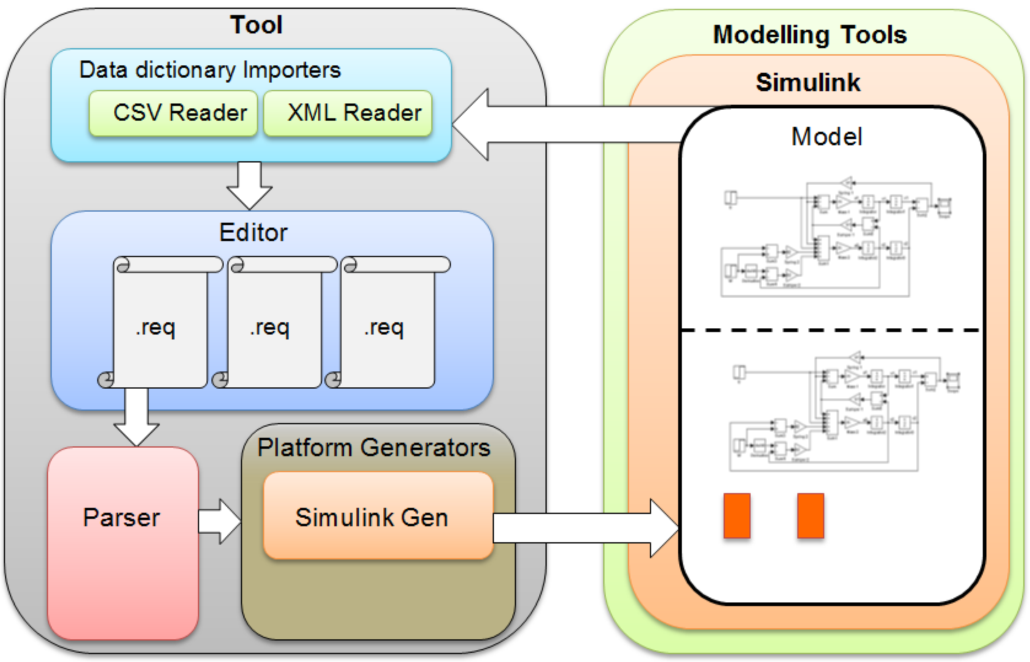


Fig. 4.1 Requirement Framework

Model to Tool The tool cooperates with a modeling environment through two actions. The first consists of using a model reader to import the list of all model's signals and parameters into a data dictionary. This allows to write requirement referring entities directly with the same names as they are specified in the model. In addition to names, the import phase concur to gather information about data types, boundary values and measuring units.

Tool to Model The second actions consists of populating the model with monitors generated out of the requirements. Such action is strictly dependent on the platform on which the model is defined. For this reason the tool has been designed to be as flexible as possible, by providing a set of *Platform Generators* modules, each one for a specific target platform. In general, in order to ease its task, a platform generator could be supported with some platform libraries enclosing either operators or atomic instructions. In the case of Simulink the produced result is a script which, once executed, adds new requirements blocks to the provided model.

The Editor In order to write requirements the tool offers a smart editor. It provides the most common user relieving features, such as the context-aware syntax completion ad coloring. Requirements are written according to a specific loose syntax, which is basically equivalent to an ordered sequences of natural language keywords. Even in this case, due to the large variability of possible syntaxes, the tool shows flexibility and provide support for syntax updates with very low effort.

The Parser The syntax accepted by the editor is not as formal as a the one of a programming language, this complicates not a little the translation of a requirement as it is written. Therefore, before being delivered to a platform generator, the content of a requirement document is transformed into a syntax tree by means of a *Mid Level Parser*. This action, although adds a further step to the synthesis process, ease the task of the platform generator since restructures its input in order to be programmatically analyzable.

Following sections describe in depth all the components. As first version the tool support interactions only with Simulink, which is one of the most used modeling tools. However, for the sake of completeness, during explanation particular attention will be pose to emphasize all techniques adopted, during the development, to enforce extensibility of the tool. For this reason, also a brief survey of common Design Patterns will be conduced in 4.3.

4.1.1 Syntax Definition

The syntax of a requirement document has been developed in order to define requirements as contracts, the guarantee will be referred as an assertion, while the concept of precondition has not been explicitly used.

The grammar of the syntax is proposed as a list of recursive rules, each mandatory rule is enclosed within angular parentheses, while an optional one is enclosed within square parentheses. At the end of the recursion, rules are primitive, in the sense that they could be expressed through primitive data types (String, Integer, . . .).
The entire syntax is defined as follow

```
Requirement Document = <Requirement List>
Requirement List = <Requirement>[Requirement List]
Requirement = <Header><Assumptions Section><Assertions Section>
Header = <ID><Title>
ID = 'R'<Requirement Number>
Requirement Number = <unsigned int>['.'Requirement Number]
Title = <string>
Assumption Section = <"ASSUMPTION"><Assumption List>
Assumption List = <Assumption>["AND" Assumption List]
Assumption = <Comparison statement> | <Signal Generator> | <Reference>
Comparison Statement = <Signal ID><Intermission><Comparison-op><SPV>
SPV = <Signal ID> | <Parameter ID> | <Value>
PV = <Parameter ID> | <Value>
Signal ID = FROM_DATA_DICT
Parameter ID = FROM_DATA_DICT
Value = <double>
Intermission = [Free-text]<"is" | "not">[Free-text]
Free-text = <string>
Comparison-op = <L_EQ> | <G_EQ> | <EQ>
L_EQ = <"less than">["or" EQ]
G_EQ = <"greater than">["or" EQ]
EQ = <"equal to">
Signal Generator = <Signal ID><"is"><Generator Type>
Generator Type = <Step> | <Ramp>
Step = <Step Type><"with amplitude"><PV><"beginning in"><PV>
Step Type = <"step"> | <"downstep">
Ramp = <"ramp with amplitude"><PV><",duration"><PV><"beginning in"><PV>
Reference = <Signal ID><"is reference">
Assertion Section = "ASSERTION"<Assertion List>
Assertion List = <Assertion>["AND" Assertion List]
Assertion = <Control Assertion>
Control Assertion = <Overshoot> | <Undershoot> | <Rise_Fall Time> |
      <Settling Time>
Overshoot = <Signal ID><"overshoot shall be less than"><PV>
Undershoot = <Signal ID><"undershoot shall be greater than"><PV>
Rise_Fall Time = <Signal ID><"shall"><Rise_Fall><"from"><PV><"in less
    than"><PV>
Rise_Fall = <"rise"> | <"fall">
```

```
Settling Time = <Signal ID><"time to settle to"><PV><"shall be less
        than"><PV>
```

4.2 Support from Modeling Environment

The cooperation of the requirements tool with a modeling environment could be easier if from the latter's side are provided some utilities. A good practice when writing requirements is to fill a data dictionary with system's components names and always refer them with that. To enforce the use of a data dictionary a modeling environment may offer the capability to partially generate it by exporting all entities in the model. From the other side, the process of auto-generate monitors for a targeted modeling environment can be lightened if the requirement tool could make use of built-in libraries. The libraries used to provide support for monitors generation have been already described in [**?**]. Since this thesis is meant as continuation of that work such libraries are reanalyzed, possibly providing further details, also in this context. Next sections explore in a more detailed way these utilities, providing also examples in the case of Simulink.

4.2.1 Signal Exporter

A Simulink model can be view as a collection of connected blocks, where each connecting line represents a signal. In complex models the number of signals is in the order of thousands. The environment allow to name every signal but, for the sake of practicality, does not force users to do so. Although naming, at least the more significant, signals indisputably represents a good modeling practice, rarely professional models comply with such rule. The common situation is that in real models names are provided only for blocks and subsystems' ports, which are not allowed to be unnamed.

In order to be effective, a good Simulink signals exporter cannot avoid to consider the above aspect. In other words, it cannot rely on the fact that all meaningful signals' lines are named, and, somehow, has to overcome this problem. As possible solution, the signals exporter can, in first instance, collect all the lines' names and then continue with all the subsystems' output ports' names. Such approach allows to pick up a broad number of signals' names, however, it does not prevent from possible inconsistencies. As example, it does not produce results for models not defining subsystem blocks and for which no names are associated to lines. Further, since it prioritize lines rather than ports, for models in which a line and a port have the same name, which is perfectly legal in Simulink, the exporter will consider just one name for both. These limitation, however, represents borderline situations which are also deprecated by modeling practices.

A signals exporter can be implemented with the following Matlab function, it requires as input the model and the file to fill with the signals' names.

```
function signalExport(sys, fname)
    ssys = find_system(sys, 'BlockType', 'SubSystem');
    ssys = {sys,ssys{1:end}}';
    lines = find_system(ssys,'FindAll','on','type', 'line');
    signals = containers.Map;
    for i=1:length(lines)
        conn = get_param(lines(i), 'Connected');
        name = get_param(lines(i), 'Name');
        if (size(name, 2) > 0 && strcmp(conn, 'on'))
            signals(name) = 1;
        end
    end
    ports = find_system(ssys, 'BlockType', 'Step');
    ports = [ports ; find_system(ssys, 'BlockType', 'Ramp')];
    ssys = ssys(2:end);
    ports = [ports ; find_system(ssys, 'BlockType', 'Outport')];
    onames = get_param(ports ,'Name');
    t = isKey(signals, onames);
    allkeys = [keys(signals) onames(t==0)'];
    signals = containers.Map(allkeys, ones(length(allkeys),1));
    file = fopen(fname,'a+');
    k = keys(signals);
    for i=1:length(k)
        fprintf(file, 'signal,%s,,,,,\n', strjoin(k(i)));
    end
    fclose(file);
end
```

In addition to line and subsystems' outputs, it further refines the collection including also names of particular signals generation block, i.e. *Step* and *Ramp*. The choice of these two block is motivated by the fact that inside the editor's syntax (4.1.1) there is an explicit reference to these kind of signal. The exporter can be extended as will simply adding clauses for new block types, as interesting new candidates, signal generators available in Simulink are the *Pulse Generator* and the *Sinusoidal Wave*.

It's possible to notice that the functions write the file according to a CSV format. This choice is dependent on the way the requirements tool expects to import data dictionaries, and will be clarified in section 4.4.1.

4.2.2 STL operators library

In order to generate online monitors, the following restrictions has been introduced to the STL language.

1. The maximum level of nesting for temporal operators is two.
2. If there is a nested temporal operator, the condition on which the outer operator is evaluated must be a conjunction and at least one of the terms of the conjunction must be a proposition (not a temporal operator).
3. If T_b is the maximum value for all the endpoints of the intervals defined in the inner (nested) temporal operators, then the terms of the conjunction that are not temporal operators can only be true at time instants that are separated by a time interval always greater than T_b.

The STL operators library implements a Simulink version of the *And* plus the most common STL operators (Fig.4.2).

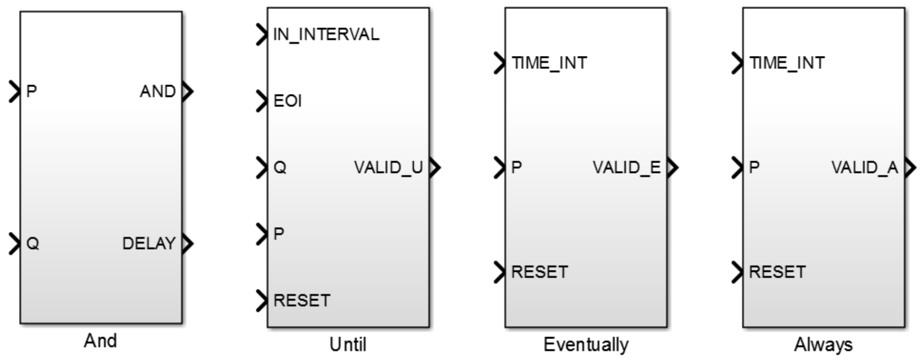
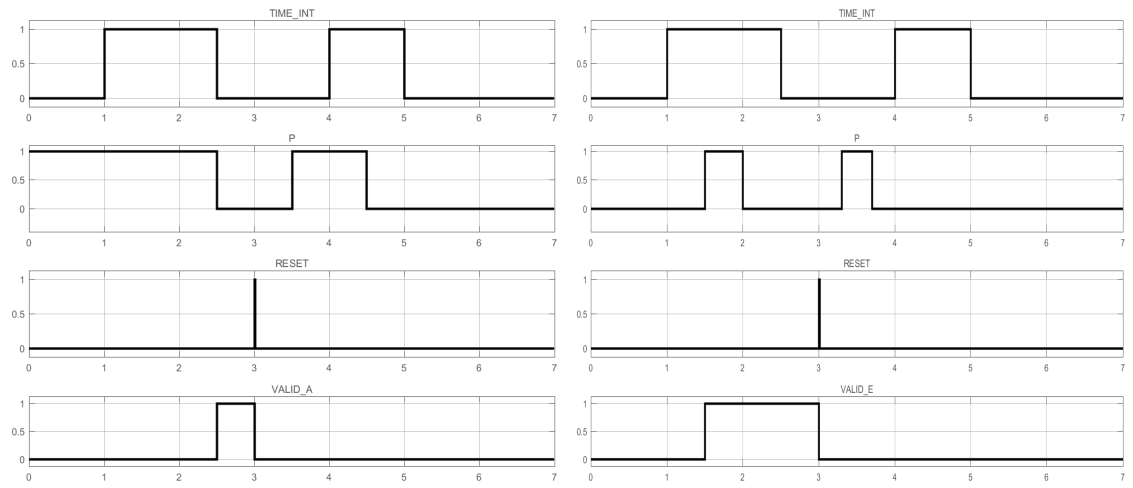


Fig. 4.2 STL Simulink

The *And* block is needed to implement nested temporal operators, in particular, it represents the conjunction among outer and inner propositions. Despite the name, it is quite different from the classical logical And since it does not returns **true** if *P* and *Q* are **true** at the same time, but rather if *Q* becomes **true** at some the time *t* after *P* became **true**. In order to verify *Q* it returns, through the *DELAY* port, the time in which *P* was verified, such time is needed to correctly compute the interval of the *Q*'s temporal operator.

All the operators blocks keep their output constant after a the property is detected. However, to facilitate their use in simulations concatenating several test cases, a reset input is also provided. Fig. 4.3 shows two test cases for *Always* and *Eventually*, one causing them return true and one false.



(a) Always (b) Eventually

Fig. 4.3 Test of STL Always-Eventually

In particular, it is possible to notice that the two operators have different behaviors due to their dynamics. The Always produces positive outputs only at the end of the time interval relative to the current test case (Fig.4.3a). Conversely, the Eventually operator returns positive feedback immediately after the occurrence of a positive input (Fig.4.3b).

Compared to its brothers, the timed *Until* operator has a more complex dynamics, in fact, in presence of multiple test case during the same simulation, it is not possible to determine a priori the instant in which it reacts to each input sequence. Fig. 4.4 present a possible multiple cases simulation for the timed Until.
The operator returns true if within a time interval the proposition P is true at least once and, for all the time before this happen, the proposition Q is true. The simulation provide four different time intervals. For the first, the operator always returns true since Q is always true until P becomes true. For the second interval, it immediately returns false since, after the reset, the proposition Q is false. After the second pulse of signal reset the operator returns again true, indeed this scenario is perfectly analogous to the first one. During the last time interval, there is yet another behavior. The operator is not immediately returning false, Q is true after the reset, but rather when Q becomes false since P has not yet became true. Note that another behavior, which has not been included for the purpose of not overload the graphics, is observable for the edge case in which Q is always true within a reset and the end

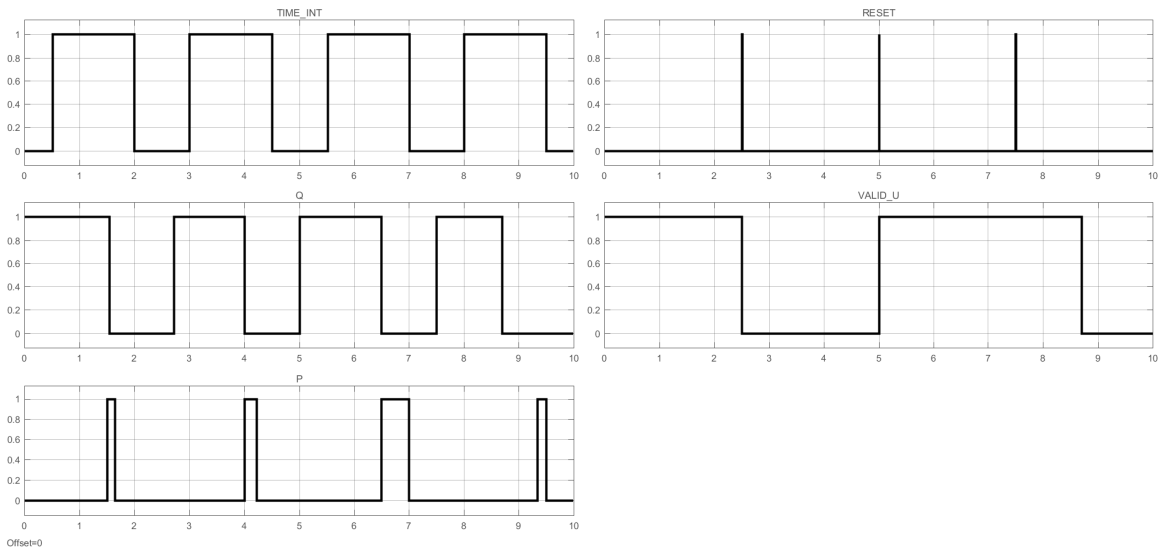


Fig. 4.4 Until Simulation

of a time interval, and *P* is always false in the same time. In this case the operator reacts with a negative output only at the end of the time interval, since, till the end , it "waits" for *P* becoming true.

The Simulink implementation of the STL timed Until is depicted in Fig. 4.5. The other operators are implemented in a similar, and simpler, shape. The complete library is available at [**?**].

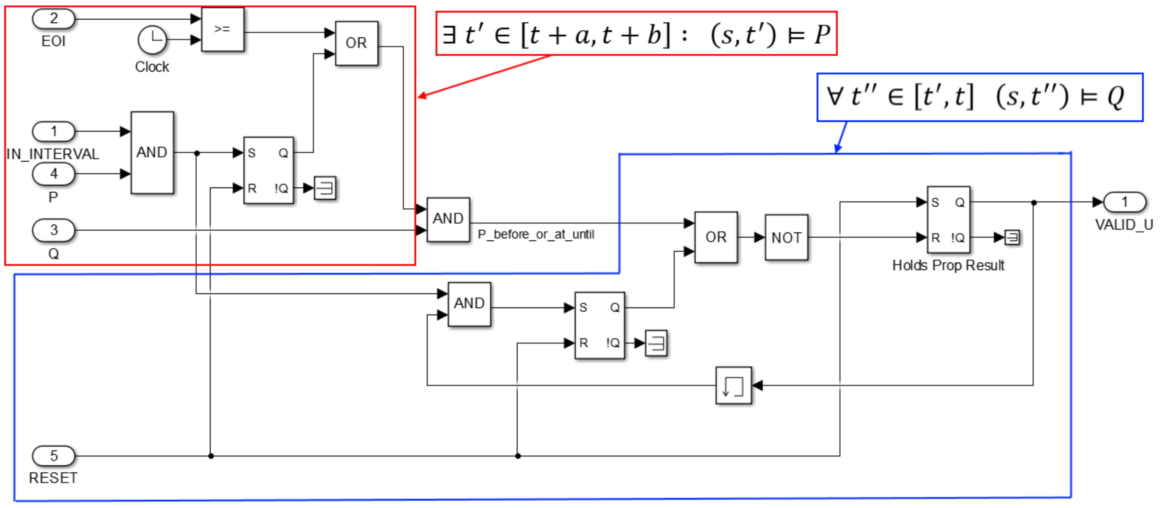


Fig. 4.5 Timed Until Implementation

4.2.3 System Performance Control Library

Section 3.2.2 provided an STL formalization of the most common control system performance requirements. In the same has been argued that such formalizations represent patterns, namely parameterizable properties having a fixed formal structure. Offering an implementation of those pattern is the main purpose of the *System Performance Control Library* (SPCL), it provides several atomic blocks able to verify the violation of performance requirements.The library block-set is presented in Fig. 4.6.

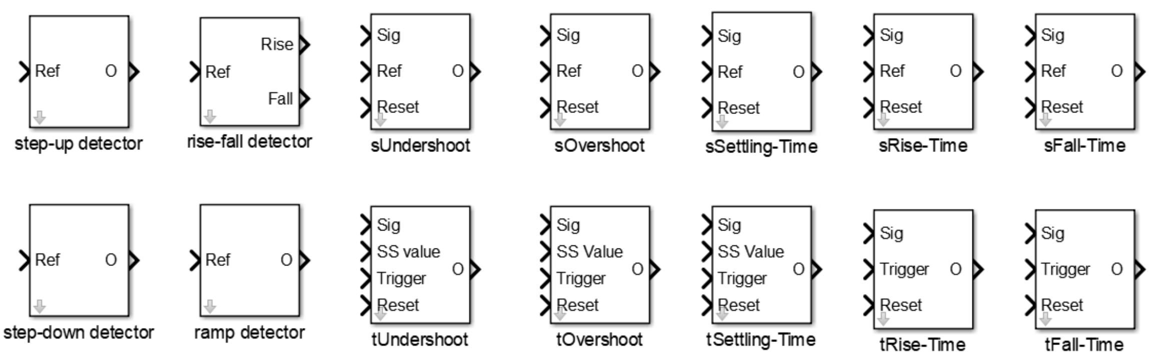
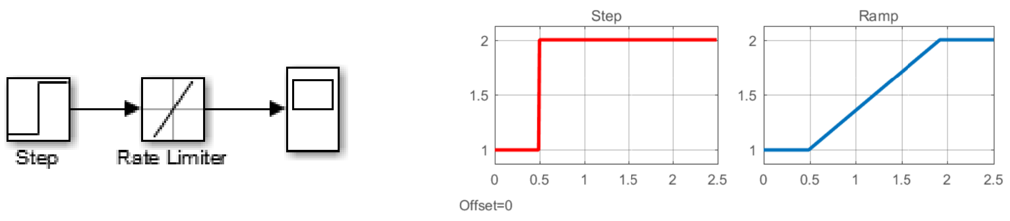


Fig. 4.6 System Performance Control Library

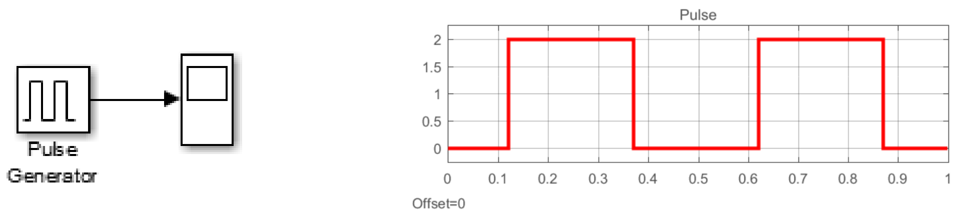
The entire block-set has been developed to operate in a simulation environment that uses a fixed-step solver for *ODE* integration. Possible future versions of the library will eliminate such limitation, allowing their use with a variable-step solver, by simply forcing the user to set a sample time for the block to discretize time and input signals.

A first categorization of all blocks could be performed by dividing them in *checkers* and *detectors*. Checker blocks, as the name suggest, are those in charge of effectively assert the violation of the property. In order to do so, they need to cooperate with the detectors blocks, which are pulse emitters based on the behavior of their input signals. The cooperation between detectors and checkers can be *implicit* or *explicit*, and allows to split the latter into two more subcategories depending on the type.

Blocks belonging to the first, whose name starts for "*s*", internally perform the classification of the input type by means of a step (up or down) detector. Therefore, those blocks direct implement the formulas of section 3.2.2. Ideally this subcategory is enough to verify performance requirements, since these properties are defined under the hypothesis of step as reference input. The reason why the library includes more general versions of this blocks, based on an explicit cooperation with an input detector, comes from the practical user experience. Indeed, many times, along multiple experiments, users may adopt reference signals sharing common features with steps, but having some substantial differences which makes them undetectable as steps. Two common examples are steps followed by a rate limiter (Fig. 4.7a) and pulse generators (Fig. 4.7b). The first is typically used to avoid sharp changes on input and basically transforms a step into a ramp, while the second is used to provide multiple inputs since it can be view as a sequence of up and down steps.



(a) Rate Limiter

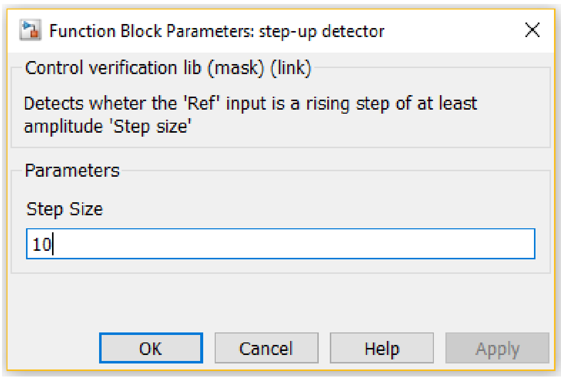


(b) Pulse Generator

Fig. 4.7 Input References

In order to support those kind of inputs, without forcing the user to modify existing model to comply with the library, the blocks family starting with "t" label can be used. Unlike the implicit blocks, in addition to the *Trigger* produced by a detector some of them may need more information that normally can be inferred by the reference signal, like, for instance, the steady-state value.

Signal Detectors As previously said, detectors blocks emit pulses every time the input signal shows an expected behavior. The parameter to evaluate the conditions are set from the user by means of the blocks' masks. In what follows will be focused just the step-up and the ramp detectors, step-down and rise-fall detectors are not considered since the first works reciprocally to the step-up while the second is just a parallel composition of a step-up and a step-down. From the blocks' masks (Fig.4.8) is possible to note that step-up detector, unlike ramp-detector which requires start and duration time, has not parameter which are functions of time, this allow to use it multiple times in the same simulation. A possible time behavior for both detectors is shown in Fig. 4.9, in the case of the step, due to the detectors definition, does not make difference if the input signal is a single step or a step train. Indeed, the detector implementation expects that it reacts every time there is a difference of at least


Function Block Parameters: step-up detector

Control verification lib (mask) (link)

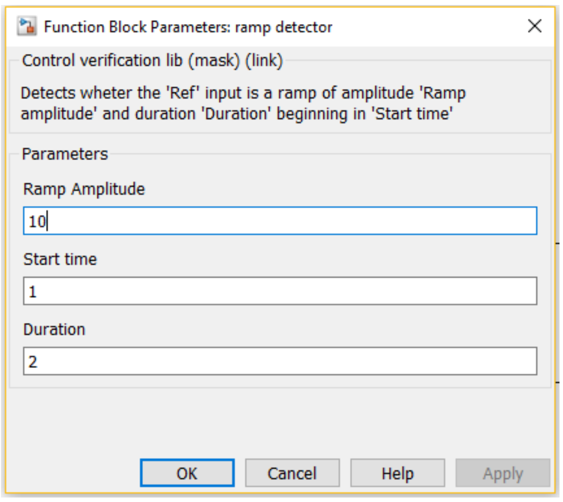
Detects wheter the 'Ref' input is a rising step of at least amplitude 'Step size'

Parameters

Step Size

10

(a) step-up detector’s Mask


Function Block Parameters: ramp detector

Control verification lib (mask) (link)

Detects wheter the 'Ref' input is a ramp of amplitude 'Ramp amplitude' and duration 'Duration' beginning in 'Start time'

Parameters

Ramp Amplitude

10

Start time

1

Duration

2

(b) ramp detector’s Mask

Fig. 4.8 Detector’s masks

Step size between two time-consecutive samples of the input signal. The implementation of

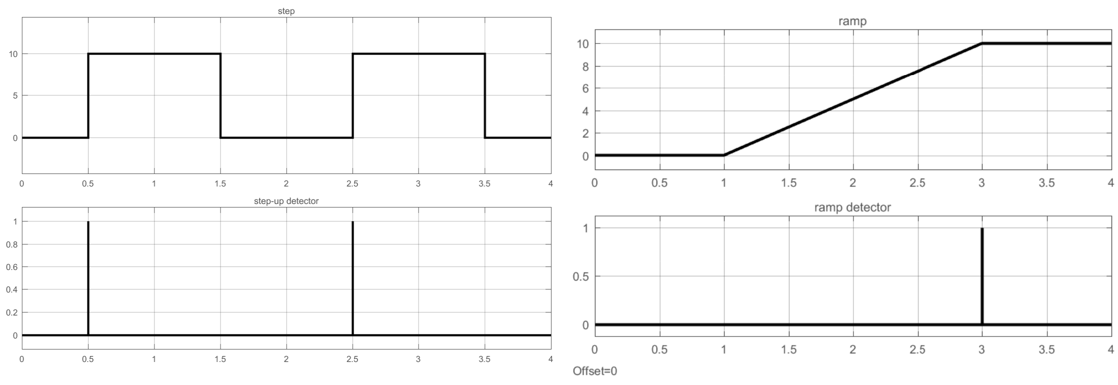


Fig. 4.9 Detectors’ time behavior

the ramp detector is more complex, it is based on the idea that the block must check if, for all t in $[Start, Start + Duration]$, the input signal is a line with a slope equal to $\frac{Amplitude}{Duration}$. This definition can be formally written with the following STL

$$\square_{[St,St+Dur]}\left\{\frac{dx}{dt} == \frac{Ampl}{Dur}\right\}$$

Such formula can be easily implemented with the Always operator of the STLib (4.2.2). Please note that in the real implementation (Fig.4.10) the equality of slopes is not checked through the equal operator, but rather ensuring that their difference is less than a very small quantity. This approach has been adopted to avoid possible numerical issues on determining the exact equality of two numbers.

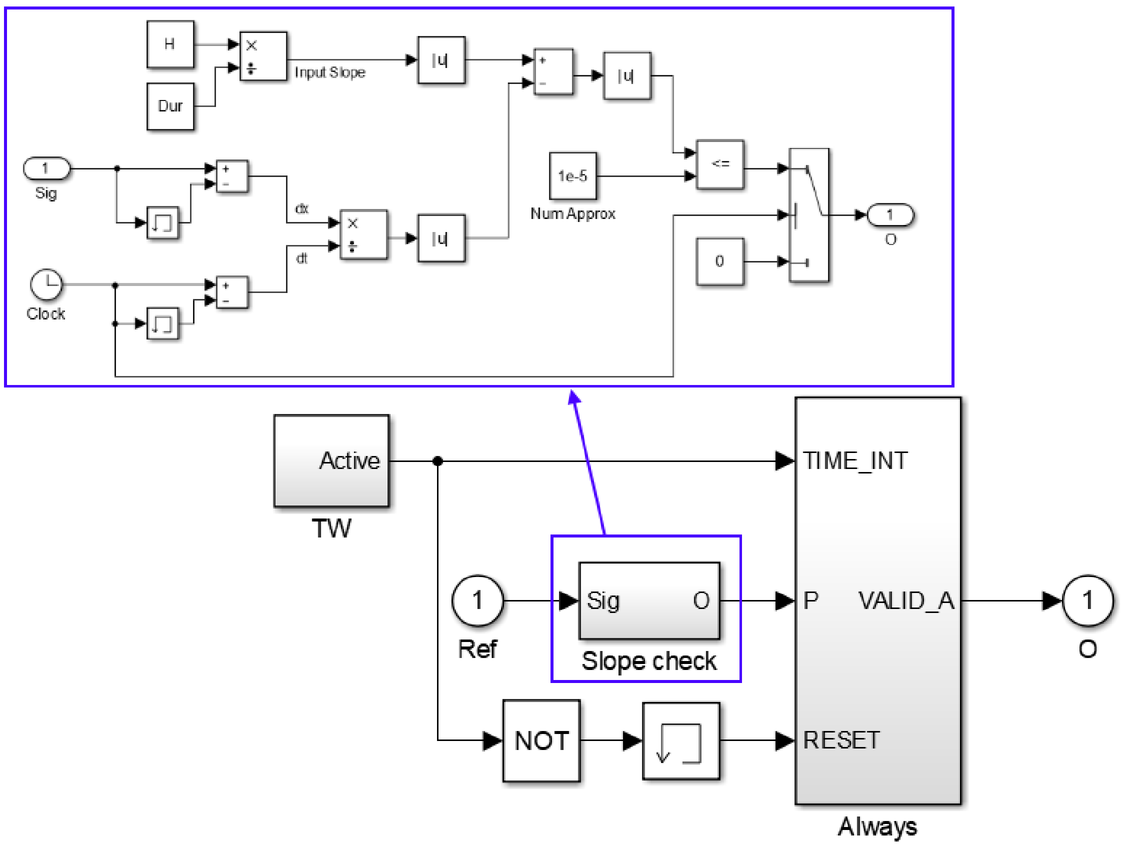
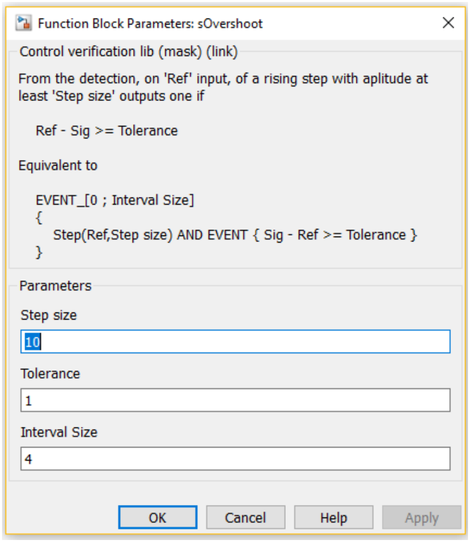


Fig. 4.10 Ramp Detector Implementation

Property Checkers For each performance requirements the library provides two different blocks depending on the type of cooperation with a signal detector. Time behavior and internal structure of blocks-couples relative to each property are very similar. Therefore, without losing generality only overshoot requirement will be analyzed as a case study.

The property's parameters, provided through the blocks' mask, are slightly different in the case of implicit (Fig.4.11a) or explicit (Fig.4.11b) reference detection. Indeed, if the blocks has to internally perform the step-up detection it needs to know the step size. Conversely, if the block only receives a detection trigger such parameter is not needed anymore. However, in order for the latter working properly, the information needed to estimate the overshoot has to be provided through a further input port, which has been labeled as "*SS Value*"(Fig.4.6). The common parameters of the two blocks are *Tolerance*, which represents the maximum tolerated distance among *Sig* and *Ref* (or *SS value*), and *Interval Size*, which corresponds to the simulation time.


Function Block Parameters: sOvershoot

Control verification lib (mask) (link)

From the detection, on 'Ref' input, of a rising step with aplitude at least 'Step size' outputs one if

```
Ref - Sig >= Tolerance
```

Equivalent to

```
EVENT_[0 ; Interval Size]
{
    Step(Ref,Step size) AND EVENT { Sig - Ref >= Tolerance }
}
```

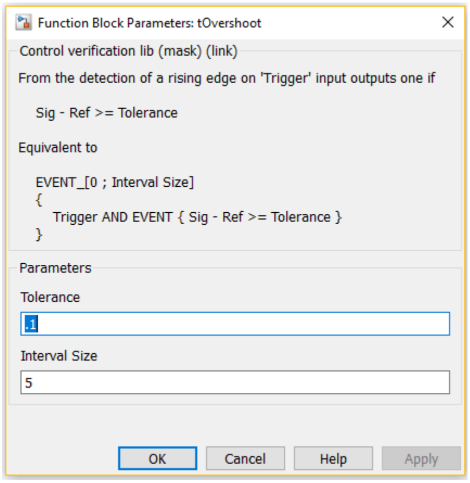
Parameters

Step size: 10

Tolerance: 1

Interval Size: 4

(a) *sOvershoot*


Function Block Parameters: tOvershoot

Control verification lib (mask) (link)

From the detection of a rising edge on 'Trigger' input outputs one if

```
Sig - Ref >= Tolerance
```

Equivalent to

```
EVENT_[0 ; Interval Size]
{
    Trigger AND EVENT { Sig - Ref >= Tolerance }
}
```

Parameters

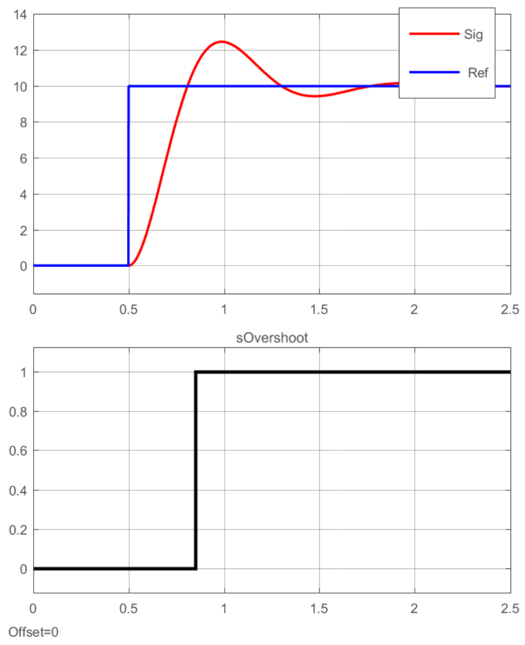
Tolerance: 1

Interval Size: 5

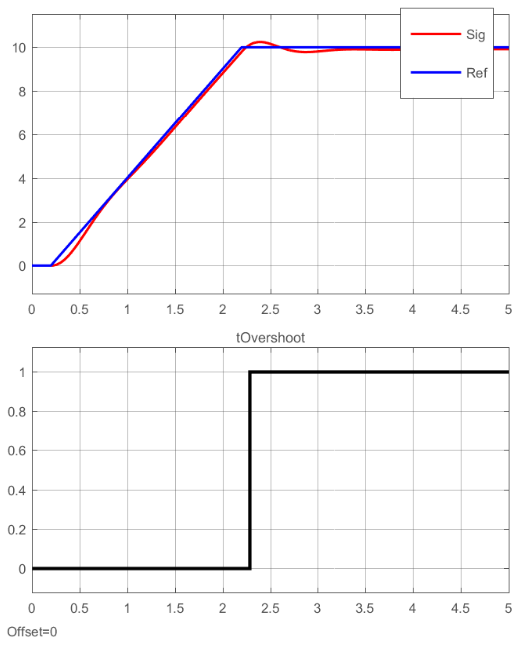
(b) *tOvershoot*

Fig. 4.11 Overshoot Blocks' Masks

Two possible simulations involving *sOvershoot* and *tOvershoot* are shown in Fig.4.12. For the first a step has been used as reference input, while for the second a ramp. Both blocks reacts immediately after the violation of the requirement.



(a) *sOvershoot*



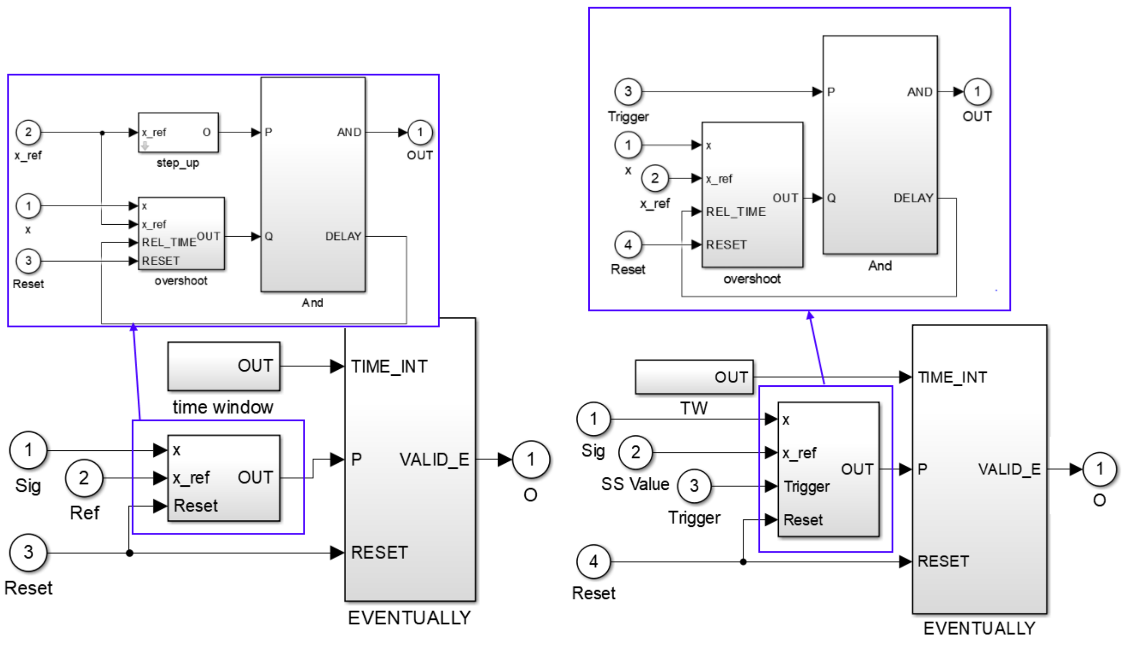
(b) *tOvershoot*

Fig. 4.12 Overshoots' Time response

The blocks have a layered structure which recalls the level of nesting of the parent STL formula. Essentially, the Simulink implementation of the two blocks only differ for the signal used as first input of the *And* operator inside the blocks' first layer (Fig.4.13). Such signal correspond to the proposition used at the left of the internal conjunction in nested STL temporal operators. Therefore, the STL formula, implemented in the *tOvershoot* block, can be rewritten as

$$\Diamond_{[0,T]}\{Trigger \wedge \Diamond\{Sig - SSValue \geq Tolerance\}\}$$

In order to avoid redundancy, the content of *overshoot* subsystem of Fig.4.13 has not been reported. Indeed, for both versions of the blocks, its structure is analogous to the first layer, i.e. an STL eventually with a less than or equal to proposition as input.



(a) *sOvershoot* (b) *tOvershoot*

Fig. 4.13 Overshoots' Implementations

All other property checkers follow the same philosophy of overshoot blocks. Although with them a good coverage of the performance requirements has been achieved, as future extension other properties may be included, making the *SPCLibrary* a complete tool-chain for the analysis, during simulation, of several control systems models.

4.3 Design Patterns

In object-oriented software design a pattern describes a problem which occurs over and over in the environment, and then describes the core of the solution to that problem, in such a way that you can use this solution multiple times [**?**]. Design patterns make it easier to reuse successful designs and software architectures, and helps avoiding design alternatives that compromise reusability.
Design patterns solve many problems that object-oriented designers face every day, like

Finding appropriate abjects The hard part about object-oriented design is decomposing a system into objects. Design patterns help identifying abstractions and the objects that can capture them.

Determining object granularity Objects can vary tremendously in size and number.Design patterns address this issue as well. Many of them describe specific techniques of decomposing an object into smaller objects.

Specifying object interfaces Interfaces are fundamental in object-oriented software systems. There is no way to know anything about an object without going through its interface. Design patterns help defining interfaces by identifying their key elements and the kinds of data that get sent across an interface.

An exhaustive treatment of all known design patterns requires an huge effort, and leads to cover situation that are out of the scope of this work. Hence, herein only two of them, those really involved, will be taken into consideration. Being scalability and extensibility key factors of the proposed framework, particular care must be payed on the software architecture. The use of patterns belonging to *creational* and *behavioral* classes greatly helps to accomplish this objective.

4.3.1 Factory

Creational patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object. Creaional patterns become important as system evolve to depend more on object composition than class inheritance. As that happens,emphasis shifts away from hard-coding a fixed set of behavior toward defining a smaller set of fundamental behavior that can be composed into any number of more complex ones. Thus creating objects with particular behavior requires more than simply instantiating a class.

The intent of the *Factory Method* is defining an interface to create objects, but let subclasses decide which class to instantiate. The UML representation of the Factory pattern is shown in Fig.

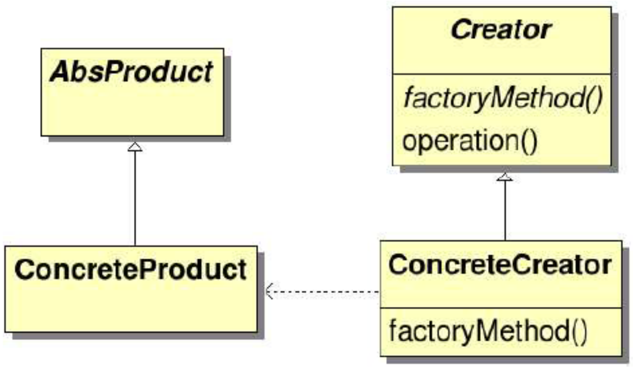


Fig. 4.14 Factory

Product defines the interface of the objects the factory method creates

Concrete Product implements the Product's interface

Creator declares the factory method

Concrete Creator overrides the factory method to return an instance of a ConcreteProduct

Sometimes class Factories are implemented as *Singleton*, which is another design pattern ensuring that in the system there is only one instance of the class. The client class request for the creation of new a new concrete product to the Factory and uses object through their interface. Typically the client provide to the Factory a products identifier, this is then used, for instance, inside a switch/case to determine the correct concrete product. However, managing all possible Concrete Products by hard-coding the logic to create them is not flexible. For this reason, more sophisticated techniques are usually adopted, they make use of the so called *registry* and let Concrete Products register themselves into the factory. Follows an example of the Factory pattern, it make use of *Reflections*, which is a Java mechanisms to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts.

```
interface Product {
  String getName();
}
class ProductOne implements Product {
  static {
    ProductFactory.instance().register("One", ProductOne.class);
  }
  public String getName() { return "instance of ProductOne"; }
}
class ProductTwo implements Product {
  static {
    ProductFactory.instance().register("Two", ProductTwo.class);
  }
  public String getName() { return "instance of ProductTwo"; }
}

class ProductFactory {
  // Singleton
  static private ProductFactory inst = new ProductFactory();
  static public ProductFactory instance() { return inst; }
  // The registry
  private HashMap registry = new HashMap();
  public void register(String productID, Class productClass) {
    registry.put(productID, productClass);
  }
  public Product create(String ID) {
    Class pClass = (Class)registry.get(ID);
    if (pClass == null) {
      System.err.println("Product " + ID + " not registered");
      return null;
    }
    try {
      Constructor pConstructor = pClass.getDeclaredConstructor(null);
      return (Product)pConstructor.newInstance(null);
    }
        ...
}
...
```

```java
public static void main(String[] args)
{
  Class.forName("ProductOne");
  Class.forName("ProductTwo");
    ...
}
```

The Factory uses an associative set in order to couple Concrete Products with their identifier. Each Concrete Product register itself through the initial static block, however, in order to be sure of having all the registration done before start creating Concrete Product, the loader method "*Class.forName(ClassName)*" has to be called.

4.3.2 Visitor

4.4 Eclipse Editor

4.4.1 Data Dictionary Importer

4.4.2 Context Aware Syntax Helper

4.4.3 Abstract Syntax Tree Generation

4.5 Model population

References

[] Acceleo. http://www.eclipse.org/acceleo/.

[] Rajeev Alur, Tomás Feder, and Thomas A Henzinger. The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, 43(1):116–146, 1996.

[] Alessio balsini repo. https://github.com/balsini/SignalTemplateLibraryAutogen/.

[] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *International Symposium on Formal Methods for Components and Objects*, pages 200–225. Springer, 2007.

[] Alessio Balsini, Marco Di Natale, Marco Celia, and Vassilios Tsachouridis. Generation of simulink monitors for control applications from formal requirements. 2017.

[] Luca Benvenuti, Alberto Ferrari, Leonardo Mangeruca, Emanuele Mazzi, Roberto Passerone, and Christos Sofronis. A contract-based formalism for the specification of heterogeneous systems. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 142–147. IEEE, 2008.

[] IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board. Ieee recommended practice for software requirements specifications. Institute of Electrical and Electronics Engineers, 1998.

[] Edmund Clarke and E Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *25 Years of Model Checking*, pages 196–215, 2008.

[] Matthew B Dwyer, George S Avrunin, and James C Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. ACM, 1998.

[] Cindy Eisner and Dana Fisman. *A practical introduction to PSL.* Springer Science & Business Media, 2007.

[] IEEE-Commission et al. Ieee standard for property specification language (psl). Technical report, Technical report, IEEE, 2005. IEEE Std 1850-2005, 2005.

[] Rumbaugh James, Jacobson Ivar, Booch Grady, et al. The unified modeling language reference manual. *Reading: Addison Wesley*, 1999.

[] James Kapinski, Xiaoqing Jin, Jyotirmoy Deshmukh, Alexandre Donze, Tomoya Yamaguchi, Hisahiro Ito, Tomoyuki Kaga, Shunsuke Kobuna, and Sanjit Seshia. St-lib: A library for specifying and classifying model behaviors. Technical report, SAE Technical Paper, 2016.

[] Ni labview. http://www.ni.com/labview/.

[] Leonardo Mangeruca, Orlando Ferrante, and Alberto Ferrari. Formalization and completeness of evolving requirements using contracts. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 120–129. IEEE, 2013.

[] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[] Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of computer science*, pages 475–505. Springer, 2008.

[] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.

[] United States. Department of Defense. *Software Technology Plan: Volume II Plan of Action*. Department of Defense, 1991.

[] United States. Department of Defense. *MIL-STD-498: Software Development and Documentation*. Department of Defense, 1994.

[] Amir Pnueli and Zohar Manna. The temporal logic of reactive and concurrent systems. *Springer*, 16:12, 1992.

[] Simulink. http://www.mathworks.com/products/simulink/.

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

[] Sysml. http://www.omgsysml.org/.

[] Uppaal. http://www.uppaal.org/.

[] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.