Synchronous Programs Testing Language (SPTL)

Tka Mouna, Christophe Deleuze, Ioannis Parissis

Synchronous Programs Testing Language (SPTL)

Mouna Tka Mnad, Christophe Deleuze, and Ioannis Parissis

Univ. Grenoble Alpes, LCIS, Valence, France
mouna.tka@lcis.grenoble-inp.fr

Abstract. SPTL is a language designed to test applications developed for synchronous controllers. It makes possible to provide a specification of the software external environment. This specification can then be processed to generate test input sequences guided by directives such as profiles of use and scenarios. We introduce a definition and an overview of the language through a simple example of a reactive system that we present in this paper.

1 Introduction

A "reactive system" [4] is a system that continuously interacts with its environment. Synchronous systems [1] is a class of reactive systems that are based on the synchrony hypothesis: they are supposed to be fast enough to take into account any external event. They are widely used in industry for the automation of various processes.

Several programming languages have been defined in the past for developing synchronous systems such as Esterel, Signal or Lustre [3]. In this research work, we are especially interested in a specific class of synchronous controllers, produced by a leading manufacturer of automation control components. These controllers are used in applications such as heating, air conditioning, access control, water and air treatment, waste treatment or pump management. Unlike usual synchronous systems, these devices are intended to be programmed by a large scope of users, including non-experts. A user-friendly graphical programming environment is used to build and integrate software applications as data-flow networks made of already developed components.

Even if the applications defined by such non-expert users are not highly critical, it is however necessary to adequately test them. The objective of this work is to design an efficient and easy to use environment for testing programs developed with this environment. Users must be able, as they write programs, to express various test scenarios that will be used for testing purposes. These scenarios must be compiled into test data generators that will be used to simulate and test the devices.

In this paper, we present the first component of such a testing environment, an adequate and easy to use test description language.

This document is structured in 6 sections. In the following section, we present related work and the motivations for our research while in the third section we

define the main concepts of the testing language. A overview of the language through examples is provided in section 4. In section 5, we describe the semantics of the language.

2 Related Work

Automatically testing synchronous systems has been addressed in the past in several research works. We focus on investigations on testing Lustre programs [3], because of the similarity between this synchronous data-flow language and the programming environment of our synchronous controllers.

– **Lutess** [9] has been designed as a black-box testing environment that automatically generates test input sequences from a formal specification of the software external behavior provided in a language similar to Lustre.

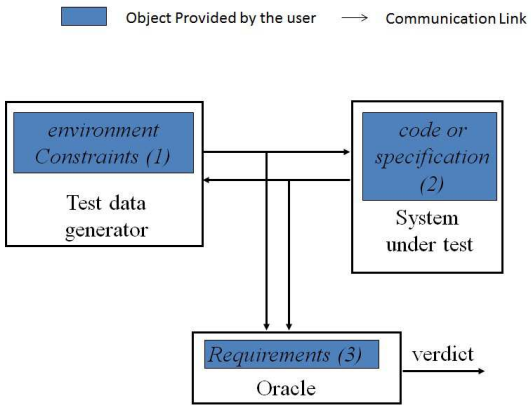

Fig. 1. The Lutess environment

Lutess requires three components: the software environment description (1), the executable code of the system under test (2), and a test oracle (3) describing the system requirements (see Figure 1). The system under test and the oracle are both synchronous executable programs. The environment description is composed of a set of constraints, provided in a Lustre-like language, that the environment of the software application is assumed to satisfy. Lutess translates the above constraints into a random test generator operating on a single action-reaction cycle. The generator randomly selects and sends a valid input vector to the program under test which reacts with an output vector and feeds back the generator with it. The generator proceeds by

producing a new input vector and the cycle is repeated. It must be noted that Lutess constraint resolution mechanism is implemented using constraint logic programming (CLP). A case study using Lutess to test a steam-boiler controller is described in [8].

- **Lurette** [5] is an automatic test generator for Lustre programs the main concepts of which are similar to Lutess: a specification of the environment behavior is provided and translated into a test generator. The specification language is ad hoc and the constraint resolution mechanism is based on convex polyhedra.
- **GATeL** [6] has been developed at the French Nuclear Research Agency (CEA). Its approach is quite different from the two above described Lustre related tools. Indeed, GATeL uses the Lustre program itself and a test objective to generate an appropriate test sequence. In other words, GATeL is a "white-box" testing tool as opposed to Lutess and Lurette. Similarly to Lutess, GATeL employs CLP: both the program and the test objective are translated into constraints that are solved to get the effective test sequence.

The above approaches are specifically designed for synchronous programming. We can mention TTCN-3 (Testing and Test Control Notation 3),[1] a well-known test specification and test implementation language standardized by the European Telecommunications Standards Institute (ETSI) and the International Telecommunication Union (ITU).

TTCN-3 as a modeling language can support test generation tools by means of some annotations (see for example [10]). It applies to a variety of application domains and types of testing. TTCN-3 is a popular test definition language mostly used by experts for communications protocols testing.

The above tools and approaches are either research prototypes or industry tools intended to be used by domain and test experts. Lutess and Lurette use rather complex test modeling languages making the specification of test scenarios difficult while GATeL requires the user to determine a test objective and to assist the tool during the constraint resolution. Similarly, TTCN-3 seems too complex for non-expert users and is not designed for synchronous applications. However, we use the main concepts of these tools to define an easy-to-learn test modeling language, described in the next sections.

3 Basic Concepts

To make the new testing language easy to use, we define four simple concepts, defined in the next subsections: *profiles*, *categories and groups*, and *scenarios*.

3.1 Profile

A system performance and functioning are significantly dependent on the environment in which it operates. A "profile" represents a context in which the

[1] `http://www.ttcn-3.org/`

system is used. ZigBee [11], for instance, uses profiles defined as message formats and processing actions to allow devices to exchange data in a given application domain. Profiles are developed by companies to help providing specific needs. For example: commercial building automation, telecom applications, hospital care ... More generally, the notion of "operational profile" has been introduced by Musa [7] and defined as a quantitative characterization of how the system will be used. Let us see an example of program to illustrate the notion of profile and its utility in our case: an air conditioner. It will not be used in Tunisia in a small house in winter the same way it will be used in Canada in a big building during the summer, because temperatures and time slot of use will be very different. So, to every profile correspond constraints of use that will necessarily affect testing.

3.2 Categories and Groups

Profiles are composed of *categories* of constraints provided by the tester. As shown in Figure 2, each category contains *groups* of constraints. The constraints in each group define limits related to a particular environment or users.

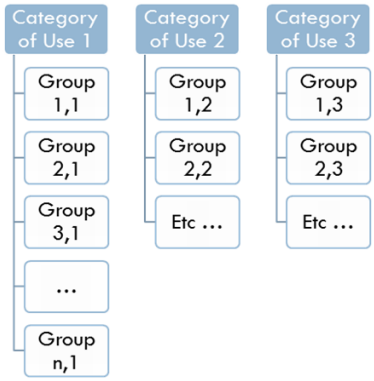

Fig. 2. Categories

A profile is obtained by combining groups from different categories. For example in Figure 3 profile 1 is composed of Group1,1, Group3,2 and Group2,3: Profile1 = Group1,1 ∪ Group3,2 ∪ Group 2,3.

3.3 Scenarios

When testing reactive systems, we need to express how they are constrained by the context and the user activity. A direct approach is to explicitly provide typical and significant user activities in the testing process. Such descriptions, often called "scenarios", support reasoning about situations of use.

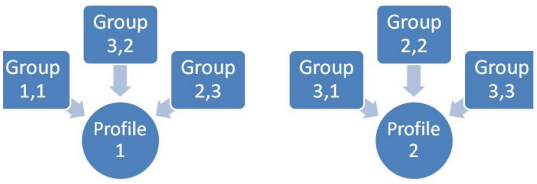

Fig. 3. Profiles

A scenario can be seen as a story [2]. For example, at the beginning all the buttons of a lighting system are not pressed then the user presses rapidly button1, then presses button2, then keeps pressing button1 until the permanent lighting mode is on, etc.

4 Overview of SPTL

In this section we present the syntax of SPTL using an example of a reactive system.

4.1 Example: Air Conditioner

Let's consider the following simple reactive system, an air-conditioner controller, shown in Figure 4.

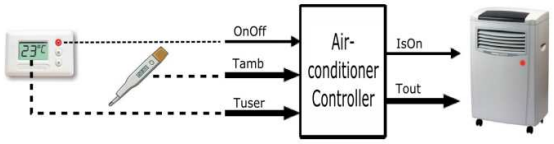

Fig. 4. Air conditioner

The program inputs are:

- OnOff (Boolean): true when the user pushes the On-Off button of the air-conditioner;
- Tamb (integer): value in Celsius degrees of the ambient temperature;
- Tuser (integer): value in Celsius degrees of the user selected temperature.

The program outputs are:

- IsOn (Boolean): indicates the state of the air-conditioner;
- Tout (integer): indicates the temperature of the air that the air-conditioner blows.

4.2 Variables

The first part of the test model is a set of variable declarations: the system inputs and outputs are declared as input and output variables. A variable declaration associates a type with an identifier and optionally an initial value. Each identifier is unique. The basic types that are considered are boolean, integer, and time. The word "random" means that the value of the variable will be randomly generated, if there is no other specification about it in the program.

```
var
input bool OnOff;
input random int Tamb = 10;
input int Tuser = 7;
output bool IsOn;
output int Tout;
```

4.3 Categories Specification

Each *category* contains a set of *groups* of constraints to be respected and possibly sub-programs used for the calculation of values of certain variables. Thereafter, a set of *profiles* can be generated combining groups of different categories. The tester can then choose a profile consisting of multiple constraints (environment, use ...), modify it or add test cases.

```
categ CountrySeason
{  group FranceSummer
    { 20<Tamb ; Tamb<44 ;
      Tuser = ComputeTemperature(Tamb) }

   group TunisiaWinter
    { 6 <= Tamb ; Tamb <= 14 }

sp ComputeTemperature(int Tamb) returns(int Tuser)
    { Tuser = pre(Tamb) - 3 }
}
```

A subprogram, like ComputeTemperature in the example, connects an input with a set of outputs together and admits intermediate variables. The declaration of a subprogram contains a header, an optional block for variables declaration and the body of the subprogram that contains the equations.

```
categ TypePlace
{
   group Company { OnOff=prob(true, 0.001) }
   group House   { OnOff=prob(true, 0.1) }
}
```

A constraint involves one or more variables, and restricts values that these variables can take simultaneously. These relationships are given by the environment of the system and the user profile. Expressions are defined by combining sub-expressions using arithmetic, logical and temporal operators as in `Tuser = pre(Tamb) - 3`. `pre` is a temporal operator used to get the value of the variable in the previous cycle. For this to be meaningful `Tamb` must have a value before the first cycle. Such an initial value is to be specified in the variable declaration. Every output variable must appear within the `pre` operator since in the calculation of the test values we consider the system outputs generated in the previous cycle. The `prob` operator is used to define probabilities, useful to guide test data selection. The expression `prob(`v`,`p`)` has the value v with probability p, otherwise it takes a random value different from v.

4.4 Scenarios Specification

A scenario is composed of point-wise steps and interval steps. A point-wise step, enclosed between "{" and "}", is a set of constraints to obey in one test generation time. It can be used either in the first step for initializing environment variables or in any step of the execution. An interval step, enclosed between "[" and "]", is a set of constraints on the test values to generate that remains valid until a condition holds. The condition is a boolean expression enclosed between "(" and ")". Below we describe two example scenarios.

Scenario 1

```
6 ≤ Tamb ;
Tamb  ≤ 14 ;
OnOff = prob(true, 0.1 ) ;
begin
  { Tuser = 8 }|
  [ Tuser = pre(Tuser) +1 ( Tuser = 10 ) ]|
  { Tuser = 12 }|
  [ Tuser = pre(Tuser) + 2  ( Tuser = 22 ) ]
end
```

This first scenario means that Tuser is initialized to 8. Then this temperature will be increased until reaching 10 °C. After this Tuser is set to 12 °C . Finally, for the rest of the test data that will be generated Tuser will be increased by 2 at each cycle until it reaches 22 °C.

Table 1 shows a trace that could be generated from scenario 1. This trace can be examined to check for expected properties from the air-conditioner such as:

- $\text{IsOn}_t \wedge (\text{Tamb}_t < \text{Tuser}_t) \Rightarrow \text{Tout}_t > \text{Tamb}_t$: when it's cold, the air conditioner heats and conversely.
- $\text{OnOff}_t \Rightarrow \text{IsOn}_{t-1} \neq \text{IsOn}_t$: the OnOff button turns on / off the air conditioner.

Table 1. An extract of execution traces of scenario 1

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
OnOff	0	1	0	0	0	0	0	0	0
Tamb	13	14	13	12	11	10	10	11	9
Tuser	8	9	10	12	14	16	18	20	22
IsOn	0	1	1	1	1	1	1	1	1
Tout	7	8	9	12	15	18	21	23	27

Scenario 2:

```
var
time t ;
begin
 { Tamb=10; Tuser=2; t.start } |
 [ Tuser=pre(Tuser)+3; Tamb=10 (t = 3c) ] |
 [ Tuser=pre(Tout)-1; Tamb=pre(Tamb)-1 (Tamb=6) ] |
 { Tuser=10; Tamb=6 } |
end
```

This second scenario shows the use of a variable (`t`) of type `time`. Time values are positive numbers followed by a unit identifier (`s` for seconds, `ms` for milliseconds, `c` for cycles). (A cycle in the tested program is the necessary time for a full run from starting entries, calculating outputs' values, to finally generating them). `t.start` is a pseudo constraint having the side effect to start counting time in `t`. In the example `t` is used in the condition of the interval step. Table 2 represents a possible trace for scenario 2.

5 Semantics of the Language

In this section, we give a description of the language semantics. In order to precisely describe how test sequences are generated, we start by defining the semantics of three basic SPTL constructs: expressions, constraints and scenarios.

5.1 Expressions

The value of an expression depends on the values of the variables it contains. We call *environment* and note σ a mapping from variables to their value. Thus $\sigma(x)$ is the value of variable x in environment σ.

We note $[\![e]\!]$ the semantic function of expression e. The (denotational) semantics of most expressions is rather straightforward as shown in Figure 5: the

Table 2. An extract of execution traces of scenario 2

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
OnOff	1	1	0	0	0	0	0	0	0
Tamb	10	10	10	10	9	8	7	6	6
Tuser	2	5	8	11	11	11	11	12	10
IsOn	0	1	1	1	1	1	1	1	1
Tout	1	4	8	12	12	12	13	14	12

semantics of a constant value is this very value, whatever the environment; the semantics of a variable expression is the variable value in the considered environment; arithmetic and boolean operators map to the obvious mathematical operators.

$$
\begin{aligned}
[\![cst]\!](\sigma) &= cst \\
[\![id]\!](\sigma) &= \sigma(id) \\
[\![e_1 + e_2]\!](\sigma) &= [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma) \\
[\![e_1 \texttt{ and } e_2]\!](\sigma) &= [\![e_1]\!](\sigma) \wedge [\![e_2]\!](\sigma) \\
&\ldots \\
[\![\texttt{prob}(v,p)]\!](\sigma) &= \begin{cases} v & \text{with probability } p \\ w & \text{with probability } 1-p, \quad w \neq v \end{cases} \\
[\![\texttt{pre}(e)]\!](\sigma) &= [\![e]\!](prev(\sigma)) \\
&\text{where } prev(\sigma) = \lambda \mathtt{v}. \begin{cases} \sigma(\mathtt{v}_m) & \text{if } \mathtt{v}_m \text{ exists} \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 5. Semantics of expressions

The probabilistic operator introduces non determinism: the expression `prob(`v`,`p`)` has the value v with probability p, or a value w randomly chosen among the other possible values in the type.

The temporal operator `pre` allows to refer to the value of a variable (or more generally an expression) at the previous cycle. To define it we extend the environment to include *memory variables*: such a variable $\mathtt{v}_m$ always has the value variable `v` had at the previous cycle. As noted in section 4, initial values must be provided for variables appearing in a `pre` so that memory variables can be initialized. We show in section 5.4 how these variables are updated.

The set of memory variables is finite and can be built from a static analysis of the model. Let's start with an empty set ϕ_m. At each occurrence of the `pre` operator in the model, we consider its expression argument `e` and add $\mathtt{v}_m$ to

ϕ_m for each variable `v` occurring in `e`. Nested `pre` expressions can also easily be dealt with by adding a "second order" memory variable (*e.g.* $\mathtt{v}_{mm}$ for variable $\mathtt{v}_m$.)

We define *prev* as the function mapping an environment σ_1 to an environment σ_2 where each variable `v` has the value of the corresponding memory variable $\mathtt{v}_m$ in σ_1 if it exists. We can now state that the semantics of `pre(`e`)` in environment σ is the semantics of e in environment $prev(\sigma)$.

5.2 Constraints

When considering constraints we need to distinguish the several kinds of variables that exist in an SPTL model:

- *output* variables are set by the system under test,
- *input* variables are to be computed by the SPTL model,
- *memory* variables remember the value of variables in the previous cycle.

Since output variables can only appear in a `pre` operator, a constraint only contains occurrences of input and memory variables.[2] It has the form of a boolean expression but its semantics is very different: a constraint defines the possible values of its input variables, according to the values of its memory variables.

Let's split the environment σ into the environment of memory variables σ_m and the environment of input variables σ_i. We note $\sigma = \sigma_m \oplus \sigma_i$. It is clear that a constraint defines the possible values of σ_i from a given value of σ_m.

$$\begin{aligned} \langle\!\langle c \rangle\!\rangle(\sigma_m) &= \{\sigma_i, [\![c]\!](\sigma_m \oplus \sigma_i) = true\} \\ \langle\!\langle c_1; ...; c_n \rangle\!\rangle(\sigma_m) &= \langle\!\langle c_1 \rangle\!\rangle(\sigma_m) \cap ... \cap \langle\!\langle c_n \rangle\!\rangle(\sigma_m) \end{aligned}$$

Fig. 6. Semantics of constraints

Noting $\langle\!\langle c \rangle\!\rangle$ the semantic function of constraint c, Figure 6 shows these possible σ_i are those that, when associated with σ_m, form a complete environment σ in which the boolean expression has the value *true*. Also shown on the figure is the rule for constraint composition: the set of possible values for the input variables is the intersection of the sets of possibles values for each constraint.

5.3 Scenarios

Recall that a scenario is a sequence of

- *point-wise steps*: they specify constraints that apply at one cycle;
- *interval steps*: they specify constraints that apply until some *exit condition* is verified.

[2] Of course, memory variables do not appear explicitly in the model. We call memory variable any occurrence of a variable in a `pre` expression.

We can describe a scenario by an automaton where each scenario step is a state. A point-wise step has a single *true*-labeled transition to the next state, while an interval step has a transition to the next state, labeled by its exit condition, and a transition to itself, labeled by the negation of its exit condition.

The scenario 1 in section 4.4 would thus be described by the automaton of Figure 7 where states 1 to 4 hold the sets of constraints of the four scenario steps.

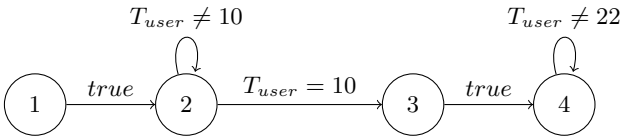

Fig. 7. Automaton for scenario 1

Scenario 2 shows the use of *time variables* (variables of type `time`) and *time values*. Although the language provides three units for time values, time variables only count in `c` (cycles). Time values expressed in other units (`s` and `ms` for second and millisecond) are first converted to `c` unit by dividing them by the cycle time. *ct* being the cycle time in seconds, we have:

$$\begin{aligned} e \text{ s} &\rightarrow \lfloor e/ct \rfloor \text{ c} \\ e \text{ ms} &\rightarrow \lfloor e \times 1000/ct \rfloor \text{ c} \end{aligned}$$

Time variables are discussed in the next section.

5.4 Programs

Recall that a set of profiles have been generated as combinations of groups from the categories. To generate a test sequence we need to consider:

- variable declarations,
- a profile,
- a scenario.

Let's call this triple an SPTL *program* (as opposed to the whole *model*) since that's what will be used for generating a test sequence. Figure 8 shows an informal operational description of the semantics of such a program. The basic idea is that at each cycle a current set of constraints *cset* is in force (step 1): it is the combination of the profile constraints and the set of constraints of the currently active step in the scenario (`scen_step`).

Solving *cset* in environment σ_m, possible values for the input variables are computed. One of these values σ_i is randomly selected (2) and fed to the inputs of the system under test (3).

System outputs are read (4), an oracle can optionally be given current inputs and outputs (4b), scenario active step is changed (5) if its exit condition is *true* (evaluated in the environment $\sigma = \sigma_m \oplus \sigma_i$).[3]

Memory variables are updated (6). The test terminates if scenario ends, else next cycle begins (7).

(0) `scen_step` $\leftarrow$ `1`
initialize σ_m

1. `cset` $\leftarrow$ profile $\cup$ `scenario[scen_step].constraints`
2. compute σ_i by solving `cset` (using σ_m)
3. apply σ_i to system inputs
4. read system outputs to σ_o

(4b) feed oracle with σ_i and σ_o

5. evaluate exit condition from scenario step, increment `scen_step` if *true*
6. update σ_m (from σ_i and σ_o)
7. if end of scenario then stop else goto (1)

Fig. 8. Execution cycle of an SPTL program

Initially, `scen_step` is set to 1, the first step of the scenario. Initial values for the memory variables are provided with variable declarations (0).

For clarity, we haven't included the handling of time variables so far. A time variable counts a number of executed cycles, thus it holds an integer value initialized at 0. The program state includes an initially false internal boolean variable t_{active} for each such time variable. Solving the pseudo constraint `t.start` has the effect of setting t_{active} to true. At end of cycle all time variables whose associated t_{active} boolean variable is true are incremented. Time variables can not appear in a `pre`.

6 Conclusion and Future Work

We have proposed a language to write models to generate test sequences for synchronous reactive systems. It is based on describing both the constraints on the system environment and the profile of the users of the program under test. One of the main design objectives is to allow non testing experts to make realistic tests. This has led us to design a language with few simple concepts.

Moreover, the notion of categories, groups and profiles will reduce the work needed to prepare realistic tests. Indeed, some usual profiles may be predefined, by the manufacturer's engineers or by a third party, to further assist the user in its testing design. Test scenarios make possible to take test objectives into account during test generation.

[3] A point-wise step can be defined as an interval step with a *true*-labeled exit condition.

For the next step in our work, we will focus on the implementation of a prototype of test data generator based on models written in the SPTL language and on its evaluation on case studies. We intend to extend the language to include test oracles, for comparing the actual system outputs with the expected ones, and for observing functionalities and properties of the program under test.

References

1. Benveniste, A., Berry, G.: The synchronous approach to reactive and real-time systems. In: Proceedings of the IEEE. pp. 1270–1282 (1991)
2. Carroll, J.: Five reasons for scenario-based design. Interacting with computers 13(1), 43–60 (2000), http://archive.itee.uq.edu.au/~comp3503/Resources/_pdf/CarrollScenariosIwC.pdf
3. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE 79(9), 1305–1320 (Sep 1991)
4. Harel, D., Pnueli, A.: Logics and models of concurrent systems. chap. On the development of reactive systems, pp. 477–498. Springer-Verlag New York, Inc., New York, NY, USA (1985), http://dl.acm.org/citation.cfm?id=101969.101990
5. Jahier, E., Raymond, P., Baufreton, P.: Case studies with lurette v2. STTT 8(6), 517–530 (2006), http://dblp.uni-trier.de/db/journals/sttt/sttt8.html#JahierRB06
6. Marre, B., Blanc, B.: Test selection strategies for lustre descriptions in gatel. Electr. Notes Theor. Comput. Sci. 111, 93–111 (2005), http://dblp.uni-trier.de/db/journals/entcs/entcs111.html#MarreB05
7. Musa, J.D.: Operational profiles in software-reliability engineering. IEEE Softw. 10(2), 14–32 (Mar 1993), http://dx.doi.org/10.1109/52.199724
8. Papailiopoulou, V., Seljimi, B., Parissis, I.: Revisiting the Steam-Boiler Case Study with LUTESS : Modeling for Automatic Test Generation. In: Proceedings of the 12th European Workshop on Dependable Computing, EWDC 2009. p. 8 pages. Helène Waeselynck, Toulouse, France (May 2009), http://hal.archives-ouvertes.fr/hal-00381548
9. Seljimi, B., Parissis, I.: Using clp to automatically generate test sequences for synchronous programs with numeric inputs and outputs. In: Proceedings of the 17th International Symposium on Software Reliability Engineering. pp. 105–116. ISSRE '06, IEEE Computer Society, Washington, DC, USA (2006), http://dx.doi.org/10.1109/ISSRE.2006.49
10. Wu-Hen-Chang, A., Adamis, G., Erős, L., Kovács, G., Csöndes, T.: A new approach in model-based testing: Designing test models in ttcn-3. In: Proceedings of the 15th International Conference on Integrating System and Software Modeling. pp. 90–105. SDL'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-25264-8_9
11. Zucatto, F., Biscassi, C., Monsignore, F., Fidelix, F., Coutinho, S., Rocha, M.: Zigbee for building control wireless sensor networks. In: Microwave and Optoelectronics Conference, 2007. IMOC 2007. SBMO/IEEE MTT-S International. pp. 511–515 (Nov 2007)