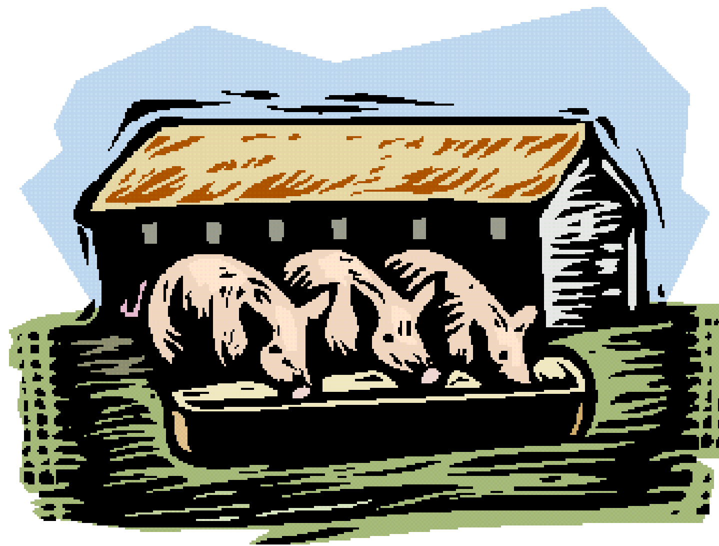
Model-based online testing — A case study on SKOV Feeding System

Department of Computer Science

supervisor: Brian Nielsen



SSE4 d605a

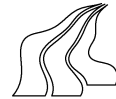
Weiwei Zheng

09.2006- 06.2007

AALBORG UNIVERSITY

Department of Computer Science

Aalborg University



Software Systems Engineering

THEME:
Model-based online testing

TITLE:
A case study on SKOV Feeding System

SUBJECT:
Distributed systems and semantics

PROJECT PERIOD:

September 2006 - June 2007

GROUP:
d605a

GROUP MEMBERS:
Weiwei Zheng

SUPERVISORS:
Brian Nielsen

NUMBER OF DUPLICATES: 4

NUMBER OF PAGES IN REPORT:
114

Abstract

The feeding system is the product of the SKOV company. It will be used in the livestock buildings. It needs to be demonstrated, that it can be implemented correctly in terms of its system specification. During the project period, we model its system specification in UPPAAL. Provided the emulate feeding system(Dub99) as the implementation under test(IUT), we perform the model-based testing manually using the feeding system models. It is demonstrated that, provided the IUT and the adapter, the model can be applied on the UPPAAL TRON engine for model-based black-box online testing.

Contents

1 Introduction **9**

1.1 Software testing 9

1.2 The testing life cycle 9

1.3 Black box conformance testing 10

1.3.1 Black-box testing 10

1.3.2 Conformance testing 11

1.4 Model-based testing 11

1.5 Offline testing Vs. Online testing 12

1.6 Project motivation & description 14

1.6.1 motivation 14

1.6.2 Case description 15

1.7 Structure of the report 15

2 Preliminaries **16**

2.1 Timed Input/Output Transition Systems 16

2.2 UPPAAL 18

2.2.1 Timed Automata 18

2.2.2 UPPAAL toolset 20

2.3 Relativized Timed Input/Output Conformance 22

2.4 TRON 24

2.5 Testing setup 27

2.6 Case study of the Dimmer(smart lamp) 27

2.6.1 Testing purpose & Testing setup 28

2.6.2 Model description 29

2.6.3 Hints in Adapter Modeling 33

2.6.4 IUT 34

2.6.5 Experiments 38

3 Case study: the feeding system **40**

3.1 System Overview 40

3.2 System functionality decomposition 40

3.2.1 Demand feed function 43

3.2.2 Weighing function 44

3.2.3 Calibration function 46

3.3 System structure 46

3.4 Emulate Labview system 48

3.5 Summary 51

4 Model1 - Demand feed **53**

4.1 Model purpose and structure 53

4.2 Model description 54

4.2.1 The feed demand template 54
4.2.2 The Dol7 template 55
4.2.3 The Viper1 template 56
4.2.4 The Viper2 template 58
4.2.5 The shutter template 58
4.2.6 The alarm template 59
4.3 Model checking 59
4.4 Manual testing 60
4.4.1 Test case1 61
4.4.2 Test case2 63
4.4.3 Test case3 65
4.5 Ambiguity 67
4.6 Summary 68

5 Model2 - Calibration 71

5.1 Model purpose and structure 71
5.2 Model description 72
5.2.1 The CalibrateUser template 72
5.2.2 The Viper1 template 72
5.2.3 The DrumPosition template 74
5.2.4 The Action template 74
5.2.5 The CheckSignal template 74
5.3 Model checking 75

5.4 Manual testing 75
5.4.1 System modification 75
5.4.2 Test case 1 76
5.4.3 Ambiguity 78
5.5 Summary 79

6 Model3 - Weighing **80**

6.1 Model purpose and structure 80
6.2 Model description 81
6.2.1 The Demand template 81
6.2.2 The Viper1 template 82
6.2.3 The OneWeighSub template 83
6.2.4 The RollDrumSub template 84
6.2.5 The Drum template 85
6.2.6 The SiloAuger template 85
6.2.7 The CheckSignal template 86
6.3 Model checking 86
6.4 Manual testing 87
6.4.1 Test case 1 87
6.5 Ambiguity 91
6.6 Summary 92

7 Conclusion & Future work **93**

7.1 Summary 93
7.2 Conclusion 94
7.3 Future work 94

A Climate Controller **96**

A.1 Cooling Control System 96
A.1.1 Control Process 96
A.1.2 Model Structure 98
A.2 Model description 99
A.2.1 Environment 99
A.2.2 SUT 103
A.3 Temperature Curve 106
A.4 Model checking 108

Acknowledgement

We would like to thank our supervisor Brian Nielsen, whose academic guidance and illuminating suggestion have encouraged and helped us greatly during this project. Our heartfelt gratitude goes to him for his patience and advise at all times.

We would like to thank Marius Mikucionis for his support.

We would like to thank Shuhao Li for his reviewing and criticizing on the near-final manuscript.

GROUP MEMBERS

Weiwei Zheng

Chapter 1 Introduction

1.1 Software testing

In the software development life cycle, testing is an importance phase to improve the quality of the product. The objective of the testing is to find errors, to evaluate the product and to check whether the product fulfills the needs of the customers or not.

The missions of testing are summarized as follow [1]:

1. To demonstrate that the product performs each function as intended;
2. To demonstrate that the internal operation of the product performs according to specification and all internal components have been adequately exercised;
3. To increase our confidence in the proper functioning of the software;
4. To show the product is free from defects.

Due to the inherent complexity of software products, software testing is usually a difficult work. With the widespread deployment of safety-critical software systems and software devices and controllers, testing is becoming more and more important. Traditional manually testing is labor-intensive, error-prone and tedious. Therefore, it is beneficial for us to look to automated approach of software testing.

1.2 The testing life cycle

An appropriately formed and performed test process can contribute to the success of the product. Although testing differs between testers, the cycle to testing is almost the same. The figure 1.1 shows one simple cycle for software testing.

It starts from the test PLANNING, which is required to propose a systematic way to test a software. Typically, it contains detailed and clear idea of how the execution will be. In the DESIGN phase, testers will analyze the requirements in the software development life cycle, and to determine which one is testable and what's needed to perform this test. The third step,

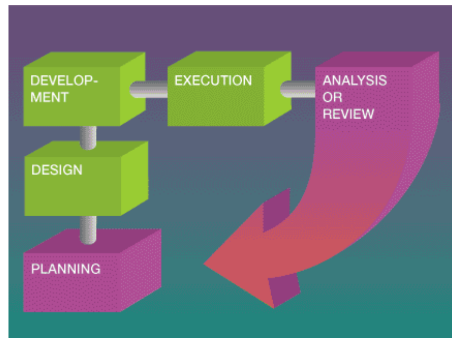


Figure 1.1: Software testing cycle[2]

DEVELOPMENT, is test specification. Basically, here test cases are being generated, and being collected into test suites. Based on those plans of the test, testers EXECUTE the test, and give out the result. Then, ANALYSIS the results, report the test effort and to determine whether or not the product is ready for release, if not, do REVIEW(retest).

1.3 Black box conformance testing

1.3.1 Black-box testing

According to the software accessibility, there are two methods to derive test case. One is white-box testing, the other is black-box testing. See figure1.2.

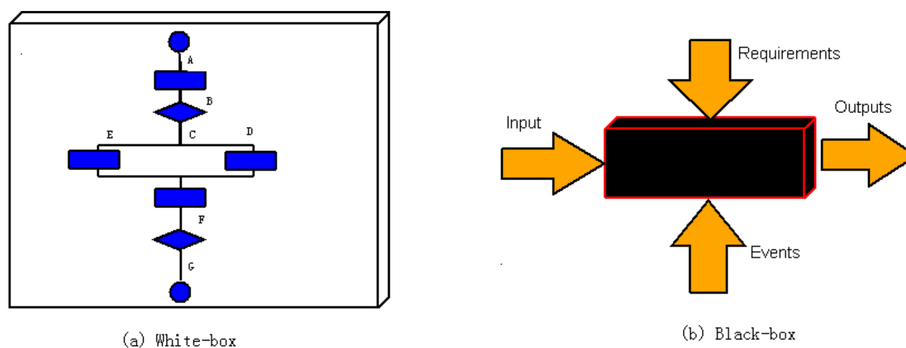


Figure 1.2: White-box testing VS. Black-box testing [3]

White-box(glass-box) is referred to as code coverage method, aiming at checking the basic structure of the program. With this method, the test cases are derived (1) to ensure that all the independent paths have been covered at least once, or (2) to execute the both sides of all the logical decisions, or to (3) perform all the loop statements within their restriction boundary, or to (4) execute to ensure all the internal data structure. See Figure 1.2 (a), in this case, it will execute paths ABCDFG, ABCEFG, with intention to cover both sides of the decision.

In contrast to white-box testing, there is another way to derive test case, which is called Black-box testing, that does not care about the code. It only uses the functional specification documents. It tries to find out: (1) functions which are concluded not according to the specification, or (2) interface mistakes, or (3) incorrect data structure or external access manner, or (4)initial and terminal phase mistakes.

These two testing methods are complementary. White-box tests the control structure. It is executed at the early phase of the testing process. Whereas Black-box pays attention to the functions. It will be applied in the later phase of the testing process.

1.3.2 Conformance testing

With respect to the different aspects of the system behaviors, testing can be classified into conformance testing, performance testing, stress testing and others.

Conformance testing [4] Conformance testing checks the implementation of the system behaviors against the functional specification of the system [5]. It can be dealt with black-box testing method, where the internal states of the implementation under test(IUT) are assumed to be unavailable. Test cases are applied to IUT, and then "pass/fail" verdicts are assigned according to whether the observations conform to the system specification or not.

Performance testing [6] Performance testing is to figure out how fast the particular system aspect performs with respect to a certain workload. It is used for different purposes. For example, its result can be used to decide which one performs better among several systems. It can be also used to check whether the system meets the performance criteria or not. Or it can measure that which parts of a device or the system contributes to that bad performance.

Stress testing [7] This form of testing evaluate the stability of a particular system under heavy load. Compared to conformance testing, it prefers to determine how easily the software crash under stressful conditions, e.g. insufficient computation resources. It can be classified into robustness, availability, and error handling.

1.4 Model-based testing

System specification can be written in natural languages, e.g. English or Danish, that is so called informal specification. The informal specification leads to different interpretations. So that it will not be clear about what the system should do, sequentially, test can not be explicitly performed [8].

Formal methods are proposed to resolve those defects. With formal methods, testers interpret systems into mathematical models. The test cases will be generated from the models. The main advantage of the model-based testing can be collected as followed [9]:

- In contrast to the informal specification, the specifications based on formal models are precise and unambiguous.
- The notation of formal models, makes the automated error checking possible, so that error can be found in the early stages of the system development life cycle.
- Formal models make the automation of the whole testing process possible. Test cases can be produced algorithmically from the formal models. After executing those test cases, observations will be compared to the expected outputs to make the "pass/fail" verdict.

There are many kinds of formal models, e.g. Decision Tables, Finite State Machine, Markov process. Model-based testing is a method for generating test cases automatically from behavioral models of the system under test(SUT). The model is the presentation of the system requirements and the functional specifications. It can be constructed before or during the development of the SUT. It can also be developed from the complete system.

A variety of the formal models exist for modeling the system behavior. In our project, we prefer to use timed automaton(TA). TA is extended the Finite State Machine with the real-value of clocks. That makes it an expressive modeling tool for real-time systems.

The test cases are produced from the model. They are consisted of the specified system inputs and the expected outputs. Then the inputs are executed in the implementation under test(IUT). And outputs of that test execution will then be compared with the expected outputs to determined success or failure.

Model-based testing makes it possible for the tester to easily guide the test case generation according to the specific interests. However, to perform model-based testing requires highly skilled testers. For example, as TA will be used in this project, testers are required to have knowledge about the theory of the timed automata and the modeling techniques. In this sense performing model-based testing is not always a good choice

1.5 Offline testing Vs. Online testing

There are two testing manner[10], one is called Offline or batch testing, as shown on the Figure 1.3 a, the other is online testing or on-the-fly testing, see Figure 1.3 b.

Offline approach in Figure 1.3 a does the test generation and the test execution separately. It completely generates the test cases, and stores them in a test notation language, like TTCN[10].

Then test cases are executed against the IUT, and determine "pass" or "fail" for the outputs according to the specification. For all of the time constraints in the system specification have been solved in advance.

Online testing in Figure 1.3 b, joins the two steps: test generation and the test execution together. Here a single test event is generated from the model each time, and is executed on the SUT at once, and then the Outputs from the SUT and the time of occurrence are compared to the specification to check the conformance requirement. And new cycle is performed so forth until being stopped.

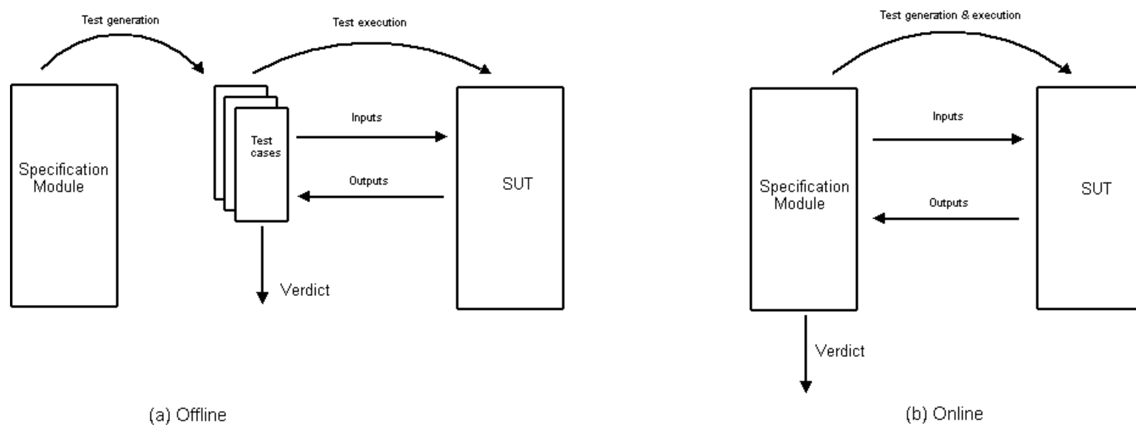


Figure 1.3: Offline VS. Online test gerneration

There are some remarks for both the offline testing and the online testing.

- Offline testing is better choice for applying the manual or semi-manual test suite preparation[4]. Humans can make selection among test cases according to the test requirements. However it is not fast enough, if considering the speed requirement at run-time.

- Using offline strategy, all the required time constraints can be resolved during test generation. So offline strategy can also guarantee some restrictions, for example, the given test-goal should consume as few resource as possible. All those may result in producing easy and fast test case for execution. Compared to the online testing, that do all the computations in run-time, the pre-computations of the offline approach make offline easier to satisfy the IUT's real-time requirements.

  However, sometimes using offline generation is too expensive. For instance, in the non-deterministic cases, outputs are infeasible to be predicted, it may lead to a very large test cases. And it is not fit for handling the specification with huge size as well, because it may cause state-explosion problem. But the complexity can be resolved by using means of the user guidance and the online(on-the-fly) approaches[10].

  Compared to offline testing, online method do not need to predict the test steps. That results in reducing the state-space-explosion. Online test makes a single test run possible to last for a very long time, since complicated test cases can be generated and executed without interrupted.

- During the test execution, translation is necessary for the interface with the IUT[4]. Online approach is more difficult, because all the translation must be done at run-time. It requires such a run-time translator must meet the time constraints, and that translator must be generic enough for being reusable. Whereas, offline derivation, is possible to pre-computed the abstract test cases into concrete ones with all the mapping details included.

1.6 Project motivation & description

1.6.1 motivation

Although testing is still the dominating technique for software quality assurance, it is in practice labor-intensive, error-prone, and tedious. As an emerging technique, model-based testing (MBT) offers the advantages of being a systematic, rigorous test method that can be fully automated and can be used in the early stage of the software life cycle. These features make it very attractive to industry. The DSS group has expertise and competence in real-time modeling, verification and testing, and has been maintaining cooperations with companies such as Danfoss and SKOV. These put us at an advantageous position to carry out further research on and further application of our techniques in this area.

During the passed academic year, we have already gained the knowledge on the modeling, and the basic theory of MBT. In order to validate the theory of model-based black box conformance testing, we use tron in the industrial case study offered by the SKOV company. The case study is about demonstrating the functionality of the SKOV feeding system.

The MBT is suitable for the SKOV case. Firstly, it is because that, the SKOV intends to verify the system behavior. At this early stage of designing the feeding system, MBT helps to expose the ambiguities in the system specification. Using MBT, makes it possible to produce many non-repetitive and the meaningful test cases. It makes it easy to update the test cases in terms of the changed system specification by means of modifying the model.

With the current specification of the feeding system, the model can be built. And the expected test case will be generated from the model, including the expected inputs and outputs. The inputs are used to enabled the corresponding actions of the implementation (Dub99). The outputs from the Dub99 are compared to the expected output, and pass/fail verdict will be made. The test results help to make further decisions, e.g. the failure may be lead to the modification of the model.

1.6.2 Case description

At the beginning of the testing, an industrial product called climate controller is proposed by the SKOV company. The expected functions of that controller need to be maintained. So that, we planed to apply the model-based online testing on this industrial case. We spend the first half project period in translating the specification of the climate controller into the UPPAAL model. Then, it is found that, the real climate controller product can not be offered. That means, we do not have the implementation under test(IUT) for the model-based online testing using TRON. The testing can not be moved further.

Then we accepts another proposal, which is also from SKOV. It is the feeding controlling system, name viper. We complete the modeling phase. That is, modeling the system specification of the feeding system, as well as its working environment(Dol99B) in UPPAAL. And the expected property is also verified. However, due to the limit of the time, we do not perform the online testing using tron. With the emulate feeding system(Dub99), which is used to be the IUT, we applied manually model-based testing using the UPPAAL feeding system.

1.7 Structure of the report

The remainder of the report is organized as follow. Chapter 2 give you the semantics necessary for the online model-based testing, and also the testing setup structure. After that, we do a small case study for smartlamp, present you a general idea for the testing using TRON. In chapter 3, we presents the overview of the feeding system testing case. In chapter 4, 5, 6, we propose three model systems based on the three function of the feeding system. We applied each model system for manually testing with the Emulate feeding system(implementation). Chapter7, summaries the work we have done during the project period, makes the conclusion, and outlines the work for future. In Appdendix A, we describe the Skov climate controller model for testing purpose in UPPAAL, which is done in the first semester.

Chapter

Preliminaries 2

The goal of the thesis is to use **UPPAAL TRON** to perform model-based online black-box conformance testing. Tron uses relativized timed input/output conformance relation [11] as the implementation relation, with intention to check whether the Implementation Under Test (IUT) conforms to its specification S when being operated under environment assumptions ξ.

This section introduces the Timed Input/Output Transition Systems(TIOTS), Timed Automata and the conformance relation we used named Relativized Input/Output Conformance relation, and presents the testing setup.

More information about this can be found in [12], [11] and [13].

2.1 Timed Input/Output Transition Systems

We assume there is a set of actions Act that is consisted of input actions Act_{in} and output actions Act_{out}, where $Act_{in} \cap Act_{out} = \emptyset$. We also assume that there is a distinguished unobservable action τ, where $\tau \notin Act$. We denote that $Act \cup \{\tau\} = Act_{\tau}$.

DEFINITION **1.** *A labelled transition systems(LTS)* S *is a triple* $(S, Act_{\tau}, \rightarrow)$*, where:*
S *is a set of states,*
Act_{τ} *is the set of actions,*
$\rightarrow \subseteq S \times Act_{\tau} \times S$ *is a transition relation such that* $s \xrightarrow{a} s^{'}$*, iff* $(s, a, s^{'}) \in \rightarrow$*, where* $a \in Act_{\tau}$*,* s*,* $s^{'} \in S$*.*

DEFINITION **2.** *A timed input/output transition systems(TIOTS, where TIOTS* $\subseteq$ *LTS)* S *, is a tuple* $(S, s_0, Act_{in}, Act_{out}, \rightarrow)$*, where:*
S *is a set of states,*
$s_0 \in S$*, is the initial state,*
$\rightarrow \subseteq S \times (Act_{\tau} \cup \mathbb{R}_{\geq 0}) \times S$ *is a transition relation, which satisfies those properties:*

time determinism: if $s \xrightarrow{d} s^{'}$*, and* $s \xrightarrow{d} s^{''}$ *then* $s^{'} = s^{''}$*,*

time additivity: if $s \xrightarrow{d_1} s^{'}$*, and* $s^{'} \xrightarrow{d_2} s^{''}$ *then* $s \xrightarrow{d_1+d_2} s^{''}$*,*
$d, d_1, d_2 \in \mathbb{R}_{\geq 0}$*, where* $\mathbb{R}_{\geq 0}$ *denotes non-negative real numbers.*

DEFINITION **3.** *Let* $a, a_1, a_{2...n} \in Act$, $\alpha, \alpha_1, \alpha_{2...n} \in Act_\tau \cup \mathbb{R}_{\geq 0}$ *and* $d, d_1, d_{2...n} \in \mathbb{R}_{\geq 0}$*. we denote that:*
$s \xrightarrow{a}$ *iff* $\exists s'$ *such that* $s \xrightarrow{a} s'$,
$s \overset{a}{\Rightarrow} s'$ *iff* $\exists s'$ *such that* $s \xrightarrow{\tau^*}\xrightarrow{a}\xrightarrow{\tau^*} s'$,
$s \overset{d}{\Rightarrow} s'$ *iff* $\exists s'$ *such that* $s \xrightarrow{\tau^*}\xrightarrow{d_1}\xrightarrow{\tau^*}\xrightarrow{d_2}\xrightarrow{\tau^*} ... \xrightarrow{\tau^*}\xrightarrow{d_n}\xrightarrow{\tau^*} s'$, *where* $\sum_{i=1}^{N} d_i = d$.
We extend $\Rightarrow$ *to sequence of actions and delays in the usual manner.*

DEFINITION **4.** *The Timed Input/Output Transition System (TIOTS)* S *is strongly input enabled, iff* $\forall s \in S, \forall i \in Act_{in}$*, such that* $s \xrightarrow{i}$
TIOTS is strongly input enabled implies it can accept any input in any state.

DEFINITION **5.** *The Timed Input/Output Transition Systems (TIOTS)* S *is non-blocking, iff*

$\forall s \in S, \forall t \in \mathbb{R}_{\geq 0}, \exists \sigma = d_1 o_1 ... d_n o_n$*, such that* $s \overset{\sigma}{\Rightarrow}$ *and* $\sum_{i=1}^{N} d_i \geq t$.
Where $d_1, d_{2...n} \in \mathbb{R}_{\geq 0}$*,* $o_1, o_{2...n} \in Act_{out}$*.*

Thus we assume the TIOTS S is strongly input enabled and non-blocking. It will not block time in any input enabled environments. We also assume that the S with two properties: isolated input, and deterministic.

DEFINITION **6.** *The Timed Input/Output Transition Systems (TIOTS)* S *has isolated outputs:*
if $s \xrightarrow{o}$ *then* $s \overset{\tau}{\nrightarrow}$ *and* $s \overset{d}{\nrightarrow}$ $(d > 0)$*,*
if ($s \xrightarrow{o} \wedge s \xrightarrow{o'}$ *) then* $o = o'$*.*

DEFINITION **7.** *The Timed Input/Output Transition Systems (TIOTS)* S *is deterministic if* $\forall \alpha \in Act_\tau \cup \mathbb{R}_{\geq 0}, \forall s \in S$*, such that, if (*$s \xrightarrow{\alpha} s' \wedge s \xrightarrow{\alpha} s''$*), then* $s' = s''$*.*

We assume there are two input enabled, non-blocking TIOTS, $S = (S, s_0, Act_{in}, Act_{out}, \rightarrow)$ and $\xi = (E, e_0, Act_{out}, Act_{in}, \rightarrow)$. ξ is regarded as an environment of the S. e_0 is the initial state of the E, where E is the set of the states of the environment ξ. Similarly, $s_0 \in S$ is the initial state of S. The input actions of environment ξ are the output actions of S, whereas, the input actions of S are identical to the output actions of ξ.

DEFINITION **8.** *Let* $S||\xi$ *be a closed system parallelly consisted of* S *and* ξ*. The TIOTS* $S||\xi$ *is of the form* $(S \times E, (s_0, e_0), Act_{in}, Act_{out}, \rightarrow)$*, where* $\rightarrow$ *is defined as:*

$$\frac{s\xrightarrow{\alpha}s' \; e\xrightarrow{\alpha}e'}{(s,e)\xrightarrow{\alpha}(s',e')} \qquad \frac{s\xrightarrow{\tau}s'}{(s,e)\xrightarrow{\tau}(s',e)} \qquad \frac{e\xrightarrow{\tau}e'}{(s,e)\xrightarrow{\tau}(s,e')} \qquad \frac{s\xrightarrow{d}s' \; e\xrightarrow{d}e'}{(s,e)\xrightarrow{d}(s',e')}$$

In the following paragraph we will preset you the definition of the Time Automaton (TA), and its semantics with regard to TIOTS.

2.2 UPPAAL

UPPAAL is mature integrated tool environment for modeling, verification, and simulation of real-time system. In UPPAAL, the system is modeled as networks of automata extended with integer variables, structured data types, synchronizing channels [14], [12].

2.2.1 Timed Automata

Using UPPAAL-TRON (TRON for short) to carry out the testing, is based on the formal description of the real-time embedded systems as timed automata in UPPAAL. By definition a timed automaton is a finite state machine extended with clock variables. Clocks progress synchronously and can evaluate to real numbers (as can be seen in the definition of semantics). This notion of dense time, implicit to a timed automaton, makes it a most useful tool to model systems where time plays an important role and different entities of the system should be able to interact through communication.

DEFINITION **9** (Timed Automaton)**.** *A timed automaton is 6-tuple* $(L, l_0, C, Act_\tau, E, I)$ *where:*

L *is the set of locations*

l_0 *the initial location,* $l_0 \in L$

C *is the set of clocks*

Act_τ *a set of actions, co-actions and internal* τ *- actions*

$E \subseteq L \times Act_\tau \times B(C) \times 2^C \times L$ *is a set of edges between locations with an action, a guard and a set of clocks to be reset*

$I : L \to B(C)$ *assigns invariants to locations*

B(C) is the set of conjunctions of conditions of the form $x \bowtie c$ or $x - y \bowtie c$ where $x, y \in C$ (are clocks), $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.

Invariants on locations will be better understood after we give the semantics of TA. Before that, let us introduce the notion of clock valuation: a clock valuation is a function $u : C \to \mathbb{R}_{\geq 0}$ that assigns a non-negative real number tos each elements in the **C**. Let $\mathbb{R}^C$ be the set of all clock valuations. And we assume that $u_0(x) = 0$ for all $x \in C$.

We adopt the notation $u \in I(L)$ and $u \in g$ meaning that the clock valuation u satisfies the invariant in location L and that the clock valuation u satisfies the guard g.

DEFINITION **10** (Semantics of TA). *Let (L,l_0,C,Act_τ,E,I) be a TA. The semantics is defined as a TIOTS over state of the form $s = (l, u)$, where $l \in L$ is a location, and the $u \in \mathbb{R}^C_{\geq 0}$ is a clock valuation satisfying the invariant of the location l, C is a set of non-negative real valued clocks. There are two kinds of the transitions:*

delay transition $(l, u) \xrightarrow{d} (l, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \Longrightarrow u + d' \in I(l)$
(The values of all clocks of the automaton are increased by the amount of the delay, and still satisfies the invariant of the same location)

discrete transition $(l, u) \xrightarrow{a} (l', u')$ *if* $\exists e = (l, \alpha, g, r, l') \in E$ *s.t.* $u \in g$, $u' = [r \mapsto 0]u$ *and* $u' \in I(l')$
(When fires the edge $e = (l, \alpha, g, r, l')$, the guard g is satisfied by u, the clock valuation of the target state u' simply maps to 0 according to the updates r, and satisfies the invariants on l')

Let us note that for a TA to be on a given location it **must** satisfy the location invariant and the guard condition in its outgoing edge, thus, a location with an invariant must always have an outgoing edge.

DEFINITION **11** (Network of Timed Automaton). *A Network of timed automata (NTA) is a collection of n TA $A_i = (L_i, l_i^0, C, Act, E_i, I_i), 1 \leq i \leq n$ over a common set of clocks (C) and of actions (Act) where a location is now a vector $\bar{l} = (l_1, ..., l_n)$, the initial location is a vector $\bar{l}_0 = (l_1^0, ..., l_n^0)$ and the invariant function over such a position vector is defined as the conjunction of the invariant functions of each TA over the respective location $I(\bar{l}) = \bigwedge_i I_i(l_i)$.*

The semantics of NTA is defined. And it is useful to adopt the notation $\bar{l}[l_i'/l_i]$ that denotes the vector where the ith element l_i of $\bar{l}$ is replaced by l_i'.

DEFINITION **12** (Semantics of NTA). *Let $A_i = (L_i, l_i^0, C, Act, E_i, I_i), 1 \leq i \leq n$ be an NTA. Let $\bar{l}_0 = (l_1^0, ..., l_n^0)$ be the initial location vector. The semantics is defined as a TIOTS over state of the form $s = (\bar{l}, u)$, where $\bar{l} \in (L_1 \times ... \times L_n)$ is a location, and the $u \in \mathbb{R}^C_{\geq 0}$ is a clock valuation satisfying the invariant of the location $\bar{l}$, C is a set of non-negative real valued clocks. The transitions are defined as:*

- $(\bar{l}, u) \rightarrow (\bar{l}, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(\bar{l})$
- $(\bar{l}, u) \rightarrow (\bar{l}[l_i'/l_i], u')$ *if* $\exists l_i \xrightarrow{\alpha,g,r} l_i'$, *where* $\alpha \in Act \cup \{\tau\}$ *s.t.* $u \in g, u' = [r \mapsto 0]u$ *and* $u' \in \bar{l}[l_i'/l_i]$

- $(\bar{l}, u) \rightarrow (\bar{l}[l_i'/l_i, l_j'/l_j], u')$, *where* l_i', $l_j' \in \bar{l}'$ *and* $\bar{l}' = (..., l_i', ..., l_j', ...)$, *if* $\exists l_i \xrightarrow{a!g_i r_i} l_i'$ *and* $\exists l_j \xrightarrow{a?g_j r_j} l_j'$, *where* $a \in Act$, *s.t.* $u \in (g_i \wedge g_j)$, $u' = [r_i \cup r_j \mapsto 0]u$ *and* $u' \in I(\bar{l}')$

(This is when two TA in the system synchronize through an action and a co-action, both fire an edge).

2.2.2 UPPAAL toolset

UPPAAL toolset is consisted of the graphical user interface and the model checker engine [12], [15]. The graphical user interface, is implemented in Java. The model checker engine, also called verifier. It is by default that, two parts are run on the same OS. The latest release version of UPPAAL is available for Windows, Linux, Solaris and Mac OS X.

The UPPAAL tool serves for modeling the system into the network of the timed automata using the graphical editor, simulating it to check whether the dynamic system behavior is expected, verifying the expected system properties. Those three intentions are reflected by the graphical interface. It contains the editor, the simulator and the verifier. They can be accessed via three tabs marked with their names respectively. The following sections will give you a general idea about them, for more detail, please refer to the UPPAAL tutorial [12].

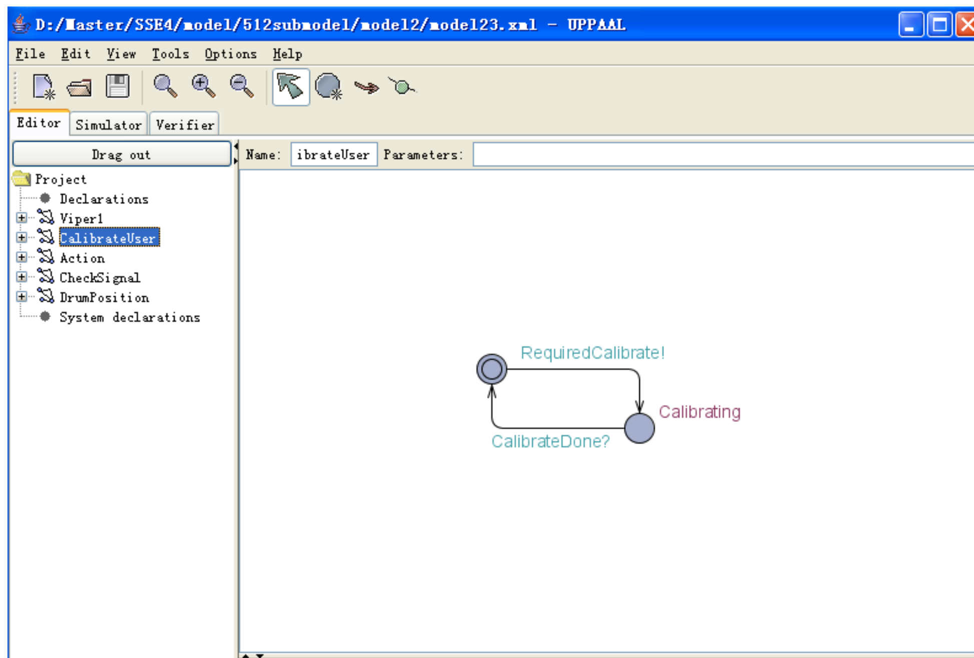


Figure 2.1: The snapshot of the UPPAAL editor

Editor

The editor(see the Figure 2.1), is mainly consisted of two parts: the template drawing panel and the tree panel. The template drawing panel is used to design the template. A template is consisted of the location and the edge (transition). The location may be labeled with name, status (initial, urgent, committed), invariant. The edge may be labeled with guard conditions, synchronized signal, variant or clock updates. Each template must have at most one initial location. Templates will be instantiated into the process in the system declaration part. All the processes in the system consists that, is so called the network of the timed automata.

Using the tree panel on the left of the Figure 2.1, can easily access to the Declarations part, different template object, the System declaration part. The Declaration parts, contains the declaration of the global invariants, variants, clocks and the synchronized channels. Each template objects contains, the declaration of the local invariants, variants, clocks, and the channels. It also has the drawing of the template behavior. The System declaration is the place to instantiate the template into process, and list and declare all the processes.

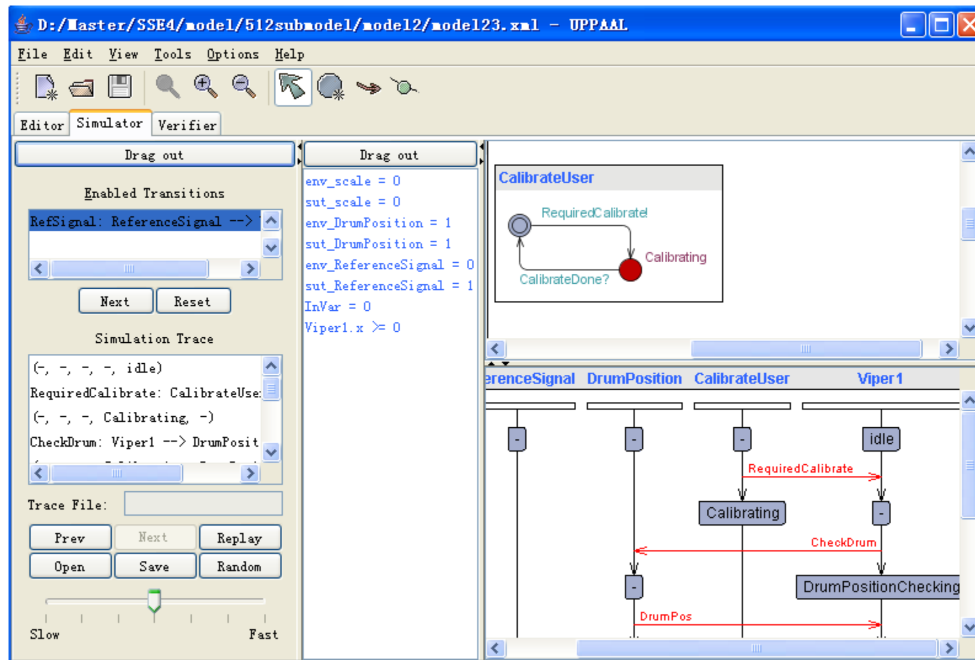


Figure 2.2: The snapshot of the UPPAAL simulator

Simulator

The Figure 2.2 is the snapshot of the simulator. It mainly contains five panes, each pane concerns different aspects of the current simulation. The top left pane indicates the possible transition for next step. The one below it, shows the state transited so far. The middle pane displays all the value of all the variables and the clock constraints of current state. The one in the top right, including all the automata, and the current system state by means of turning all the active locations red. The right bottom pane is the message sequence chart, describing the synchronized between process so far.

Verifier

The Figure 2.3 is the verifier. The overview panel lists the properties, that need to be checked. The properties UPPAAL query language. That query language is a subset of the CTL. It contains the state formulae and the path formulae [12]. The state formulae describes whether the formulae is true at some particular state. The path formulae describes whether the states is true

for some particular path. The path formulae are used to describe reachability, safety, liveness properties. The reachability properties asks, whether the specified states are reachable. The safety is used to detect that, the unexpected state will never occur. And the liveness properties are detected, when the specified state are required to be eventually reached.

The comment panel is used to explain the meaning of the current selected property, that is presented in the query panel. The status panel shows the checking status presented by server. When it is demonstrated to be a satisfied property, it will be marked green, otherwise marked red.

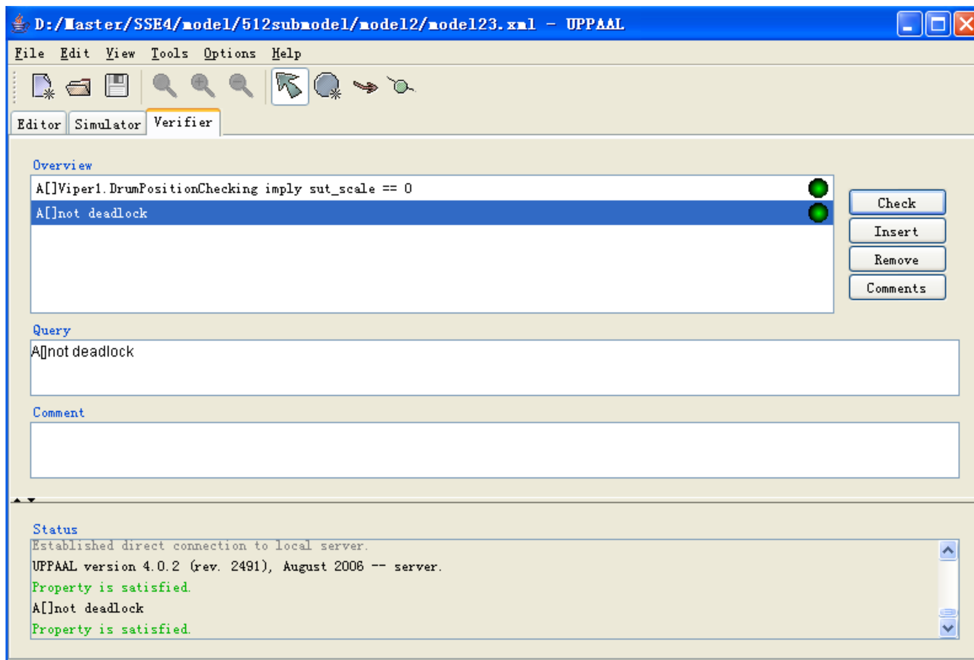


Figure 2.3: The snapshot of the UPPAAL verifier

2.3 Relativized Timed Input/Output Conformance

The following section presents the notion of relativized Timed Input/Output Conformance[11]. It is obtained from the input/output conformance relation (ioco) of Tretmans[16]. The relativized conformance makes sure that the implementation behaves exactly according to the specification, i.e. the following two ways are not allowed:

- produce an output at a time which is not allowed by the specification
- omit to produce an output within certain delay which is required by the specification.

Let an observable timed trace σ with the form $\sigma = d_1 a_1 d_2 a_2 ... d_n a_n$ where $\sigma \in (Act \cup \mathbb{R}_{\geq 0})^*$.

DEFINITION **13.** *The observable time traces TTr(s) of a state s is defined as:*

$$TTr(s) = \{\sigma \in (Act \cup \mathbb{R}_{\geq 0})^* \mid s \overset{\sigma}{\Rightarrow}\}$$

Like in Figure 2.4 (a), there exists a time trace $\sigma_1 = 10 \cdot coin \cdot 20 \cdot req \cdot 20 \cdot weakcoffee$, where $\sigma_1 \in TTr((L0, 0))$

DEFINITION **14.** *The set of states that can be reached after time trace σ, is defined as s After σ:*

$$s\ After\ \sigma = \{s' \mid s \overset{\sigma}{\Rightarrow} s'\},\ S'\ After\ \sigma = \bigcup_{s \in S'} s\ After\ \sigma$$

$$where\ s \in S,\ S' \subseteq S.$$

See Figure 2.4 (a), if there exists $\sigma_1 = 10 \cdot coin \cdot 20 \cdot req \cdot 20 \cdot weakcoffee$ such that $(L0, 0) \xrightarrow{10} \xrightarrow{coin} \xrightarrow{20} \xrightarrow{req} \xrightarrow{20} \xrightarrow{weakCoffee} (L0, 20)$, we say $(L0, 0)$ $After\ \sigma_1 = \{(L0, 20)\}$. Let's take $\sigma_2 = 10 \cdot coin \cdot 40 \cdot req$ as another example. Then we can conclude that, $(L0, 0)$ $After\ \sigma_2 = \{(L2, d_2), (L3, d_3) \mid 10 < d_2 < 30, 30 < d_3 < 50\}$.

DEFINITION **15.** *The set of observable outputs and delays of a state s or a state set S' is:*

$$Out(s) = \{\alpha \in (Act_{out} \cup \mathbb{R}_{\geq 0}) \mid s \overset{\alpha}{\Rightarrow}\},\ Out(S') = \bigcup_{s \in S'} Out(s)$$

$$where\ s \in S,\ S' \subseteq S.$$

Such as in Figure 2.4 (a), $Out((L0, 0)) = \{d \mid d \geq 0\}$ which means any instance can stay in location $L0$ as long as it wants to. Or e.g., $Out((L2, 0)) = \{a, d \mid a = WeakCoffee, 10 < d < 30\}$.

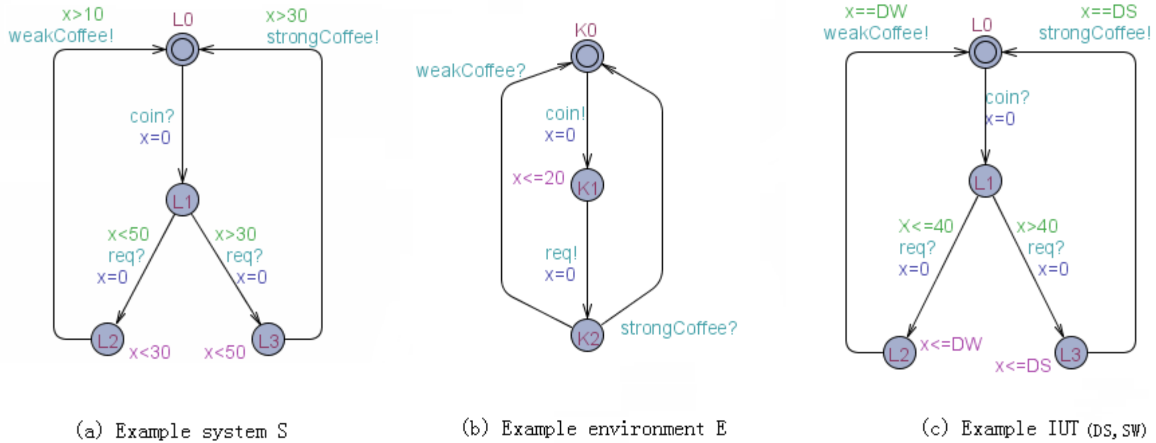
DEFINITION **16.** *The relativized input/output conformance relation between the systems states s($\in S$), t($\in S$) in terms of the given environment e($\in E$) is defined as:*

$$s\ rtioco_e\ t \Leftrightarrow \forall \sigma \in TTr(e)\text{: } Out((s, e)\ After\ \sigma) \subseteq Out((t, e)\ After\ \sigma)$$

That is, applying the same environment input action(s) on the system states s, t. If the outputs from the s is the subset of the outputs from t, then we can have $s\ rtioco_e\ t$. So that, we say, s is the right implementation concerning the specification t with the environment e .

Figure 2.4 [11] shows three timed automata separately indicating a coffee machine system specification, mimic user, and its implementation under test. Figure 2.4 (a) is the specification of the coffee machine system. It allows the user to pay at first, and then decide to get weak coffee or strong coffee by pressing the request button on the machine. Delaying for less then 30 time units will undoubtedly get weak coffee, however, delaying for more than 50 time units will certainly

obtain strong one. But the waiting time units between 30 and 50 results in nondeterministic choice, which waits for the IUT to decide which one to produce. After requesting, machine spends an additional time units 10 to 30 in producing weak coffee, or 30 to 50 in producing strong coffee.



(a) Example system S (b) Example environment E (c) Example IUT (DS, SW)

Figure 2.4: Example of **rtioco** relation

Figure 2.4 (b) defines a potential user of the coffee machine, who pays before requesting coffee and requests only before 20 time units, and we also assumes that he wants weak coffee. Figure 2.4 (c) is the deterministic implementation according to the value of the (DS, DW). It can output weak coffee after waiting for less then or equal 40 time units , can output strong one after waiting for more than 40 time units, both of them need additional brewing time DW and DS.

E.g., $IUT(60, 5)$ is not $rtioco_E$ with S , because the system specification S does not hold the possible time trace subset indicated by the $IUT(60, 5)$: $\{\ d_1 \cdot coin \cdot d_2 \cdot req \cdot 5 \mid d_1 \geq 0, d_2 < 30\ \}$, i.e., it may be too fast to product a weak coffee not even having time to insert a coffee cup. Well, it supposes to have another error, which may be producing strong coffee too slowly. However, the user E will never request strong coffee, thus we can ignore the value of DS. Such that, we can modify the pair (DS, DW) to IUT(60, 15), hence, IUT(60, 15) is $rtioco_E$ with S.

2.4 TRON

As an additional feature of UPPAAL, TRON can do the model-based black-box conformance testing of the real-time embedded systems. TRON is in the place of the IUT environment with two main functions: stimulating the IUT with Inputs, meanwhile, monitoring the Outputs to check whether the result conforms to the system behavior specification.

UPPAAL TRON randomly plays one of the three basic actions: sends an input to the IUT, waits for an output, restarts the IUT. The algorithm is shown in Algorithm1 [11]. The input for the

Algorithm 1 Test generation and execution algorithm for TRON

```
ℜ := {(s_0, e_0)}
while (ℜ ≠ ∅) ∧ (IterationNUM ≤ Max) do
    randomly choose to perform: action, delay, or restart
    action:
    if ENVoutput(ℜ) ≠ ∅ then
        randomly choose a ∈ ENVoutput(ℜ)
        send a to the IUT
        ℜ := ℜ After a
    end if
    delay:
    randomly chooses d ∈ Delay(ℜ)
    sleep for d time units and wake up on output o
    if o occurs at d' ≤ d then
        ℜ := ℜ After d'
        if o ∉ IMPoutput(ℜ) then
            return fail
        else
            ℜ := ℜ After o
        end if
    else
        ℜ := ℜ After d
    end if
    restart:
    ℜ := {(s_0, e_0)}
    restart IUT
end while
if ℜ = ∅ then
    return fail
else
    return pass
end if
```

Algorithm 1 is two TIOTS with the form of $S \parallel \xi$, where S represents the SUT, and ξ is on behalf of one of the environment. This algorithm is used to maintain the current reachable state set $\Re$ ($\subseteq S \times \xi$). It represents the current state that, the system specification can cover after the timed trace has been observed.

The main idea is the following. The test starts from where $\Re$ only contains the initial states of the system specifications and its environment assumptions. Whenever an input is produced, an output or a delay is observed, the state set $\Re$ will be updated. When an output or a delay is noted, its validity will be check according to the state set $\Re$. These will be done until no legal state in state set $\Re$, or the iteration of the test reaches its maximum number.

The three functions are used in the algorithm:

- $ENVoutput(\Re) = \{\, a \in Act_{in} \mid \exists(s, e) \in \Re.e \xrightarrow{a} \,\}$

  is the environment input actions for some states in the current state set $\Re$. It is empty only when environment model has no output to offer.

- $IMPoutput(\Re) = \{\, a \in Act_{out} \mid \exists(s, e) \in \Re.s \xrightarrow{a} \,\}$

  $IMPoutput(\Re)$ is the output action for some states in the current state set $\Re$

- $Delay(\Re) = \{\, d \mid \exists(s, e) \in \Re.e \overset{d}{\Rightarrow} \,\}$

  $Delay(\Re)$ is the set of real numbers, but it is not randomly picked up among the real-number if the environment must offer an input to IUT model before a specified moment(invariant).

According to the algorithm 1, one of the three actions shown below will be taken each time by TRON:

action : randomly choose an output among $ENVoutput(\Re)$, send it to IUT, and update $\Re$ according to $\Re$ $After\ a$

delay : randomly choose how long it should spend in waiting the output. If an output from IUT occurs less then the chosen time, and $\Re$ is updated according to the cost time. Then it checks whether the output is legal according to $IMPoutput(\Re)$. If it is, $\Re$ will be modified following the instruction $\Re$ $After\ o$, otherwise, announces the fail verdict.

If during the chosen time, no output is observed, $\Re$ is updates according to the total amount of this chosen time.

restart : initialize the state set $\Re$, and restart IUT.

2.5 Testing setup

Based on the theory we used, the testing framework is setup. The complete setup consists of three parts, the TRON, the Adapter and the IUT (represents the "Implementation under test"), as shown in Figure 2.5.

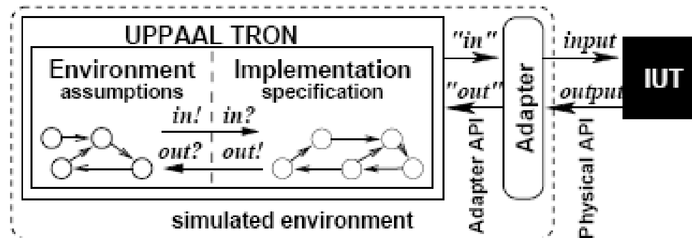


Figure 2.5: Testing using TRON

According to the relativized conformance testing theory, in UPPAAL, the model is specified into two disjunct parts, which is mentioned above, the system specification and the environment assumption part are synchronized with each other through channels which are declared in both parts. The model is executed using TRON, in order to get the "event by event" test cases.

The Adapter is an IUT specific hardware/software element that plays the role of a translator, to translate the abstract inputs into IUT recognizable inputs, and physical IUT's outputs into the SUT outputs.

And the IUT as shown in Figure 2.5, is considered as a black-box. This means that only inputs/outputs are deemed visible, and not the inner states.

Based on the theory used for our testing, the model of the environment assumption executes to offer an input, which will then be translated to an IUT input through Adapter. The Adapter will interpret the result offered by IUT into an TRON understandable output. Then the validity for the output being an legal output for the model of the system specification will be judged.

2.6 Case study of the Dimmer(smart lamp)

In order to have a good foundation for performing our industrial testing, we do a small case study of the smartlamp with UPPAAL TRON. The smartlamp example is proposed by the UPPAAL TRON group[17]. They have already interpreted the specification of the smartlamp system into UPPAAL model, called light controller. That model is considered as the SUT for this case study. They also offer the IUT for the testing, as well as the adapter. Both are written in JAVA.

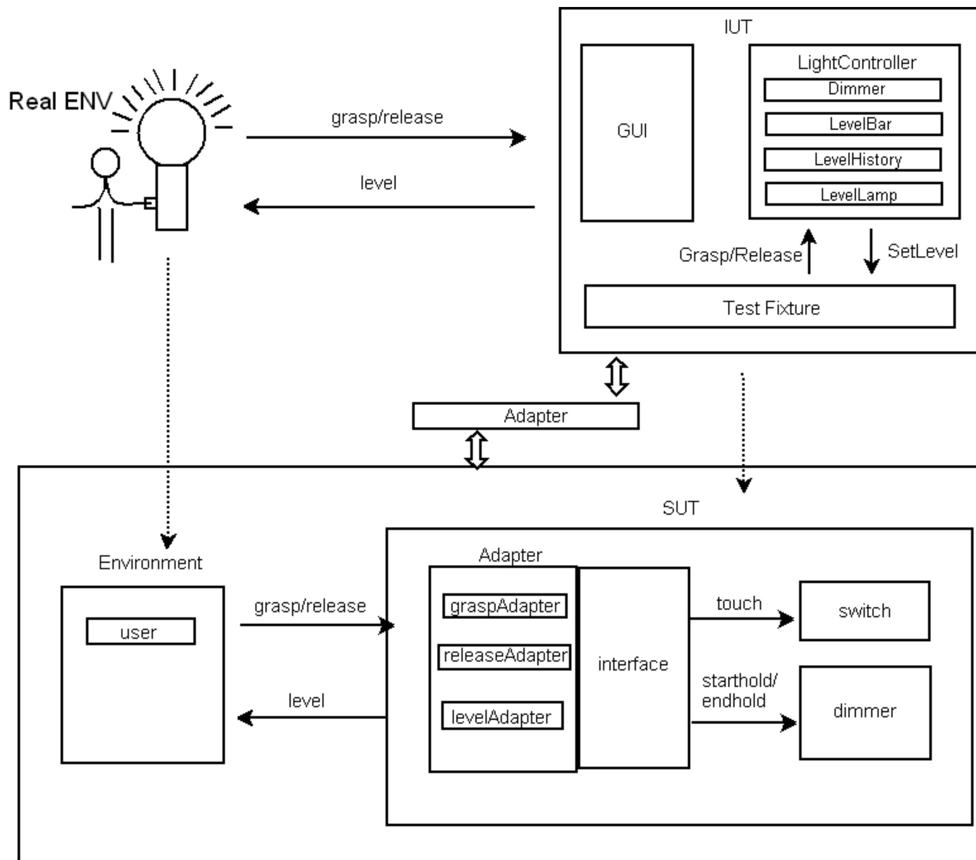


Figure 2.6: The complete testing setup of the smartlamp

Then we explain on explaining the SUT (in UPPAAL) and the IUT (in JAVA), with intention to have good understanding of the smartlamp system. The explanations are presented in the following sections. After that we apply the SUT on the testing tool TRON with its IUT, to perform the model-based online testing. We also create several mutants in the IUT, in order to prove that, TRON has the ability to detect the incongruent implementation in terms of the its system specification(SUT). This case study helps us to understand the rtioco relation, as well as the process of the model-based online testing using UPPAAL TRON.

2.6.1 Testing purpose & Testing setup

The smartlamp system is indeed a dimmer system. It is responsible for adjusting the brightness of the lamp. Its actions are conducted by the user. The user can turn on/off the lamp by quickly grasping/releasing the switch. If the lamp is in its **On** (**Off**) state, by holding the switch for certain among of time, its brightness will be increased (decreased) one level by one level until the brightest (darkest) level is reached. Then the brightness is adjusted in opposite direction. As it is mentioned that, the smartlamp is modeled in UPPAAL, and the light controller implementation is written using JAVA. The model-based online testing of the smartlamp using TRON is applied, checking whether the JAVA IUT conforms to the smartlamp specification.

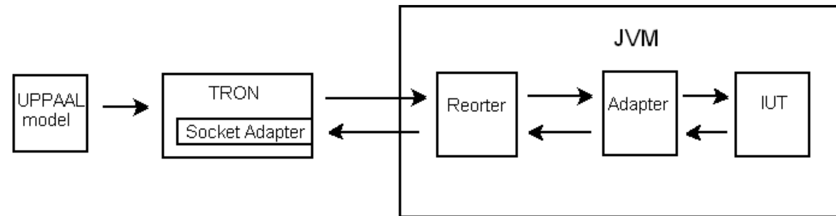


Figure 2.7: Signal Traveling

The complete testing setup is established, as shown in Figure2.6. In UPPAAL, the environment assumption and the system specification are modeled separately, as it is shown by the "Environment" part and "SUT" (system under test) part in Figure2.6. The abstract behavior models in these two parts commnicate through the **grasp/release** $\in Act_{in}$ and **level** $\in Act_{out}$.

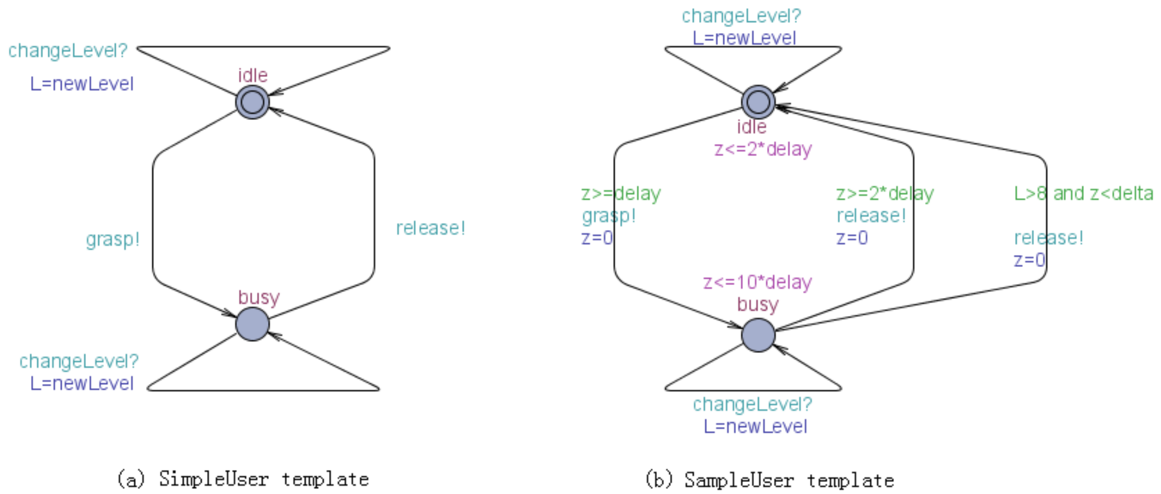
The Implementation (IUT) part is written in java code, mimicing the developed IUT. It interacts with its real environment **RealENV**, which is a real smartlamp. The user controls the level(status) of the smartlamp through java GUI. The **RealENV** and the **IUT** parts communicate with each other by means of the input signal **grasp/release** and the output signal **level**.

In order to "event by event" generate test cases, the UPPAAL models will then be executed using TRON. Each time only one test case will be translated to **IUT** through the **Socket Adapter**, it acts as an input for the **IUT**. Then the output of executing the input in the **IUT** will go through the **Adapter** reversely, as an "input" for TRON. And this "input" will be compared to the specification to determine its **pass(allowed)** or not.

Figure 2.7 shows the data stream of the testing setup. The UPPAAL model contains the system specification and the environment assumption. The model will then be applied in TRON to generate input signal for testing IUT. After the input signals are produced by TRON, it will be reported by the Reporter component to Adapter. Adapter will translate these signals into IUT understandable signal. When IUT sends out its results, Adapter will also interpret them for TRON, and TRON will get these output signals from Reporter.

2.6.2 Model description

The smartlamp, as well as its user are modeled as a network of the timed automata in UPPAAL. According to the testing purpose, the model is consisted of two parts: one simulates the environment, the other is interpreted from the system specification. This section will describe the model system template by template.



(a) SimpleUser template (b) SampleUser template

Figure 2.8: SampleUser template

Environment

- **SimpleUser** in Figure2.8(a) is the template simulating the simple user. It denotes **grasp/release** to the lamp system **SUT** through the adapter template with any time tolerance. In its **idle** and the **busy** location, it can receive input signal **changeLevel**, during this transition, it updates the value of the level **L** with new level **newLevel**.

  SampleUser is the template with two parameters **(chan &changeLevel, int &newLevel)** shown in Figure 2.8(b) . It is the abstraction of the behavior of the real environment. It performs **grasp/release** to the lamp system which is modeled as the **SUT**, and that two signal results in changing light level(status) of the lamp. Then it gets the **changeLevel** from the inner lamp system to denote the current light level(status) of the lamp.

  It starts from location **idle** with invariant $z <= 2*delay$, it may then receive an input **changeLevel** and update the lamp level **L** and stay in the same location. Otherwise when local clock **z** is greater than or equals to the value of the **delay**, it will enable the transition **grasp** which is synchronized with **graspAdapter** which is one of the instantiations of the template **Adapter** in Figure2.12 (a). And **SampleUser** reaches its **busy** location which is only satisfied when **z** does not exceed $10 * delay$. Here, it can do three actions: it can receive an input action **changeLevel**, and update the level L according to the **newLevel**; when the clock **z** is greater than or equal $2 * delay$, it will enabled the synchronized channel **release** and reassign the clock **z** to 0; or if level $L > 8$ and the clock $z < delta$, the synchronized transition **release** will be activated, and clock **z** will be updated to 0.

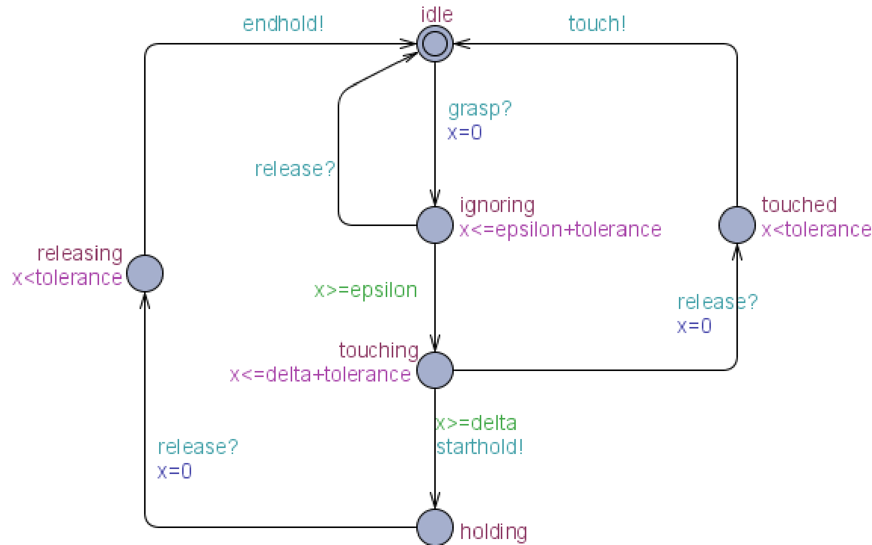


Figure 2.9: Interface template

SUT

Figure 2.9 models the interface between the adapter and the IUT. It accepts the **grasp/release** command which is translated by the adapter (in Figure 2.12 (a)) from the user template (see the Figure2.8). The duration between the arrival of the **grasp** and the **release** signal will be used to decide whether to switch the lamp by activating the switch template (see the Figure 2.10), or change brightness of the lamp using the dimmer template (see the Figure 2.11).

- The **Interface** template in Figure 2.9 starts from location **idle**, when it is grasped means the synchronized transition **grasp** is enabled, it sets the local clock **x** to 0, and reaches location **ignoring** with an invariant $x <= epsilon + tolerance$, which must be satisfied in order to validate the location. The **tolerance** here is used to make the **Interface** can distinguish among three subsequences. If the **release** signal arrives less than $epsilon + tolerance$ mtu, the template will be led to its initial location **idle**.

  Otherwise, if the guard $x >= epsilon$ is satisfied, the transition leads the system to the location **touching** will be enabled. Then if the **release** signal arrives with the local clock x being observed less than the $delta + tolerance$, it will transited to the location **touched** with invariant $t < tolerance$, after issuing **touch** signal to the **Switch** template, goes back to **idle**.

  Otherwise, in location **touching**, if the guard $x >= delta$ is satisfied, the channel **starthold** which is synchronized with **Dimmer** template in Figure 2.11 will be enabled. And the system goes to the location **holding**, and stays there until getting input signal **release**. When taking this **release** transition, it also reset local clock **x** to 0. Then it arrives at location named **releasing** with invariant $x < tolerance$. By issuing **endhold**,which is synchronized with **Dimmer** template, it goes back to **idle**.

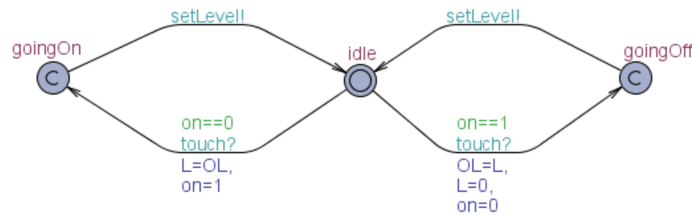


Figure 2.10: Switch template

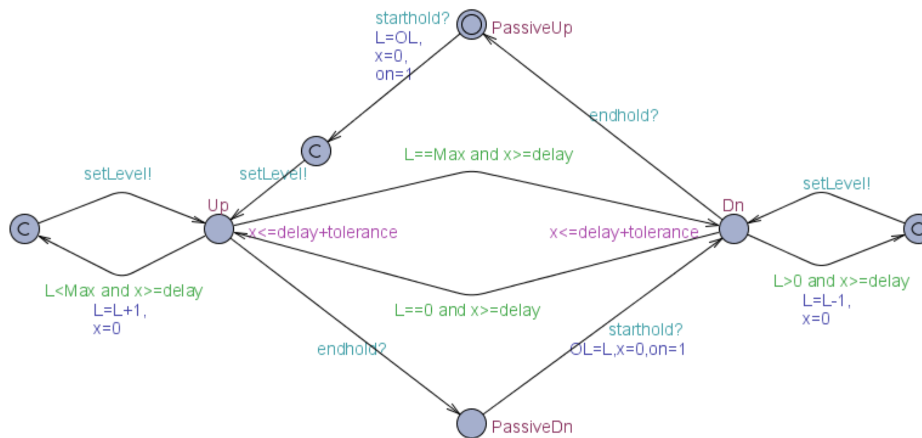


Figure 2.11: Dimmer template

- See Figure 2.10, **Switch** template is designed to mimic turning on/off the smartlamp. Its initial location **idle**. Starting from **idle**, it can receive an input signal **touch**. As it is mentioned, it is synchronized with the template **Interface**(see the Figure 2.9) by the channel **touch**. The **Switch** may perform two actions according to the variant **on**. The $on == 1$ means, the lamp is currently in its "on" phase. And the transition leading the system to the location **goingOff** will be enabled. During that transition, it does some updates: $OL = L$**,** $L = 0$**,** $on = 0$. Then it immediately issues synchronized channel **setLevel**, and goes back to the initial location **idle**.

  However, if in **idle** location, it detects that the **on** equals 0, which means it is in lamp's "off" phase. It will then execute those updates $L = OL$**,** $on = 1$, and goes to the location **goingOn**. After this, it enables the synchronized transition **setLevel**, and goes back to **idle** without any delay.

- The Figure 2.11 shows the template named **Dimmer**, which is used to adjust the brightness of the lamp. It starts from the **PassiveUp**. When the transition **Starthold** is enabled, it does some updates: $L = OL$ which is used to restore the last recorded brightness level of the lamp, and reset the local clock **x**, as well as assigns the **on** with "1". After that it reaches a committed location, and immediately transit to the location **Up** through the synchronized channel **setLevel**.

  In the location **Up**, it has three choices: if the guard $L < Max$ **and** $x >= delay$ is satisfied, it will enable the transition which does increase the current level **L** using formula

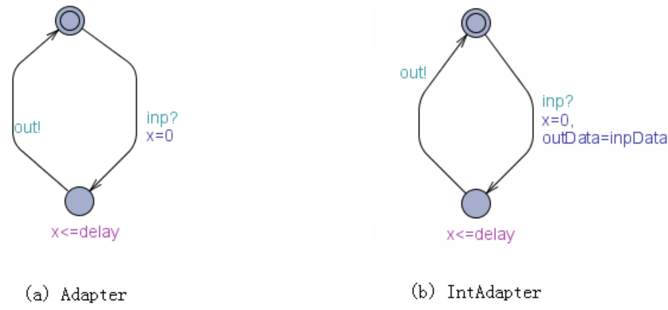


Figure 2.12: Adapter template

$L = L + 1$, and resets the local clock **x** to 0. Then it immediately sends the result to **levelAdapter** which is an instance of the **IntAdapter** in Figure 2.12 (b). However, if $L == Max$ **and** $x >= delay$, then **Up** will take the transition, leading to location **Dn**. Otherwise, when it receives the synchronized signal **endhold**, it is led to **PassiveDn** from location Up, and waits for being synchronized with template **Interface** through channel **starthold**. During the **starthold** transition, it restores the current level by means of $OL = L$, updates clock x to 0, ensures the status of the lamp is "on" through $on = 1$. And then, it is led to **Dn** from the location **PassiveDn**.

In location **Dn**, it may go further through three transitions. If the guard $L > 0$ **and** $x >= delay$ is satisfied, it will decrease the bright level of the lamp by means of $L = L - 1$, and reassigns 0 to clock **x**. Then it immediately go through a committed location, and synchronizes with the **levelAdapter** template with intention to send out the level of the lamp to **user**. If $L == 0$ **and** $x >= delay$ is satisfied, it will directly transit from **Dn** to the location **Up**, waiting for being up. In the location **Dn**, it may also wait for being synchronized with template **Interface** through channel **endhold**, and goes back to the initial location **PassiveUp**.

2.6.3 Hints in Adapter Modeling

There are two automata in Figure 2.12(a) and Figure 2.12(b), simulate the function of the adapter. As it is mentioned in Figure 2.7, the input/output signal travels through several steps in both sides: TRON decides to offer input; TRON(IUT) sends input(output); Adapter translates input(output); IUT(TRON) sense input(output).Delay exists in between each steps. The idea of modeling the adapter is to model the communication latency caused by the signal traveling, and the scheduling latency caused by OS. The invariant **delay** in both templates implies the latency.

2.6.4 IUT

The IUT is written in JAVA. The java program of IUT contains several components(classes), mainly are light controller, and the socket adapter. The following will explain you some main components:

```
//−−−−−−−−−−some code in the LightController.java−−−−−−−−−−−−−
        class LightController
        {
        ...
        run()
        {
        ...
        case Idle:
        try { cond.await(); }
        catch( InterruptedException e) { alive = false; }
        break;
        case Grasped:
        try { cond.await(epsilon, TimeUnit.MILLISECONDS); }
        catch( InterruptedException e) { alive = false; }
        finally {
            if (location == Loc.Grasped) location = Loc.Alert;
        }
        break;
        case Alert:
        try { cond.await(delta−epsilon, TimeUnit.MILLISECONDS); }
        catch (InterruptedException e) { alive = false; }
        finally {
            if (location == Loc.Alert) {
            location = Loc.Hold;
            dimmer.handleStartHold(startTime+delta);
            }
        }
        break;
        case Hold:
        try { cond.await(); }
        catch( InterruptedException e) { alive = false; }
        break;
        ...
        }
        ...
        }
```

LightController respectively carries out two basic actions: grasp and release through method **handleGrasp** and **handleRelease**. Those two methods can be called when those two cases occur: it receives the instruction signal from the mouse through **mousePressed** and **mouseReleased** methods; or when the grasp/release signal is received from the TextFixture, see Figure 2.6.

The **LightController** is the main component of the IUT. It carries out the same function

as the interface as shown in Figure 2.9, translating the outer actions to the lamp. Its **run** method mainly tried to deal with 4 different states(locations). I.e., it defines the transferring rule among those **LightController** locations. When it is in location **Idle** or **Hold**, it can wait as long as it wants. In location **Grasped**, it can delay for no more than **epsilon** time units before being transferred to location **Alert**. However, after staying in **Alert** for less than $delta - epsilon$, the state is changed to **Hold**, and it will also call the **handleStartHold** method of the class **Dimmer**.

Dimmer is used to adjust the bright level or status of the lamp by means of transforming among the four states: **UpPassive**, **UpActive**, **DnPassive**, **DnActive**. During its life cycle, it can wait at its **UpPassive** state until receive **start hold** signal which leads it to the **UpActive** state by means of the method **handleStartHold(long startHold)**. it can stay at the **UpActive** for some admitted delay, and it can perform increasing the current level of the lamp by one through **setLevel(level+1)**, however, if it is already in its highest level, and should be transformed to **DnActive** state; when it receives the **end hold** signal, it will be in **DnPassive** state through **handleEndHold()** method.

```
37 //−−−−−−−−−−−some code in handleTouch() method of Dimmer.java−−−−−−−−−−−−−−−−−
38 class Dimmer
39  {
40   ...
41  handleTouch()
42  {
43   ...
44  switch ( lightState ) {
45          case lightOff :
46          setLevel (oldLevel);
47           lightState =lightOn;
48          break;
49          case lightOn:
50          oldLevel = level ;
51          setLevel (0);
52           lightState  = lightOff ;
53          break;
54          }
55      ... }
56      ... }
57
58 //−−−−−−−−−−−some code in handleStartHold() method of Dimmer.java−−−−−−−−−−−−−−
59 class Dimmer
60 {
61  ...
62 handleStartHold (long startHold )
63 {
64  ...
65 switch
66 ( location ) {
67     case UpPassive:
68          setLevel (oldLevel);
69           lightState  = lightOn;
70           location  = Loc.UpActive;
```

```
71            cond. signalAll () ;
72            break;
73        case DnPassive:
74            //        setLevel ( oldLevel );
75            oldLevel = level ;
76             lightState  = lightOn;
77             location  = Loc.DnActive;
78            cond. signalAll () ;
79            break;
80        default:
81            System.out. println ("Dimmer:␣cannot␣accept␣startHold␣in␣"+
82                        location );
83           }
84           ...}
85           ...}
86
87  //−−−−−−−−−−−−some code in handleEndHold() method of Dimmer.java−−−−−−
88  class Dimmer
89  {
90   ...
91  handleEndHold()
92  {
93   ...
94  switch( location ){
95        case UpActive:
96             location  = Loc.DnPassive;
97            cond. signalAll () ;
98            break;
99        case DnActive:
100            location  = Loc.UpPassive;
101           cond. signalAll () ;
102           break;
103       default:
104           System.out. println ("Dimmer:␣cannot␣accept␣endHold␣in␣"+location);
105          }
106          ...}
107          ...}
```

When in the state **DnPassive**, it can wait as long as it wants until getting **start hold** again, which will activate it to go to the **DnActive** state. It can remain in the **DnActive** within the permitted delay. In contrast to the **UpActive** state, this time **setLevel(level-1)** is called, it means to decrease the level of the lamp by one. But when the level is equal 0, the state will be changed to **UpActive**. Otherwise, if it is issued **end hold**, the state will be modified to **UpPassive** by means of the method **handleEndHold()**.

LevelListener is an interface with intention to react to the change of the level according to the **Dimmer**. The **LevelBar**, **LevelHistory**, **LevelLamp** are three classes implements **LevelListener**.

LevelBar is used to intuitively demonstrate the current level of the lamp. As it is shown in the Figure2.13, the graph of the level bar is consisted of 10 small rectangles with

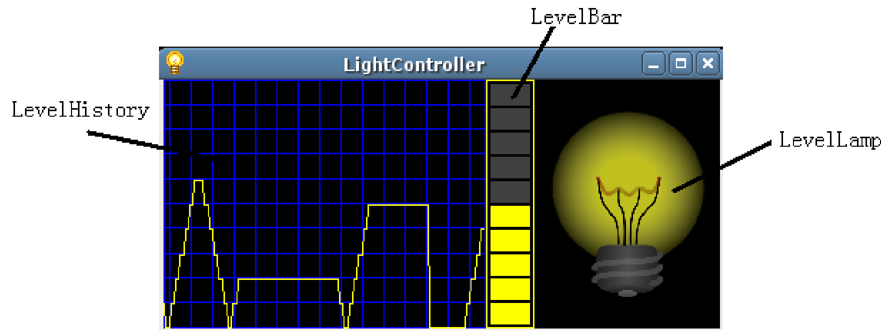


Figure 2.13: the GUI of the lamp

background color darkGray by means of the method **setColor(Color.darkGray)**. From the bottom of the bar, the number of the rectangle being turned to yellow corresponds to the level of the lamp.

LevelLamp shown in Figure2.13, works with the same goal as the **LeveBar**. The color of the bulb will be changed in terms of the level of the lamp. The color of each level is calculated by means of the method **Color(**$float\ a$**,** $float\ b$**,** $float\ c$**)**.

LevelHistory tries to clarify how was the lamp level going on. The current level of the lamp is represented through **drawLine(**$int\ x1$**,** $int\ y1$**,** $int\ x2$**,** $int\ y2$**)** with different value of the parameters. The difference between **LevelHistory** and the other two level graphical tools mentioned before, is that **LevelHistory** stores the trace of the level.

TestFixture does the basic configuration for the test. It instantiates the basic input/output interface through **configure(Reporter reporter)** method.

Adapter and Reporter The **Adapter** provides an interface to IUT for the tester. It contains two methods **configure()** and the **perform()**. Method **configure()** is called by **Reporter** when the connection with the TRON is set up. During this method, it will configure the input/output interface through those methods: **addInput()**, **addOutput()**, **addVarToInput() addVarToOutput()**, **setTimeUnit()**, **setTimeOut()**. And when it receives an input from the tester it will call method **perform()**.

Reporter maintains one connection to tester. It receives the inputs and applies them to the Adapter, or it reports the outputs from the IUT to TRON. Every connections goes through two phases: initialization and testing activation. As it is mentioned, during the initialization, it will configure the input/output interface by optionally calling the following methods:

addInput() is used to adds the input channel to the testing interface, and returns a positive channel ID; **addOutput()** is used to add the output channel to the testing interface, and returns ID of the channel; **addVarToInput()** is to bind a model variable to the specified input channel, the value of the variable is attached as parameters to an input action on that channel; **addVarToOutput()** binds a model variable to the specified output channel, the value of the variable is attached as parameters to an output action on that channel;

Error detection capability						
mutant	Input action			Duration(mtu)		
	Min	Avg	Max	Min	Avg	Max
M1	20	94	246	1230	10538	29829
M2	5	56	181	448	4629	12836
M3	3	5	9	161	512	1189
M4	1	6	18	92	175	381
M5	7	14	47	385	1838	6082
M6	3	9	27	592	1002	1881
M7	7	17	33	448	1051	2376
M0	239	7073	9937	29684	826773	1000000

Figure 2.14: Error detection capability

setTimeOut() sets the total amount of time units used for testing; **setTimeUnit()** sets the length of the model time units in the real world time units . Because the nature requirement of TRON, time units and the time out need to be set.

2.6.5 Experiments

This section presents the results of a small experiment using IUT. The purpose of this experiment is to demonstrate the feasibility of TRON in terms of the error detection capability.

In this experiment, we choose SimpleUser in Figure 2.8(b) as the environment model. And we also have created a number of inaccurate mutations(M1...7, where M1, M2 are given by the developer) based on the assumed correct implementation(M0).

M1 : After touch the level is still 0 (instead for recover the old level).

M2 : When transferring from DnActive to UpActive, the extra delay (DoDelay) is allowed (instead of forbidden).

M3 : When being issued EndHold in location UpActive, it will transfer to UpPassive (instead of DnPassive).

M4 : The initial location for the Dimmer is DnPassive (instead of UpPassive).

M5 : After startHold, the state is transferred from UpPassive to DnActive directly (instead of going through the location UpActive).

M6 : After StartHold, the state is transferred from DnPassive to UpActive directly (instead of going through the location UpActive).

M7 : After being issued EndHold in location DnActive, it will transfer to DnPassive (instead of UnPassive).

The experiments uses simulated clock progressing. Each mutant is tested 10 times with 1000000mtu as its upper bound of the time limit. It supposes that all running with M1...7 fail, all running with M0 pass. The results are presented in Figure2.14.

The experiments show that the mutants with error assumptions can be detected very fast. It used less than 250 input actions and less than 30000mtu. However, results of executing with M0 were unexpected. A few of the executions with M0 fail with unknown reasons. But the developer does not find the same problem, when performing the same experiment under the Linux OS.

Chapter 3

Case study: the feeding system

3.1 System Overview

We will do a case study with the SKOV feeding system(Viper). Briefly speaking, the main task of this case study is to apply the Model-based online testing technique on the Viper testing. Based on the product specification[18] and its executable software from SKOV, we will model the Viper system using UPPAAL.

The subsections are organized as follows. First, we sketch out the overall process of the Viper system. And we will also briefly present the overall structure of the expected model for the system. After that, we will start describing the modeling detail. We use three independent chapters to present three sub-models regarding three main functions of the feeding system. Each chapter for each sub-model includes not only the overview and the structure of each model system, but also the UPPAAL modeling process, the model checking, and manual testing for each sub-system, and some discussions.

3.2 System functionality decomposition

The complete process of the Viper feeding system is shown in Figure 3.1. When the feed bin in the livestock buildings indicates not enough food, this product is applied to automaticaly control the feeding of the animal in the livestock buildings. As it is mentioned, Viper carries out this function by means of correctly activating the feed storage and the transport system.

Figure 3.1 shows the abstract feeding process, in the case of two vipers sharing one **Dol99B**. It is mainly consisted of the **Dol99B**(Figure 3.1, no.1), **Silo Auger**(Figure 3.1, no. 2), **distribution shutter**(Figure 3.1, no. 3), and the **feed bin**(Figure 3.1, no.4,5), Dol7, and the **Vipers** marked 1,2 respectively. Figure 3.3 shows the realization of the part1 in Figure 3.1

Complete process It is shown in Figure 3.1. When there is feed demand in either **feed bin** (Figure 3.1 no.4, 5), it will send the demand feed signal to its viper. The two vipers notify the feed demand to each other through the resistor, called **Dol7**. And one of the viper is

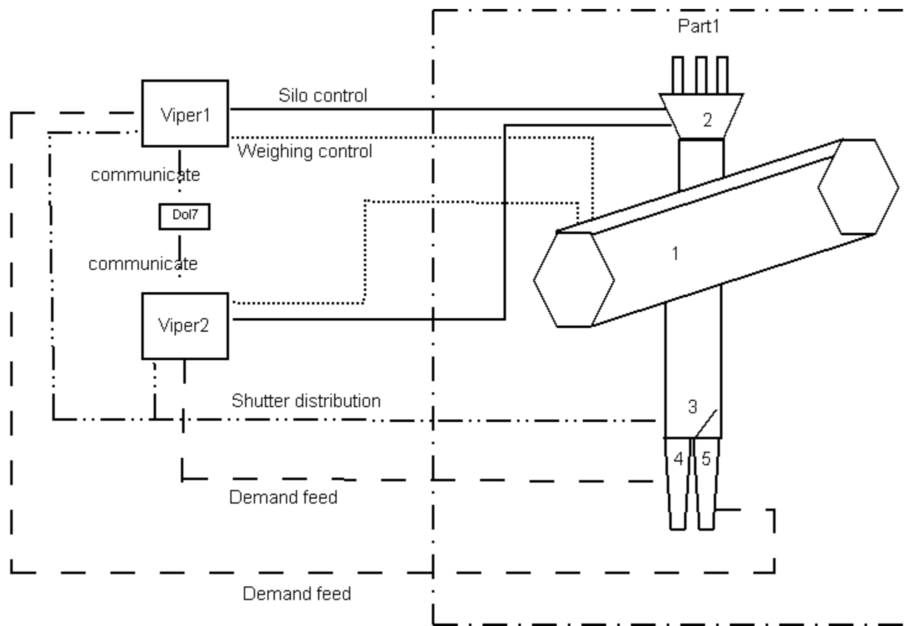


Figure 3.1: Workflow of the Viper Feeding System

decided to be the controlling viper. If there is feed demand in either bin, the controller will activate the **silo auger** (Figure 3.1 no. 2) with intention to push food into the **drum**(inside the **Dol99B**).

Using its **weighing function**(see section 3.2.2), the controller conduct the **Dol99B** to weigh the acceptable amount of feed for the **drum** each time. When the **drum** reaches that desired amount, the **silo augers** will be stopped. Then **Viper** stimulates the **drum**. **Drum** rolls, transporting the feed to the **feed bin** who requires the feed using **distribution shutter**.

Silo Auger It is shown in Figure 3.1 no. 2. It is used to transport the feed. Two silo augers will be used, in the case of two vipers sharing the **Dol99B**. The **silo auger1(2)** will be started, when the controller fulfills the feed demand from the **Viper1(2)**. And the feed will be transported into the drum in the **Dol99B**. It will not be stopped until the **drum** reaches its desired amount of weight.

Dol99B The actual **Dol99B**(Figure 3.1 no.1) system is shown in Figure 3.2. The chief components in **Dol99B** are weighing **drum**(Figure 3.2 no. 5), **stop plate**(Figure 3.2 no.8), **distribution shutter**(Figure 3.2 no.12). When it is stimulated by the controlling viper, the drum will start to weigh the demand feed(see section 3.2.2). When the desired weight has been transported into the drum, it starts rotating to empty it. **Stop plate** will be sensed by the sensor above the **drum** (Figure 3.4). The arrival of the **stop plate** indicates, the drum is emptied. The feed in the drum has been sent to the bin by means of the **distribution shutter**.

- **Roll Drum** (or weighing drum, in Figure 3.2 no.5) is a container inside the **Dol99B**(Figure 3.1, no.1). It has a **top plate**(Figure 3.4) on the interface of it. The **sensor** on top of the drum is used to sense the position of the **top plate**. Drum may be in its top plate

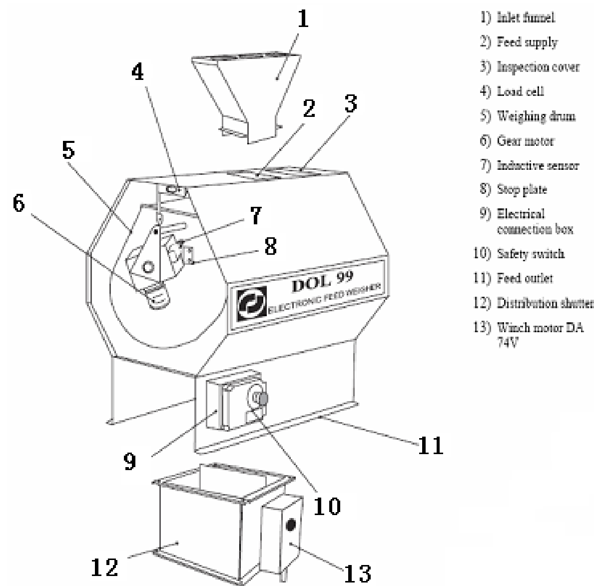


Figure 3.2: The Dol99B component [18]

or rotating. When the **top plate** is precisely under the **sensor**, **sensor** presents the highest signal than the **top plate** in any other place.

When in the top position, the drum is expected to be empty until being filled by the feed from the silo auger. After the desire amount of feed is pushed into the **drum**, it will be activated by **Viper**, and start to rotate one cycle. During that first half cycle, the position of **top plate** is from top place to the bottom place where the **sensor** can not feel it at all, so that the high to low signal will be indicated by the **inductive signal**. Then **top plate** will be from the bottom with the lowest **sensor** signal to the top position with the highest signal again, **inductive signal** will also indicate the low to high state. With this cycle, the **drum** will be empty again.

In order to perform the weighing correctly, the user must order the calibrate of the drum periodically. The weighing and the reference signals(see Figure 3.4 Vweigh and Vref) are used to calculated the current weight of the drum. Those two signals can be sensed by the **spring sensor** in Figure 3.4. Until one weigh signal above 5 volt is stable for 10 seconds, both the weigh signal and reference signal will be recorded, and the maximum weight will be calculated. The weighing signal and the reference signal will be checked, unless the first one weigh signal below 3 volt is found, then it will recorded (together with the reference signal) to calculated the minimum weight. The deviation of the maximum and the minimum weight will be regarded as the result of the calibration.

Feed bin It is also called vessel(see Figure3.3 no.6), shown in Figure 3.3 no.6. Each feed bin is responsible for one livestock building. The feed will transported to the animals from the feed bin. The amount of the feed in the bin is checked by the Viper.

Dol7 It is shown in Figure 3.1. It is the device control the communication between two Vipers. If one of the vipers has feed demand, the demand signal will be sent to the Dol7. It

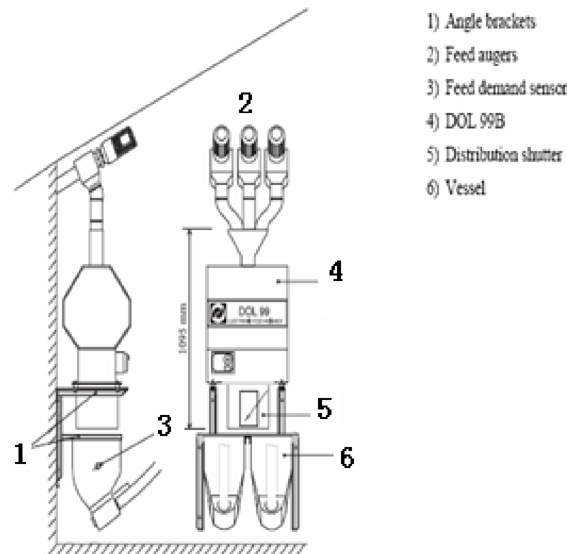


Figure 3.3: Feeding system controlled by Viper [18]

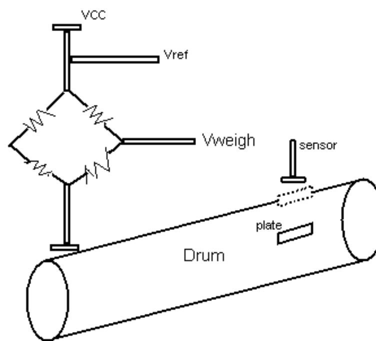


Figure 3.4: The drum component

helps to notify another viper about this demand, and also validate the holding status of the ownership of the shutter.

Viper It is used to conduct the actual feeding action of the **Dol99B**. Each Viper corresponds to one **feed bin** or **vessel** (see Figure 3.3 no. 6). In the case of two vipers sharing the **Dol99B**, one of them will be determined to be the controller. It controls the feeding system to perform the feed demand function(see the section 3.2.1), weighing function (see the section 3.2.2), and the calibration function (see the section 3.2.3).

3.2.1 Demand feed function

When two vipers share the **Dol99B** as it is shown in Figure 3.1, the sharing is controlled by the controlling viper. We assume that the **Viper1** is the controller, the **Viper2** is the not controlling viper. The controlling viper is responsible for the distribution of the ownership of the shutter, and adjust the shutter to its owner. The two vipers communicate with each other through the resistor **Dol7**(see Figure 3.1).

If either viper has the feed demand, it will try to ask the **Viper1** for the ownership of the shutter, and notify the other viper of the feed demand using the **Dol7**. There are three level of voltage in the **Dol7** represents whether the holding ownership is legal or not. The initial voltage of the DOL7 must be less than 2 v, indicating that neither vipers holding the ownership. If it keeps staying in 5 v at least 10 sec, means that only one of the vipers gets the ownership of the shutter successfully. That is the legal status. Otherwise, if it is measured 6.7 v, means two vipers get the ownership simultaneously. Then, both of them needs to release the ownership.

After one of the vipers gains the ownership successfully, the **Viper1** will control to distribute the shutter to its owner. Then, the **Viper1** will also checks for validity of the position. That is, if the shutter can not reach the position of the one holding the ownership, the shutter alarm will be generated.

3.2.2 Weighing function

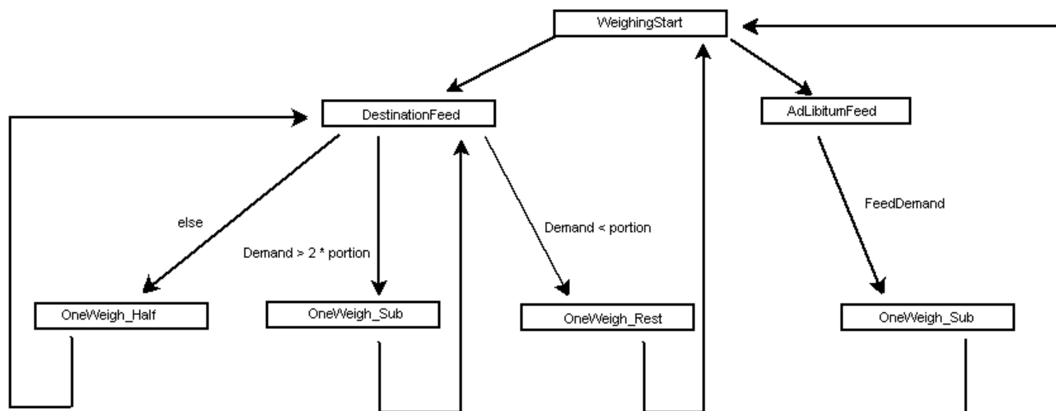


Figure 3.5: The weighing process

The **Weighing** function of **Dol99B** is used to control weighing request amount to satisfy the feed demand. The complete process for carrying out the **Weighing** function is shown in Figure 3.5.

There are two kinds of feeding: **destination** feeding and the **Ad Libitum** (Ad) Feeding. When **Ad** feeding is chosen, it will satisfy the **feed bin** with one fill amount(fill size or one portion) whenever there is a feed demand. If the **destination** feeding is chosen, it can not directly weigh the amount for the **feed bin**. It may need one or more weighing cycles, which mainly depend on the feed demand amount. The **destination** feeding will follow the below strategy.

If the feed demand is below one fill size, it will directly weigh the demand for the **feed bin** by means of the sub-process called **OneWeighing_Rest**(see following description **"OneWeigh-**

ing"). If the feed demand is above the 2 times of the fill size, it will first call **OneWeighing_Sub** to fulfill the one fill size amount. So the rest of the demand will be the former total feed demand minus the one fill size. Otherwise, if the feed demand is between one portion and two portions, it will **OneWeighing_Half** the half of the demand. The rest should be another half of the demand. So forth, until the former total demand is satisfied. And it starts to wait for another feed demand.

- **OneWeighing**

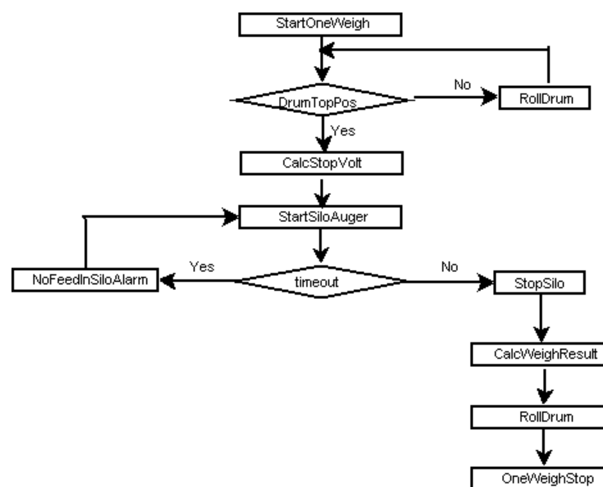


Figure 3.6: One sub weighing process

Figure 3.6 shows sub-process of the **Weighing** named **OneWeighing**. Each time, when this function is activated, it will first check the position of the **drum**(Figure 3.4). If the sensor can not feel the top plate, it will start to rotate the **drum**(see the description **"RollDrum"**). Otherwise, further steps can be carried out.

First, the volt to decide when the sub-weighing should be finished, which is so called **stop volt**, will be calculated. And the **silo auger** will be starts. If the silo auger is running for more than 5 minutes, it will active the alarm **NoFeedInSilo** alarm and the **silo auger** is stopped as well. Otherwise, it waits until the silo finishing the feed transporting. Then the weight of the feed will be calculated, and the **drum** will rolled

- **RollDrum**

RollDrum function is used to transport the weighted feed to the **feed bin**. Its workflow is shown in Figure 3.7. At first, it checks whether the roll is allowed or not, which means if the shutter is not in the desired position, it will be waited. As long as the **drum** is allowed to be rolled, the **drum** motor will be started.

If the **drum** is started within the required time(time out does not happen), the inductive signal will be checked. If the signal indicates from low to high, means the **drum is rolling**(see Figure 3.4). Until the signal is stable, the **drum** will be stopped, and the **RollDrum** is stopped as well.

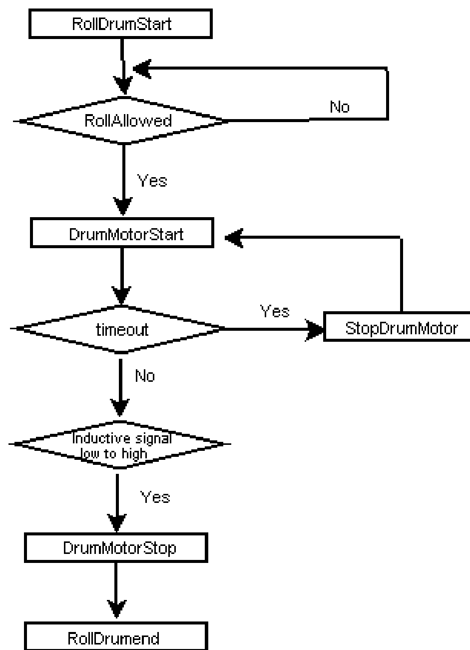


Figure 3.7: Drum roll process

3.2.3 Calibration function

Calibration is used to gain the exact value of the fill size of the **drum**. It can not be carried out simultaneously with the **weighing** function. The main process is shown in Figure 3.8. When **Dol99B** receives the **calibration** requirement from the user, it checks the position of the top plate in the **drum**(see Figure 3.4) . If the plate can not be sensed by the sensor, the drum will be **rolled**.

As soon as the top plate can be felt, the process called "weigh with weight" is started, the maximum weight of the drum will be calculated. Each time when the weight signal which is bigger than 5 volt is sensed, and lasts for 10 seconds, both the weight signal and the reference signal(see Figure 3.4 Vweigh and Vref) will be saved. When it gets a minimum value and the weight signal is below 5 volt, it will start to weigh without weight(weighing with no weight). That is the weight signal is checked. If Weight signals below 5 volt and lasts 10 seconds, the weight signal and the reference signal will be saved. Once a value below 3 volt is gained, the calibration value will be started to calculated. And the deviation to the old calibration value will also be calculated.

3.3 System structure

Based on the testing purpose of checking the correctly execution of the **viper**, and the testing tool that we will use, we plan to separate and model the components of the feeding system into

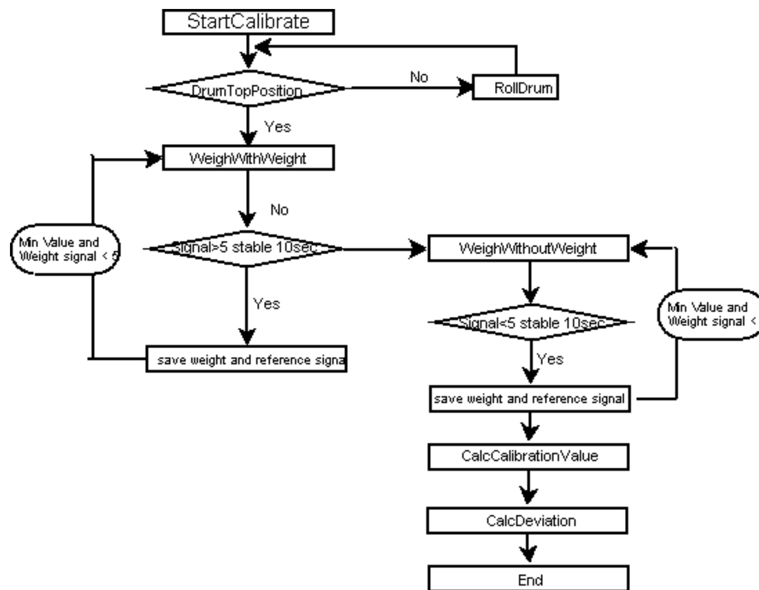


Figure 3.8: Calibration process

the **system under test(SUT)** part and the **Enviroment(ENV)** part. Figure 3.9 is the overview of the the main components and the communications between them. They are abstracted from the specification of the **viper**.

As is shown in Figure 3.9, **Viper1** is the controlling viper, and it is in the SUT(system under test) during the modeling and the testing. Other main components belongs to the ENV(environment) part. The controlling viper will receive the "calibrate" requirement from the **User** object. **Viper1** can also communicate with the **Viper2** about the feed demand. Controlling viper **Viper1** is responsible for the "start" of the **Dol99B**, **SiloAuger**, **Drum**, **Dol7**. And after they finish their task, they will send **Viper1** the "stop" signal.

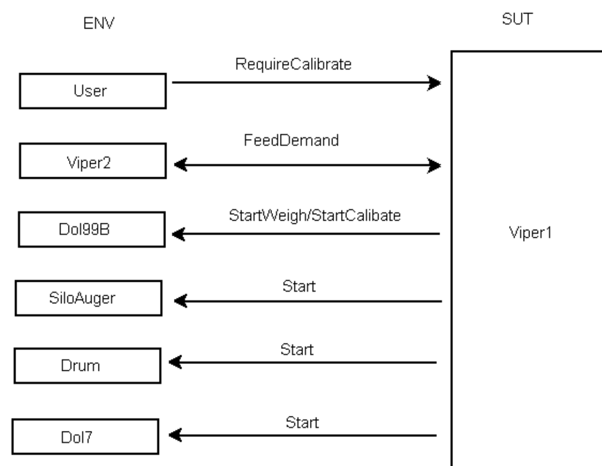


Figure 3.9: The overall cooperation between components

The real model is consisted of 3 sub-models according to its specified three main functions. Each sub-model is modeled into the ENV and the SUT parts. The structure and the detail explanation for each sub-model will be presented in later chapters.

3.4 Emulate Labview system

The feeding system in UPPAAL will later be used to perform the manually testing with the implementation(IUT). The IUT is a small executable emulate feeding system named **Dub99**. It is produced using LabView. The following section will show you the main components of the **Dub99**, that are considered useful for the manual testing.

Dol99B

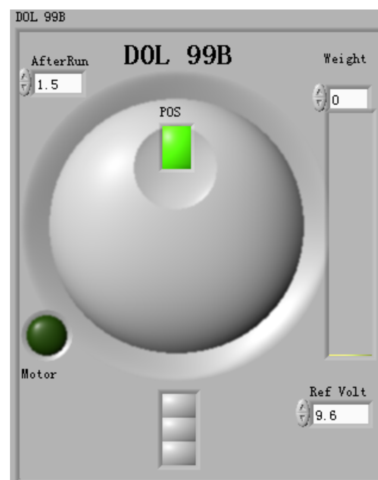


Figure 3.10: The Dol99B component

Figure 3.10 displays the components inside the **Dol99B**. They are **POS**, **drum**, **Motor**, **Weight**, **Ref Volt** box. The big rotundity outside the **POS** box represents the **drum**, the small rotundity inside the big one is the top plate of the **drum**. The moving of the small rotundity represents the rotating of the drum. The position of the top plate in Figure 3.10, indicates the **drum** is in its top position. The **drum** is started when the drum **Motor** is on. The dark green of the **Motor** in Figure 3.10, indicates the off status of the **Motor**. And the current weight and the reference voltage will be presented in the **Weight** and the **Ref Volt** boxes respectively.

Silo auger

Figure 3.11: The silo auger component

Figure 3.11 are the **silo auger1** and the **silo auger2.** The feed for the **Viper1(2)** comes from the **silo auger1(2)**. Either auger starts importing the feed, its color box will be turned from the dark green to (light) green. The **AfterRun** box displays the default times, that the silo auger may be activated during the **Max use time** (see Figure 3.16, Max Use Time IM box).

Dol7

Figure 3.12: The Dol7 component

The **Dol7** is the resistor in the IUT, it has three color boxes. They are **box1**, **box2**, **Out Volt** box. The **box1(2)** represents the ownership holding status of **viper1(2)**. I.e. if either viper gains the shutter ownership, its box will be turned green, otherwise, it is dark(see the current status in Figure 3.12). The **Out Volt** may show three level of the voltage. The initial value is "0". If one of the viper box turn green, the voltage box will display "5". If both of the viper boxes are green, the voltage will be "6.66666".

Flap(shutter)

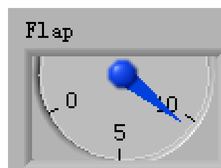


Figure 3.13: The flap(shutter) component

The **Flap** box in Figure 3.13, indicates the position of the shutter. If the blue pointer direct to the region from "0" to "1", the shutter is in the position of the **Viper1**. If it points to the "9" to "10" region, it is in the position of the **Viper2**. Otherwise, the shutter is moving between between two vipers.

Feed bin

Figure 3.14 shows the feed bin components. The light box indicates that there is feed demand in the viper. If the bin is filled, that color box will turn to dark green. The **FeedOutRate** boxes display the speed that, the feed is transited from the feed bins to the livestock buildings. And the "up" and "down" of the **Feed_Demand** bars represents that, there is feed demand or no feed demand respectively.

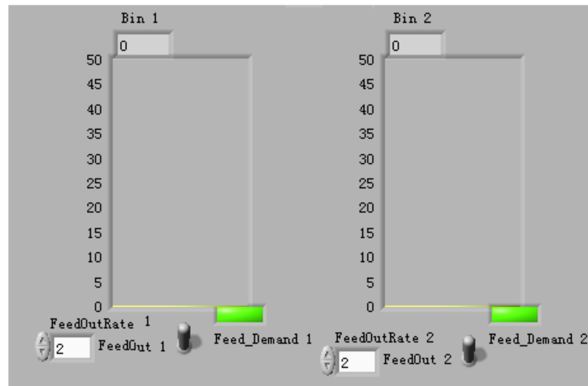


Figure 3.14: The feed bin component

Alarm

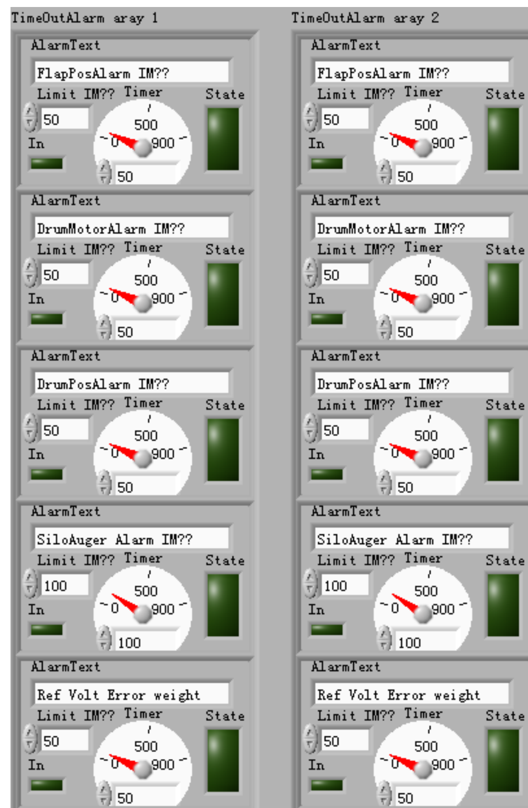


Figure 3.15: The alarm component

The alarm system contains many sub-components. Each alarm in it represents one specific error with regard to its expected status. E.g., the **FlapPosAlarm** box represents the alarm that, monitoring the flap(shutter) position. It is expected that, when the ownership of the shutter is determined, then the shutter can move to the position of its owner with certain amount of time(50 is set in the **Dub99**). Otherwise, that alarm will be activated. The color box **State** will be turned red from dark green.

Vipers

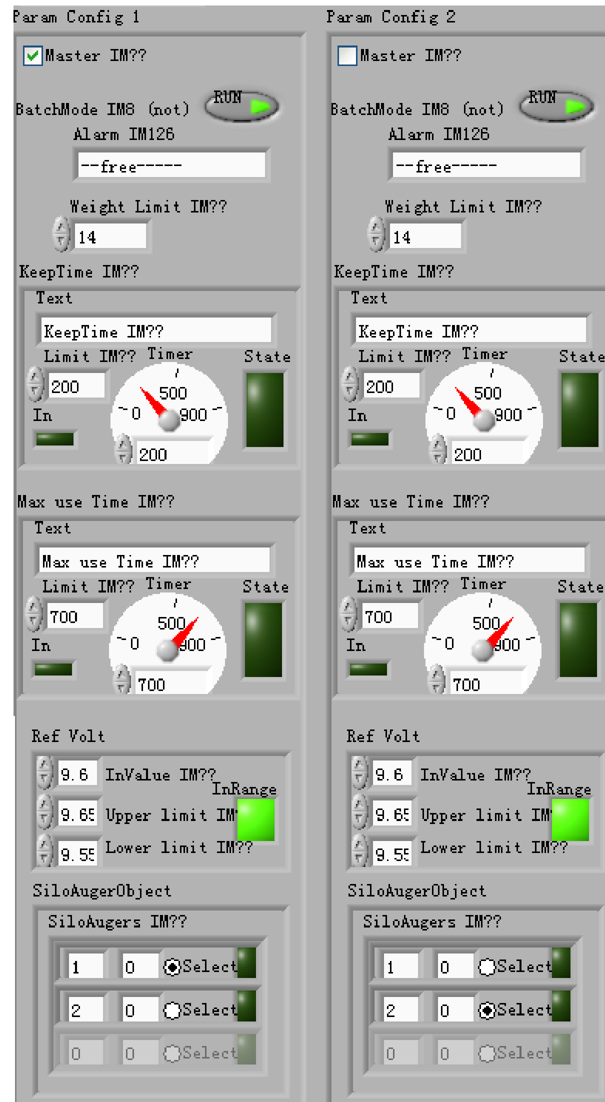


Figure 3.16: The Viper1 and the Viper2 components

The vipers are shown in Figure 3.16. The master box of the **Viper1** is clicked, meaning the **Viper1** is the controlling viper. That conforms to our model systems. And the sub-components in the vipers, are the system items required to be controlled. E.g., the **KeepTime** box is responsible for monitoring the viper who holds the ownership of the shutter. That is conforms to the specification [18] that, the owner of the shutter can not keep the ownership more than certain time.

3.5 Summary

In this chapter, we introduce you the overview of the SKOV feeding system, as well as the system structure. Then we decompose the system into three sub-systems according to its three

main functions. In the later chapters, the UPPAAL models will be established based on those three functionalities respectively. And we also explain the emulate feeding system(Dub99) established in LabView. That, not only helps us to capture the behavior of the feeding system specification, but also will be used as the emulate IUT for our testing purpose.

Chapter 4

Model1 - Demand feed

4.1 Model purpose and structure

When building the model, one of the requirements is to satisfy the control and communication between two Vipers (**Viper1** and **Viper2**) that are sharing the same **Dol99B**. So the control and communication include granting the ownership of the shutter to the viper who has the feed demand and adjusting the shutter to the position of its owner. Another purpose of the control and communication is to check whether the shutter position is in its owner, if it is not, it will generate an alarm.

In our system we assume that **Viper1** is the controlling viper. **Viper2** is the not controlling viper. The model1 system contains 6 templates. They are displayed in Figure 4.2, 4.3, 4.4, 4.5, 4.6, 4.7 respectively. Either Viper1(see Figure 4.4) or Viper2(see Figure 4.5) has feed demand, it will inform the other one through the resistor **Dol7** in Figure 4.3. The resistor will active the control function of the **Viper1**. It is responsible for granting the ownership of the shutter, and distributing the shutter to the position of its owner. The **Viper1** is also used to check if the shutter position matches with the owner of the shutter. If shutter is in the position of its owner, it will pass. Otherwise, the system will generate an alarm.

Figure 4.1 shows the structure of the model1. It contains the **ENV**(environment) part and the **SUT**(system under test) part. The **ENV** contains 5 components, and the **SUT** only has the **Viper1** in it. The whole model1 system is consisted of 12 discrete integer variables and 16 channels. In Figure 4.1, the main templates and input/output signals of the model1 are drawn. The functions of each channels are listed as following:

Bin1Demand/Bin2Demand When **Bin1**(**Bin2**) has feed demand, the **feed demand** template will issue **Bin1Demand**(**Bin2Demand**) signal to the **Viper1**(**Viper2**) indicating that feed demand.

Viper1(2)Demand/Viper1(2)Request When **Viper1**(**Viper2**) senses the feed demand from the **Bin1**(**Bin2**), it will send the input signal **Viper1Demand**(**Viper2Demand**) to the resistor **Dol7**. And the **Dol7** outputs the **Viper1Request**(**Viper2Request**) to inform the **Viper2** and the **Viper1** respectively that there is feed demand from another viper. As it is mentioned, those two vipers will communicate through **Dol7** [18].

On(Off)/ReleaseOwnership The **Dol7** will react to the demand from the vipers by displaying different levels of the voltage. When it reach to the middle amount(legal voltage), it issues

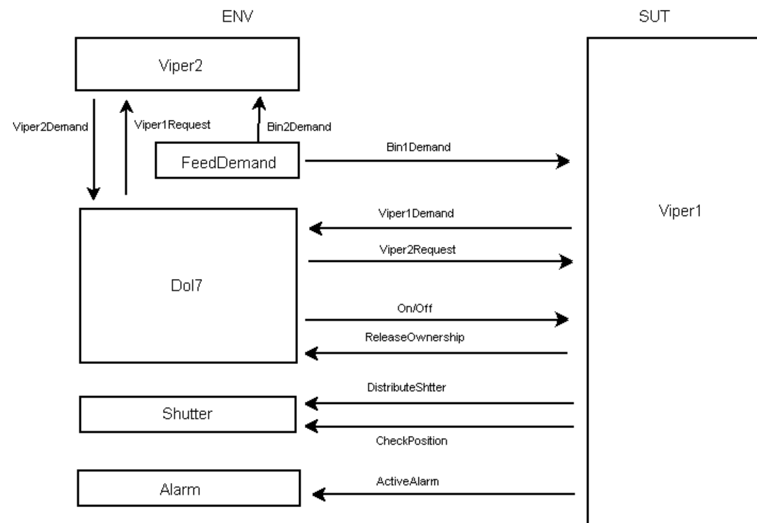


Figure 4.1: Structure of the model1

the **On** signal to active the controlling **Viper1** to distribute the shutter to the position of its ownership. Otherwise, if the voltage of the **Dol7** is illegal, it will notify the **Viper1** by the **Off** signal.

After doing the requiring internal action, the **Viper1** will enable the **ReleaseOwnership** to notify the **Dol7** the ownership of the shutter being released.

DistributeShutter When the ownership of the shutter is legal(according to the voltage of the **Dol7**), the controller **Viper1** will issues the **DistributeShutter** for adjusting the position of the shutter to its owner.

CheckPosition The controller **Viper1** has the responsibility to check for the validity of the shutter position("validity" means the ownership of the shutter matches with its position). The checking is done by controller out-putting the requiring signal **CheckPosition** to the **shutter** component(template).

ActiveAlarm If the real shutter position can not be matched with the ownership of the shutter, the controller will active an alarm by means of stimulating the **Alarm** template.

4.2 Model description

4.2.1 The feed demand template

The feed demand template in Figure4.2 is used to nondeterministically create the feed demand in either **Vipers**. From its **idle** location, it have two choices to go further. It does the selection by nondeterministically choosing one way to go. In other words, it can either choose to go through the channel leading from **idle** to **vip1**, or the channel from **idle** to **vip2**.

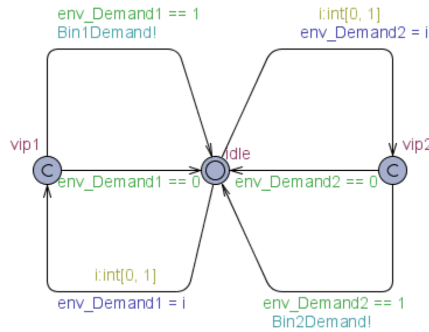


Figure 4.2: The Feed Demand template

If the former(latter) one is chosen, it performs checking the demand of **Bin1(Bin2)**. During the transition from **idle** location to **vip1(vip2)** location, it nondeterministically assigns a boolean value to the variant **env_Demand1** (**env_Demand2**). If the **env_Demand1** (**env_Demand2**) equals "1", the synchronized channel labeled **Bin1Demand** (**Bin2Demand**) will be enabled immediately, synchronizing with the **Viper1**(see Figure 4.4) and the **Viper2**(see Figure 4.5) respectively.

Otherwise, if the **env_Demand1** (**env_Demand2**) equals "0", it will directly go back to the location **idle** from **vip1(vip2)** without any action and delay.

4.2.2 The Dol7 template

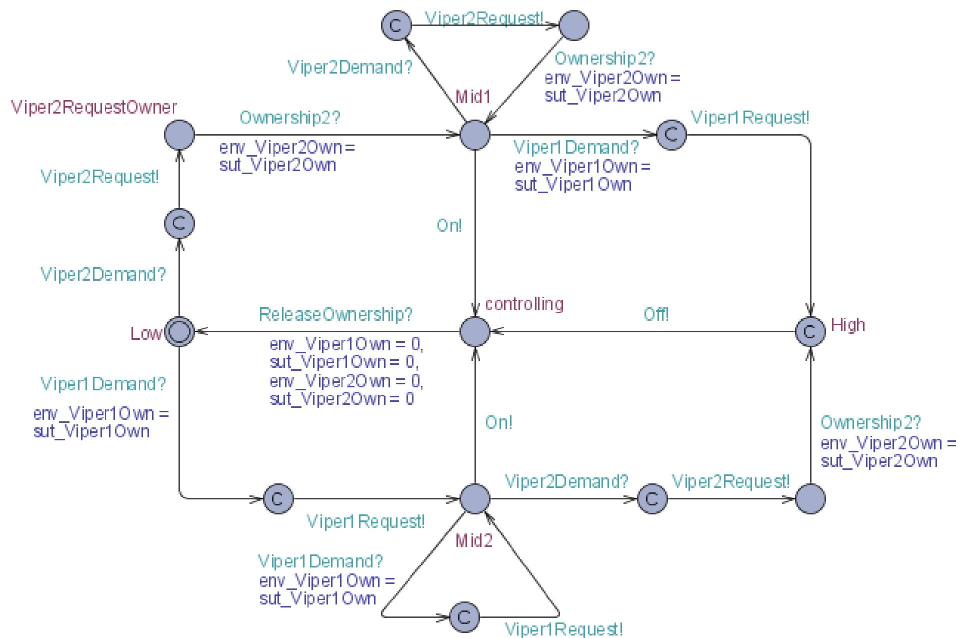


Figure 4.3: The Dol7 template

Figure 4.3 shows the resistor named **Dol7**. It is the communication tool between **Viper1** and **Viper2**. Whenever there is feed demand in either vipers, or ownership of the shutter is assigned

to either one, they will notify each other through **Dol7**. This **Dol7** template uses the **Low**, **Mid1** (**Mid2**), **High** states to correspond with three level voltages: $<$ **2v**, **5v**, **6.7v** in the real resistor **Dol7**. The **Low** state, is the initial state, representing the ownership is free. The **Mid1** (**Mid2**) states, there is only **Viper2** (**Viper1**) holds the ownership of the shutter. Whereas, the **High** is the illegal state, representing two vipers hold the ownership of the shutter at the same time. That will then lead to force two vipers to release the ownership.

The Dol7 template starts from the location **Low**. It starts when it received synchronized signal **Viper1Demand** (**Viper2Demand**) from the **Viper1** and the **Viper2**. If it receives **Viper1Demand** (**Viper2Demand**) when it is in the **Low** location, it will be led to an committed location with label "C". During the transition **Viper1Demand**, the value of **sut_Viper1Own** will be passed to the environment by means of the "env_Viper1Own = sut_Viper1Own". The demand from the **Viper1** (**Viper2**) will be notified the **Viper2** (**Viper1**) using synchronized channel **Viper1Request** (**Viper2Request**)(see Figure 4.5, 4.4) without any delay. The **Viper1Request** leads the template to the location **Mid2**, whereas the **Viper2Request** leads the system to the location **Viper2RequestOwner**. When in that location, it may pass value of a SUT variable **sut_Viper2Own** to the ENV part by means of signal **Ownership2**. After the transition labeled with **Ownership2**, the templates also moves on to the location **Mid1**.

In **Mid2(Mid1)**, it can accept any demand from **Viper1(Viper2)** through the synchronized channel **Viper1Demand** (**Viper2Demand**), and follow the same action as the description in the last paragraph. It can also asking the **Viper1** to control the distribution of the position of the shutter through the synchronized channel **On**. And it is transited to the location **controlling**. It stays in that location, until receiving the synchronized channel **ReleaseOwnership** from **Viper1** indicating that the control is done. During the **ReleaseOwnership** transition, 4 variables will be reset: **env_Viper1Own**, **sut_Viper1Own**, **env_Viper2Own**, **sut_Viper2Own**.

But when it gets the signal **Viper2Demand(Viper1Demand)** in location **Mid2(Mid1)** , it will immediately inform **Viper1(Viper2)** using two ways respectively: **Viper2Demand** $\rightarrow$ **Viper2Request** $\rightarrow$ **Ownership2**, and **Viper1Demand** $\rightarrow$ **Viper1Request**. Going through either of the two ways, it will transit to the **High** location. In that location, without any delay, it will ask the **Viper1** to release the shutter ownership of the both **Vipers** using the synchronized channel **Off**, and move to the location **controlling**. Unless getting the reaction from the **Viper1**, the transition **ReleaseOwnership** will be enabled, and the 4 variables: the **env_Viper1Own**, the **sut_Viper1Own**, the **env_Viper2Own** and the **sut_Viper2Own** will be reset as well. The system goes back to the initial location **Low**.

4.2.3 The Viper1 template

The **Viper1**(see Figure 4.4) template acts as the controlling viper. It is not only responsible for distributing the ownership of the shutter, but also controlling the shutter moving to its owner.

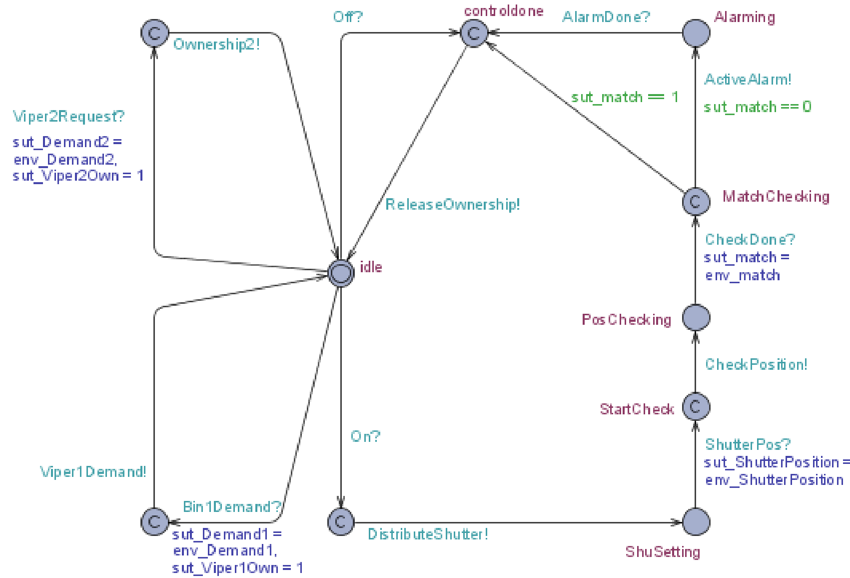


Figure 4.4: The controller Viper1 template

In its initial location **idle**, it may receive the signal **Bin1Demand**, the feed demand of the Bin1 from the ENV part will be passed to it by means of assignment "sut_Demand1 = env_Demand1". During the transition **Bin1Demand**, the controller **Viper1** grants the ownership of the shutter to itself as well. After that, it immediately inform the **Dol7** (see Figure4.3) about those updates by means of issuing the channel **Viper1Demand**.

In the location **idle**, if the **Viper1** gets the signal **Viper2Request** from the **Dol7**, the demand from the **Viper2** will be notified by using "sut_Demand2 = env_Demand2". And it assigns the ownership of the shutter to the **Viper2** (sut_Viper2Own = 1) during that transition as well. Without any delay, it enables the synchronized channel **Ownership2** to broadcast that the **Viper2** gains the ownership of the shutter.

However, it is activated by the signal **On** when in the initial location **idle**, it starts the process of adjusting the position of the shutter according to the ownership of the shutter. After the transition **On**, it issues the synchronized channel **DistributeShutter** as soon as possible, and arrives the location **ShuSetting**. Until the **Shutter** template in Figure 4.6 returns the current position of the shutter through the synchronized channel **ShutterPos**, the **Viper1** immediately enables an action **CheckPosition** asking the **Shutter** template to check the validity of the shutter position compared to the its ownership. The **Shutter** template will return the **Viper1** about the result of the comparison through the synchronized channel **CheckDone** using "sut_match = env_match". If $sut_match == 1$, meaning the shutter position is in the owner of the shutter, the transition will be led to the location **controldone**. As soon as arriving that location, it will immediately enabled the synchronized channel **ReleaseOwnership** to inform the **Dol7** the releasing of the ownership, and goes back to the location **idle**.

Otherwise, if it is not matched($sut_match == 0$), it will active the alarm by sending synchronized input signal **ActiveAlarm** to the **Alarm** object in Figure 4.7, and go to the location

Alarming without any delay. Until it receives the **AlarmDone** signal from the **Alarm** template, and goes back to the location **controldone**. As soon as possible, it issues the signal **Release-Ownership** to notify the **Dol7** the releasing of the ownership, and moves back to the location **idle**.

The **Viper1** may also react to the signal **Off** when in the location **idle**. As soon as the **Off** arrives, the **Viper1** will issue the channel **ReleaseOwnership**, and then move back to the initial location **idle**.

4.2.4 The Viper2 template

The **Viper2** template in Figure 4.5 is modeled with respect to the real **Viper2**. It reacts to the feed demand from Bin2 by means of asking the controller **Viper1** for the ownership of the shutter. It is also informed of the demand of the **Viper1**.

In the **idle** location of the **Viper2** template (see Figure 4.5), if it senses the feed demand in the Bin2 by the signal **Bin2Demand**, it will notify the **Dol7** through the signal **Viper2Demand** as soon as possible.

When in the initial location **idle**, the **Viper2** may also get the notification **Viper1Request** from the **Dol7** about the feed demand of the **Viper1**.

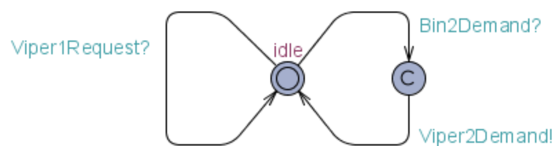


Figure 4.5: The Viper2 template

4.2.5 The shutter template

The **Shutter** template in Figure 4.6 is used to move the shutter to its expected position according to the command from the **Viper1**, and returns the current position to the **Viper1** as well.It can be activated by the synchronized channel **DistributeShutter**. During that synchronized transition, it distributes the shutter to the nondeterministically chosen position by means of assigning either "0" or "1" to the variant **env_ShutterPosition**. Then it is immediately passed to the controller **Viper1** using the synchronized signal **ShutterPos** (see Figure 4.4).

However, if the **Shutter** template receives the signal **CheckPosition**, it will also be activated and led to a committed location labeled with "C". During that transition, it checks the validity of the shutter position compared to its ownership using the function **Check(env_ShutterPosition,**

env_Viper1Own), and stores the result in the **env_match**. After doing that, it will immediately issue the signal **CheckDone** to inform the controller **Viper1**.

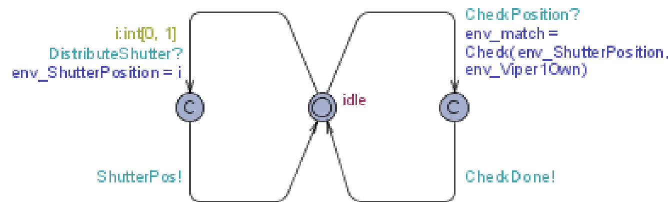


Figure 4.6: The Shutter template

4.2.6 The alarm template

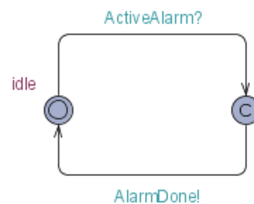


Figure 4.7: The Alarm template

The **Alarm** object in Figure 4.7 is very simple. It only reacts to the synchronized signal **ActiveAlarm**, and then notify the completeness to the **Viper1** by the synchronized channel **AlarmDone** immediately.

4.3 Model checking

As it will later be applied to the on-line testing in Tron, it is necessary that the important properties of the model are satisfied according to the system specification. The following we list the properties having been proved.

- The most important properties is to demonstrate that the model is deadlock free. That property can be written in the formal way, "A[]not deadlock". We apply this property for this sub-model in the Uppaal verifier, and it is verified the satisfied property.
- The query "A[]Dol7.Low imply (env_Viper1Own == 0 && env_Viper2Own == 0)" is a safety property. It states that, whenever the template **Dol7** in its **Low** state, indicating neither the **Viper1** nor the **Viper2** holds the ownership of the shutter.
- The query "A<>sut_Demand1 == 1 imply sut_Viper1Own == 1 ", indicates whenever the **Viper1** has the feed demand, it eventually gets the ownership of the shutter. The

same property that the **Viper2** holds, can be written "A<>sut_Demand2 == 1 imply sut_Viper2Own == 1".

- The liveness property "Dol7.High $--->$ (env_Viper1Own == 1 && env_Viper2Own == 1)" is demonstrated. It states that, the **High** location (of the **Dol7** template) is representing two vipers holding the ownership of the shutter simultaneously.
- The satisfied property "A[]Viper1.Alarming imply sut_match == 0" means the statement that, the alarm is generated because of the not match of the shutter owner and its actual position, is always true.

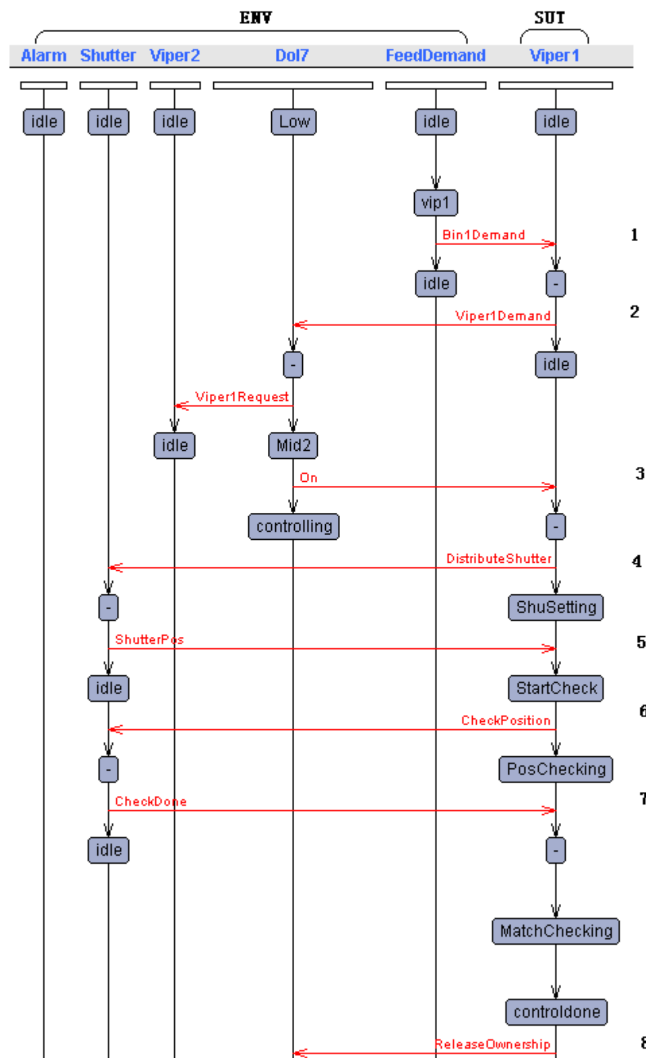


Figure 4.8: The MSC of the 1st test case for the model1

4.4 Manual testing

As it is expected that, the model of the demand feed system (as well as other two sub-models) will be applied in the UPPAAL TRON for automatically test cases generation and online test ex-

ecution. However, due to the limit of the testing resources, we propose to perform the manually offline testing. Several timed traces (with specific test purpose) are picked up directly from the UPPAAL simulator, being considered as the test cases. As it is mentioned that, SKOV offers us an emulate feeding system(see the section 3.4) as the IUT for the testing. That emulate system is established in Labview, named **Dub99**. We will then compare the selected test cases with the **Dub99**, in order to check whether the emulate system conforms to the model specification.

4.4.1 Test case1

Figure 4.8 shows the timed trace for the 1st test case. As it is known that, the **Viper1** is the controller viper among the two sharing vipers: **Viper1** and **Viper2**. The 1st case displays that, when the feed demand from the Bin1 is sensed by the **Viper1**, the controller will responsible for assigning the ownership to itself. And the shutter will successfully be moved to it. That test case will then be carried out to check the same function of the **Viper1** in the **Dub99**. The test moves on step by step, the number of the steps are shown in Figure 4.8 as well.

Step1 & Step2 According to Figure 4.8, when there is a feed demand from the **FeedDemand** template, the input channel **Bin1Demand** will be enabled to synchronized with the **Viper1**. Then, in the **Dub99**, the feed demand from the Bin1 will also be issued(see Figure 4.9).

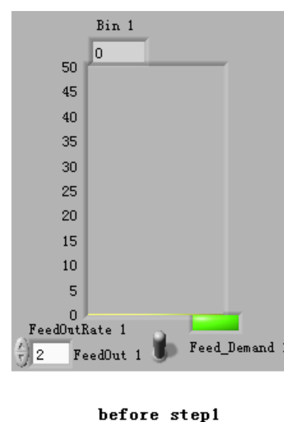


Figure 4.9: The snapshot of the feed demand in Bin1

The reaction to that demand is, the **Viper1** grants the ownership to itself. And the result will be informed the **Viper2** through the **Dol7**. The **Dol7** verdicts the legal or illegal ownership of the shutter by means of displaying different level of the voltage. Figure 4.10 shows the status of the **Dol7**(in the **Dub99**) before the **step1** and after the **step2**(in Figure 4.8). The green box and the volt box in the **Dol7**(see the after status of the Figure 4.10) represent the output from the **Viper1**. That is according to the specification of the signal **Viper1Demand** in Figure 4.8.

Step3 & Step4 After the ENV receives the information of the ownership of the shutter, it will issues an **On** event to the controller **Viper1** in the SUT part(Figure 4.8 step3). This

Figure 4.10: The snapshot of the Dol7

input will also be sent to the **Viper1** in the **Dub99**. In the **Dub99**, the **Viper1** reacts to that signal by issuing an event to distribute the position of the shutter according to the owner of the shutter. Then the **Flap**(Shutter) component in the **Dub99** displays the current position of the shutter. Figure 4.11 shows the statuses of the Flap component in the **Dub99**, that are corresponds to the before and after step3, 4 respectively in the timed trace(see Figure 4.8). Those pair of actions are legal according to the signal **On** and the **DistributeShutter** in Figure 4.8.

Figure 4.11: The snapshot of the Flap component

Step5 & Step6 When the input signal **ShutterPos** with the position of the shutter arrives in the **Viper1**, that input will also be issued in the **Dub99**. It used to active the shutter position checking function of the **Viper1** in the **Dub99**. The **Viper1** carries out that function by activating the timer of the **FlapPosError Alarm** component in the **Dub99**(see Figure 4.12). And at the same time, in the timed trace (see Figure 4.8 step6), **CheckPosition** is output to the ENV. When the **In** box in Figure 4.12 being turned to green from the before status to the checking status, indicating the arriving of the coming of the **ShutterPos** signal from the tester. If the timer is finished, and the **State** box in the after checking status of Figure 4.12 is still dark green, meaning the shutter is in the position of its owner.

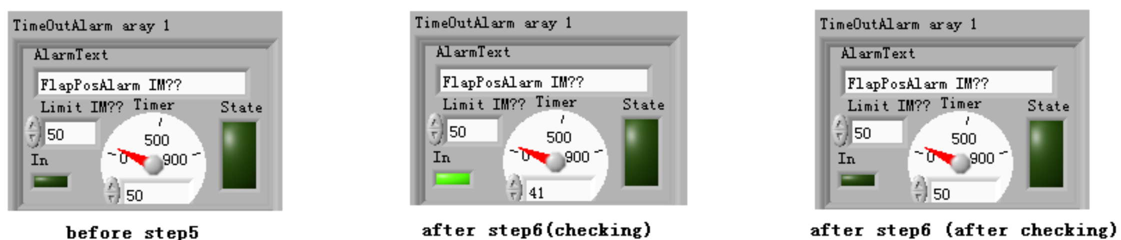


Figure 4.12: The snapshot of the FlapPosError Alarm Component

Step7 & Step8 Then the input channel **CheckDone** is enabled to notify the **Viper1** about the completeness of the position detection. The **Viper1** in the **Dub99** reacts to this input, and release the ownership of the shutter then(see the Figure 4.13). Figure 4.13 shows the statuses of the **Dol7** before and after the step7 and the step8 signals in Figure 4.8. That conforms to the output action **ReleaseOwnership** in Figure 4.8.

Figure 4.13: The snapshot of the Dol7

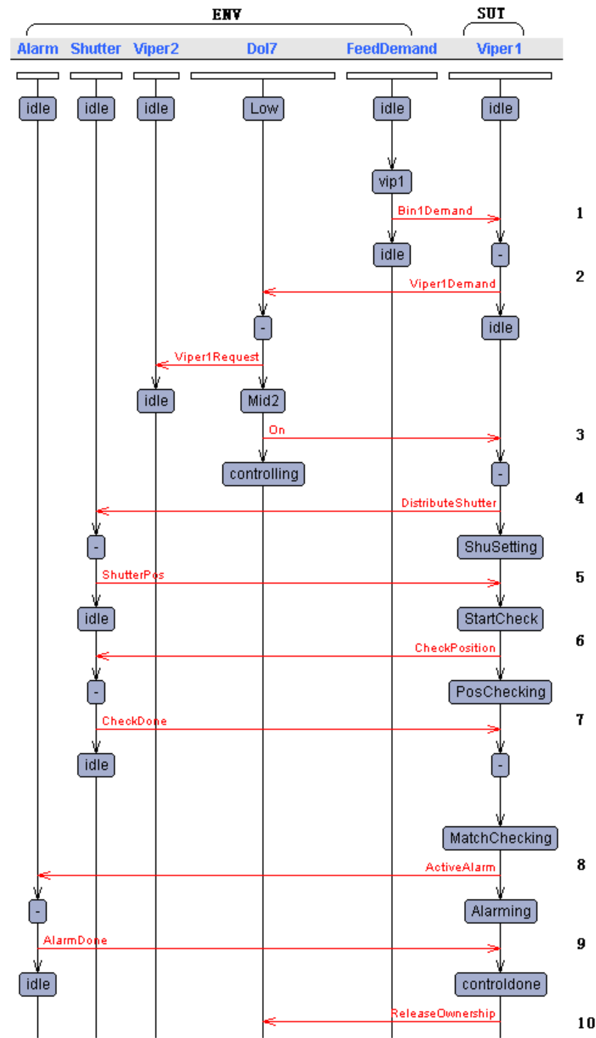
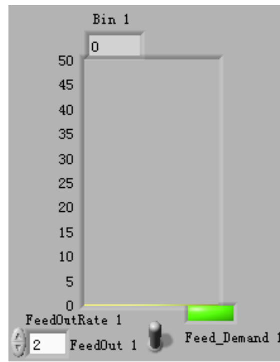


Figure 4.14: The MSC of the 2nd test case for the model1

4.4.2 Test case2

Figure 4.14 shows the 2nd selected test case of the model1. It also describes the reaction of the controller **Viper1** to the feed demand from Bin1. Compared to the test case1 in the section 4.4.1, the test case2 is not only responsible for the granting of the ownership of the shutter, and the distribution of shutter to the holder of its ownership, but also generating the alarm system when the actual position of the shutter is not the shutter owner.



before step1

Figure 4.15: The snapshot of the Bin1

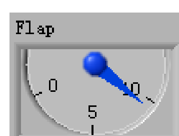
Step1 & Step2 The signal **Bin1Demand** will be issued when the **FeedDemand** template in the model creating the feed demand for the Bin1. That signal is used to inform the **Viper1** about the feed demand, and tries to ask for the ownership of the shutter. In the implementation (**Dub99**), the arriving of the input signal is interpreted into the feed demand in the Bin1 component(see Figure 4.15). When the **Viper1** in the **Dub99** receives the demand from the Bin1, it assigns itself the ownership of the shutter. And it notify the **Viper2** the result through **Dol7**. The green box1 and the voltage box in Figure 4.15 after step2 status indicates the output from the **Viper1**. That conforms to the output through the signal **Viper1Demand** in the step2 of Figure 4.14.

before step1

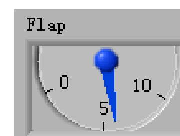
after step2

Figure 4.16: The snapshot of the Dol7

Step3 & Step4 Then the signal **On** is enabled to active the control function of the **Viper1**(both in the model system and in the Dub99 system). The **Viper1** reacts to that by means of issuing the action to move the shutter. The after status in Figure 4.17 shows the current (erroneous) position of the shutter after the **DistributeShutter** transition (see Figure 4.14, step4).



before step3



after step4

Figure 4.17: The snapshot of the Flap component

Step5...Step8 In the timed trace (see Figure 4.14), the step5 intends to sent the result of the position to the controller **Viper1** in the SUT through the synchronized channel **Shutter-**

Pos. And the **Viper1** issues the event **CheckPosition** to detect the validity of the current position of the shutter in terms of the ownership. As in this case the actual position of the shutter is not the expected one, so that the illegal result will be sent back to the **Viper1** through the **CheckDone** signal (see Figure 4.14 step7). And the controller **Viper1** will enable the transition **ActiveAlarm** in order to generate the alarm for that invalidity.

The IUT **Dub99** conforms to those specified actions(see Figure 4.12). The sub-Figure called step6,7 checking shows the status (by means of the "In" box turning green) when the **CheckPosition** requirement is issued. As soon as the "In" box changed to dark green, the result of the detection will be sent to the **Viper1** in the **Dub99**. And the alarm will be generated according the input detection result. The "State box" turns red, indicating the alarm is activated(see Figure 4.18).

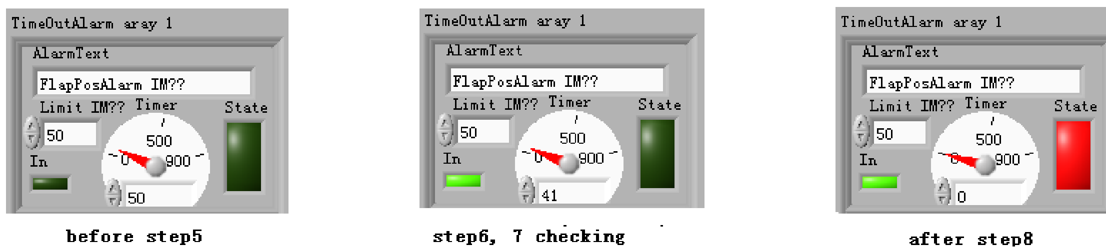


Figure 4.18: The snapshot of the FlapPosError Alarm component

Step9 & Step10 The step9 in Figure 4.14 performs an extra action to ask the **Viper1** to control the releasing the ownership of the shutter. The step10 is the reaction of the **Viper1** using the transition **ReleaseOwnership**. The after step10 sub-Figure in Figure 4.19 presents the status of the **Dol7** after being issued to release the ownership of the shutter. Until now, the test case2 is completed, and passed.

Figure 4.19: The snapshot of the Dol7 component

4.4.3 Test case3

The test case shown in Figure 4.20 describes that, when the **Dol7** measures the two vipers holding the ownership of the shutter simultaneously, the **Viper1** will control both vipers to release the ownership.

Step1 & Step2 The **Bin1Demand** signal is an input action, sent from the ENV part (the **Feed-Demand** template) to the **Viper1** in the SUT part. That input signal will be sent to the

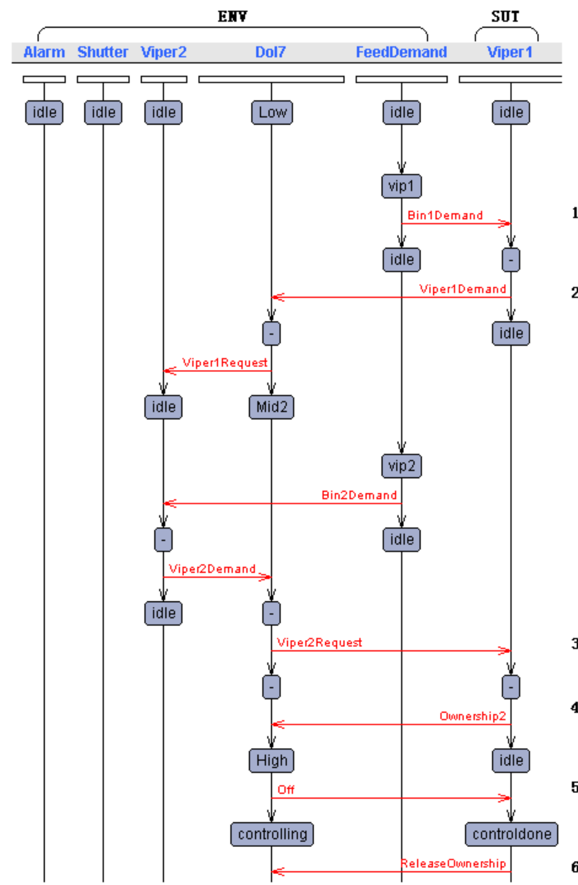


Figure 4.20: The MSC of the 3th test case for the model1

Dub99, indicating that there is feed demand in the Bin1(see Figure 4.21). The **Viper1** senses the feed demand, and assigns the ownership of the shutter to itself. Then it notifies the demand and the ownership to the **Viper2** through the **Dol7**. The box1 in the **Dol7** turns from the dark green to light green (see Figure 4.22) indicates that the **Dol7** receiving the information from the **Viper1**. As it is shown in Figure 4.20, the output channel **Viper1Demand** with the result of the ownership from the **Viper1** to the **Dol7** is the expected action. The "5" volt in the **Dol7**(see Figure 4.22) corresponds to the **Mid2** in the template **Dol7** (see Figure 4.20), which is also legal.

Step3 & Step4 Then the **Viper2** gets the demand signal **Bin2Demand** from the **FeedDemand** template. It immediately informs the **Viper1** about the **Bin2** demand through the **Dol7** template. When the input signal **Viper2Request** arrives in the **Viper1** template(see Figure 4.20). The feed demand from the Bin2 in the **Dub99**(see Figure 4.23) performs the same action as the **Viper2Request**. In the **Dub99**, the result of the reaction of the controller **Viper1** to the Bin2 demand is shown in the **Dol7** voltage box(see Figure 4.24). That is, the **Viper2** gains the ownership of the shutter as well, and the voltage of the **Dol7** is changed to "6.7"(see Figure 4.24 after step4 sub-Figure). This output action is a valid action according to the output signal **Ownership2** in Figure 4.20.

Step5 & Step6 According to the timed trace in Figure 4.20, the **Off** channel then is enabled for the **Viper1**. In the **Dub99**, the output action for the input signal **Off**, leads to release the

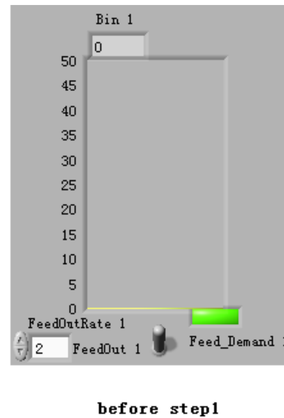


Figure 4.21: The snapshot of the Bin1 feed demand

Figure 4.22: The snapshot of the Dol7

ownership from both of the vipers. That is displayed in Figure 4.25, the green box1 and the green box2 in the before step5 sub-Figures turn to the dark green as they are shown in the after step6 sub-Figure. That conforms to the output **ReleaseOwnership** in Figure 4.20. It demonstrates that, the third test case is passed successfully.

4.5 Ambiguity

During the modeling, we found some ambiguous specifications. That we think is necessary but is not specified in the SKOV document [18]. And we also found some inconsistencies between the SKOV specification and the **Dub99**. We collect those unclear points, and discuss them in the following.

- The initial position of the shutter is in the not controlling viper **Viper2**. We capture the snapshot of the initial state of the shutter(see Figure 4.26). It is not the result of the random choosing. In our model we assume that shutter is in the position of the **Viper2** at the beginning.
- We capture the screenshot of the **Dol7** box in the Dub99B LabView(see Figure 4.27). It presents, when **Viper1** and **Viper2** ask for the shutter ownership at the very same time, the out volt in Figure 4.27 is changed from **5v** → **6.7v** → **0v** immediately, and the two color boxes after the number 1 and 2 in **Dol7** in Figure 4.27 will also be changed. The policy for those color box will be changed from **one green box** → **two green boxes**

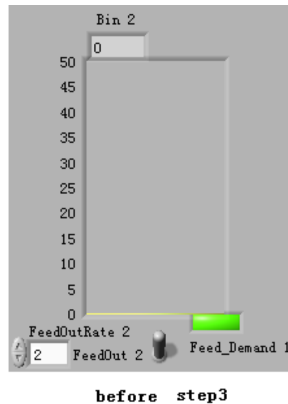


Figure 4.23: The snapshot of the Bin2 feed demand

Figure 4.24: The snapshot of the Dol7

→ **two dark green boxes**. The **two green boxes** indicates, the two vipers can hold the ownership of the shutter in a very short amount of time when both of them require the shutter. That is different from our former assumption that, two vipers can not hold the shutter at the same time. So that we modified the model according to **Dub99**.

- In our model, we assume that, there are always two vipers sharing one **Dol99B**. They are communicate with each other through the **Dol7**. In the **Dub99** LabView, it also shows the case when there is only one viper using **Dol7**. As it is specified, it don't need the resistor **Dol7**. In other words, the **Dol7** will have no output(see Figure 4.28).

- Until now, the model1 system is not so ideal. Some restrictions, which is expected in the SKOV document [18], are not specified in the model. E.g. the system specification [18] indicates that, the shutter position alarm will be generated if the shutter can not be adjusted to its owner within 5 minutes. That is presented by the FlapPosAlarm timer component in the **Dub99**(see Figure 4.29). However, our model did not set the timer for the alarm, due to that alarm timer will caused the deadlock in the model. Other small functions, e.g. the resistor alarm is not modeled as well.

4.6 Summary

This chapter describes the modeling in terms of the demand feed function of the feeding system. The model is explicitly established into the ENV and the SUT parts. Some of its important properties are verified. Three test cases are manually generated, and run against (Dub99). And it is demonstrated to pass all the three test cases. At the end of the chapter, the ambiguity

Figure 4.25: The snapshot of the Dol7

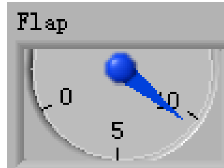


Figure 4.26: Snapshot of the flap in Dub99 LabView

points, which may be caused by the unclear system specification and the inconsistence between the specification and the Dub99, are presented.

Figure 4.27: Snapshot of the Dol7 in Dub99 LabView

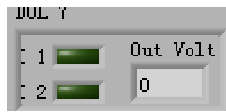


Figure 4.28: Snapshot of the Dol7 in Dub99 LabView(when there is only one viper working)

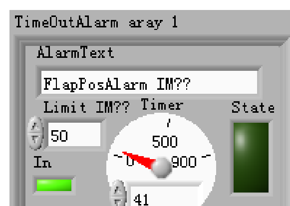


Figure 4.29: Snapshot of the FlapPosAlarm

Chapter 5

Model2 - Calibration

5.1 Model purpose and structure

The Model2 simulates the calibration process of the feed system. It contains 5 templates, mainly shown in Figure 5.2, 5.3, 5.4, 5.5, 5.6. It is also modeled into **ENV** and the **SUT** parts, see Figure 5.1.

When the controlling viper **Viper1**in Figure 5.3 gets the **calibrate** request from the **user** in Figure 5.2, it will start to react to the request by means of controlling the **Dol99B**. The function of the Dol99B is separated into several sub-functions. They are carried out by means of instantiating the template Action(see Figure 5.5) or the template CheckSignal(see Figure 5.6) with different parameters. The **Dol99B** will not only return the **Viper1** the current calibration value, but also the deviation of the old and the current calibration value.

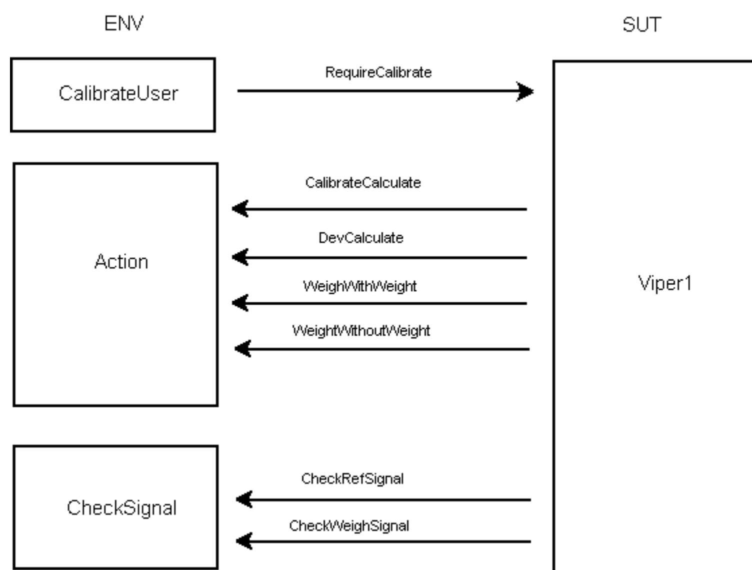


Figure 5.1: Structure of the model2

The model2 system has 19 synchronized channels, 9 variants, and 1 clock variable. Figure 5.1 shows the main components of the model2, and the communication between them. The main communication channels are explained in the following paragraph.

RequireCalibrate The signal **RequireCalibrate** is used to inform the controlling viper (**Viper1**) that, the user(farmer) asks for the calibration of the **weighing drum**. The **Viper1** will start the calibration process by means of controlling the **Dol99B**.

CheckRefSignal/CheckWeighSignal They are used to check the reference signal and the weight signal respectively. Those signals are used to calculate current weight.

WeighWithWeight/WeighWithoutWeight They are used to active the **Dol99B** to calibrate the weight of the drum in the with feed and without feed situation. In other words, they are used to calculated the the maximum and the minimum weight of the drum.

CalibrateCalculate It is used to enable calculating the deviation of the maximum and the minimum weight, which is the current calibration of the drum.

DevCalculate The model2 can not only calibrate current weight of the drum, but also calculate the deviation between the old calibrate and the current calibrate. When the **Viper1** sends this **DevCalculate** signal to the **Dol99B**, the **Dol99B** will do the Deviation calculation.

5.2 Model description

5.2.1 The CalibrateUser template

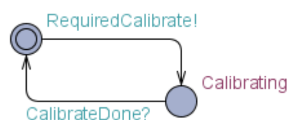


Figure 5.2: The CalibrateUser template

The user template in Figure 5.2 simulates the calibration requirement from the user(farmer). If user issues the synchronized signal **RequiredCalibrate**, the location will be transited to **Calibrating**. It indicates that, the controller **Viper1** is waked to control the process of the calibration. This template will not return to its start location until it gets the **CalibrateDone** signal from the **Viper1**.

5.2.2 The Viper1 template

The **Viper1** template(see Figure 5.3) is used to control the whole calibration process. It grants the scale to the calibration process, adjusts the drum to the desired position, does the drum weight calibration and calculates the deviation. It is activated when it receives the signal **RequiredCalibrate** from the user in Figure 5.2. During this transition, it checks the legal scale for calibration by nondeterministically making choice between "0" and "1". If the scale is illegal("sut_scale == 1"), it will enable the synchronized channel **CalibrateDone**, and goes back to the initial location.

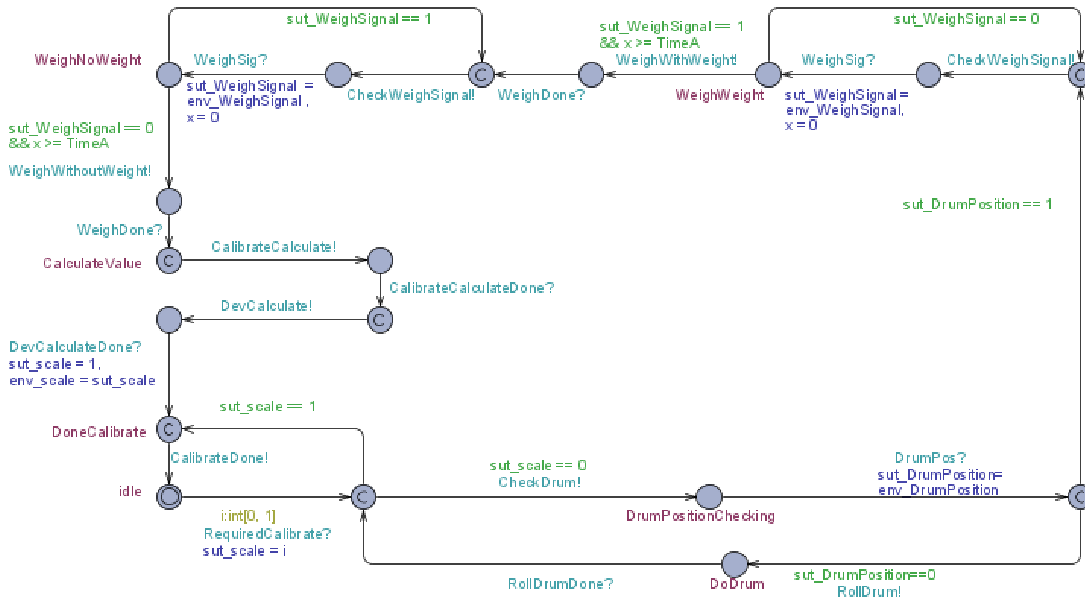


Figure 5.3: The viper1 template

Otherwise, if **Viper1** owns the scale for calibration("sut_scale == 0"), it immediately sends out the signal **CheckDrum** synchronized with the template in the Figure 5.4. The drum position will be returned through the transition **DrumPos** using "sut_DrumPosition = env_DrumPosition". If the drum is not in its top plate(**sut_DrumPosition** == 0), it will be rolled by means of the channel **RollDrum** to synchronized with the drum template which is one of the instances of the **Action** template(see Figure 5.5). The signal **RollDrumDone** indicates the channel current drum rolling is finished.

Until the **sut_DrumPosition** == 1, it starts checking the weigh signal by issuing the transition **CheckWeighSignal** as soon as possible. The **sut_WeighSignal** arrives with the input signal **WeighSig**. And During that transition the local clock variable **x** is reset, the template is moved to the location **WeighWeight** as well. The template can not go further, unless the **sut_WeighSignal** == 1, and the clock **x** $\geq$ **TimeA**(equals 2 mtu, represents 10 sec).

And the **WeighWithWeight** will be sent to the **WithWeight** template which is one of the instances of the **Action** template(see Figure 5.5). It will not be moved on until the arriving of the **WeighDone** signal.

Then the weigh signal will be observed again through the synchronized channel **CheckWeighSignal**, until the channel **WeighSig** brings the value of the **sut_WeighSignal**. And the clock **x** is reset as well. If the **sut_WeighSignal** equals "0" and lasts for **TimeA** mtu, and the **WeighWithoutWeight** synchronized transition will be issued.

Unless the **WeighDone** is detected, it starts to calculate the current calibrate by means of the signal **CalibrateCalculate**, and the deviation through the signal **DevCalculate**. The **DevCalculateDone** indicates the completion of the deviation calculation, and the template reaches

location **DoneCalibrate**. It sends out the **CalibrateDone** signal to the template in Figure 5.2, in order to inform the user about the finish of this calibration. Then it goes back to the initial location.

5.2.3 The DrumPosition template

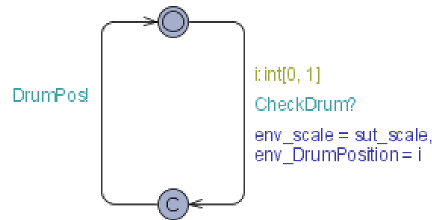


Figure 5.4: The drum position template

The **DrumPosition** template(see Figure 5.4) is activated by the synchronized signal **Check-Drum**. During that transition, the value **sut_scale** will be passed to the environment indicating that the calibrate can be started. And it also checks for the position of the drum, and stores it in the environment variant **env_DrumPosition**. Then it enables the signal **DrumPos** to bring the drum position to the controller **Viper1**.

5.2.4 The Action template

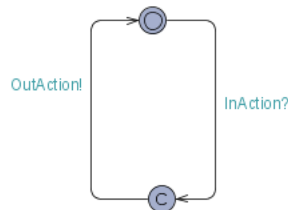


Figure 5.5: The Action template

The **Action** template in Figure 5.5, can be instantiated into 5 different templates(**Drum**, **With-Weight**, **WithoutWeight**, **Calculation**, **DevCalculation**) by the instances of the parameter **In-Action** and **OutAction**.

5.2.5 The CheckSignal template

The **CheckSignal** template in Figure 5.6 will be instantiated into **WeighSignal** template with three parameters. It is controlled by the **Viper1**(see Figure 5.3) to check the weigh signal. The signal is determined through nondeterministically making choice between "0" and "1".

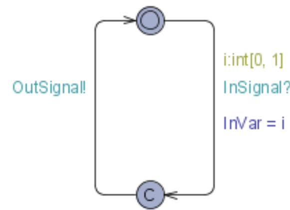


Figure 5.6: The Dol99B template

5.3 Model checking

Before going to the testing phase, there are some properties of the system need to be validated.

- The query "A[]not deadlock" is proposed. It assumes the deadlock will never happen in the model2 system. It is later demonstrated that the model2 system satisfies this query. In other words, the system is deadlock free.
- The weight calibration can only start when it owns the scale, and the drum is in its top plate. The query is written "A[]Viper1.WeighWeight imply (sut_scale == 0 and sut_DrumPosition == 1)" . And it is demonstrated by the Uppaal verifier.

5.4 Manual testing

For the model2 system, we perform the same testing manner as it is specified in the section 4.4. As we've read from the the SKOV document [18] that, there are some important signal for the calibration purpose, e.g. weigh signal, reference signal. However, we could only observe the reference signal in the IUT **Dub99**. So that, we modify some of the templates of the model2 system regarding the testing purpose.

5.4.1 System modification

The Viper1 template

Figure 5.7 displays the modified **Viper1**. It is also activated by the signal **RequiredCalibrate** from the user template(see Figure 5.2), asking for the scale for calibration. If the **sut_scale** equals "0" , it will enable the channel **CheckDrum** to check whether the drum is in its top position. Unless **sut_DrumPosition** equals "1", the main process of the calibrate will start. It is done by enabling the channel **CheckRefSignal** synchronized with new instance of the **CheckSignal** template(see Figure 5.6), to check the validity of the reference signal. If

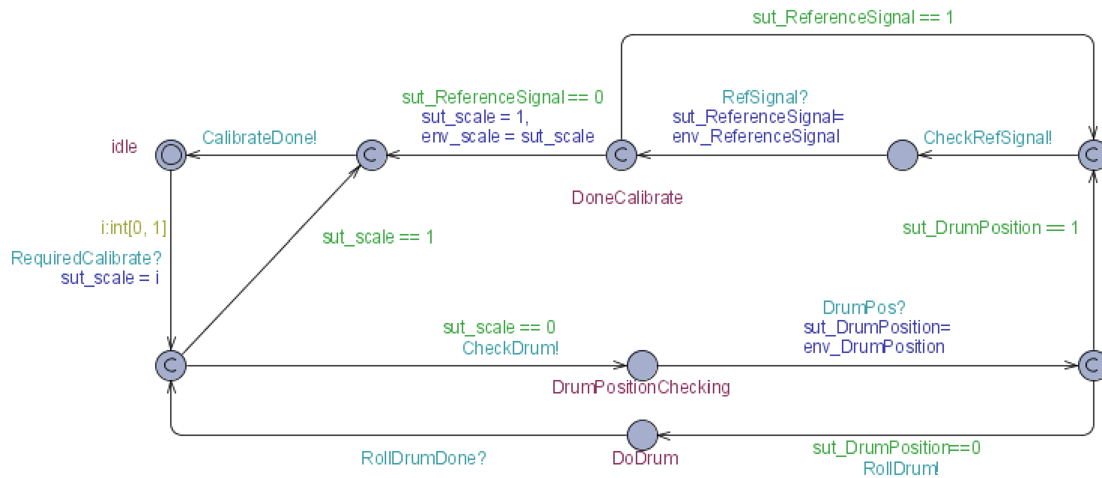


Figure 5.7: The updated viper1 of the model2

"sut_ReferenceSignal==0", the synchronized signal **CalibrateDone** will be issued to notify the user in Figure 5.2.

5.4.2 Test case 1

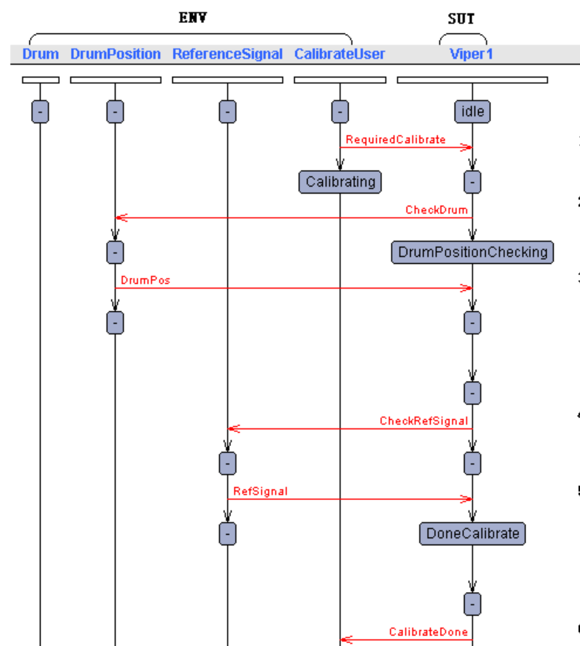


Figure 5.8: The MSC of the 1st test case for model2

The timed trace in Figure 5.8 presents you the case of successful calibration. It begins when the scale for calibration is assigned successfully. Then it is demonstrated that the drum is in the desired (top) position, and starts calibrating by means of detecting the reference signal. Unless it returns that the required value of the reference signal, the system go back to the beginning state. That means, the 1st test case is done.

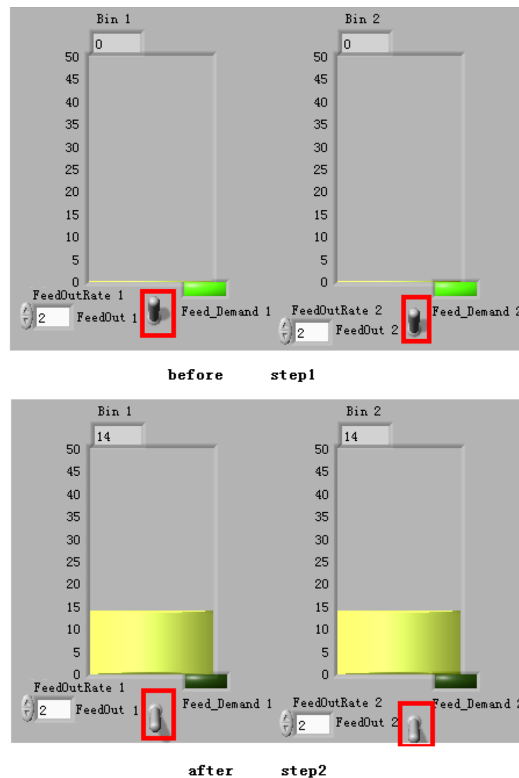


Figure 5.9: The snapshot of the feed bin1, 2

Step1 & Step2 As it is shown in Figure 5.8, the first step of the test case is, that the **Viper1** in the SUT receives the input signal **RequiredCalibrate** from the **CalibrateUser** template in the ENV. The IUT **Dub99** reacts to that signal as expected (see Figure 5.9 after step2). It grants the calibration scale by means of block the feed demand from the feed bin. That is presented in Figure 5.9 using the red rectangle, the before step1 status can still accepts feed demand. However, the red rectangle in the after step2 figure indicates the block. The step2 also issues an event to check the current drum position.

Figure 5.10: The snapshot of the Dol99B

Step3 & Step4 The step3 of the timed trace(see Figure 5.8) passes the position information from the ENV to the SUT through the transition **DrumPos**. Then SUT starts the main process of the calibration by means of outputting the signal **CheckRefSignal** to the **ReferenceSignal** template in the ENV. The performance of the **Dub99** conforms to those pair

of actions. See Figure 5.10, the drum is in its top position. When the controller **Viper1** gets that information, it will asks for calibration using (required) reference signal.

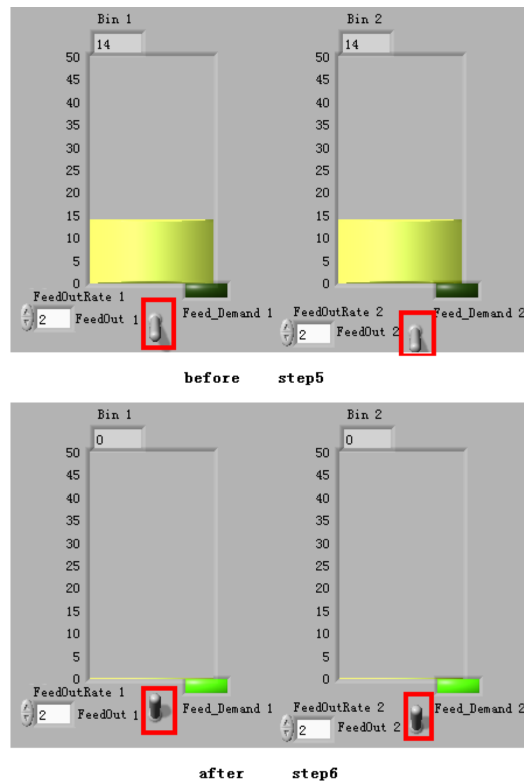


Figure 5.11: The snapshot of the feed bin1, 2

Step5 & Step6 When the expected reference signal is sensed by the **Viper1** through the channel **RefSignal**, the **Viper1** will notify the ENV the completion of this test case using the transition **CalibrateDone**(see the step5, 6 in Figure 5.8). In the **Dub99**, when the "Ref Volt" box in Figure 5.10 presents the desired reference signal, indicates the finish of the calibration. Then the **Viper1** (in the **Dub99**) release the ownership of the scale by means of enabling the feed demand function, see the status changed in the red rectangle of the before step5 and after step6 in Figure 5.11.

5.4.3 Ambiguity

The model2 also has the unclear point as it is specified in the section4.5.

- Figure 5.12 shows the reaction in the **Dub99** when the reference voltage(signal) error is issued. The sub-figure (b) is the expected reference signal, and (c) is the created incorrect voltage. When the case is enabled as shown in the (a), the "Ref Volt Error Weight" alarm timer is also activated(see Figure 5.12 (d)). If the voltage shown in the (c) sub-figure keeps more than the fixed time of the timer, the alarm will be generated(see Figure 5.12

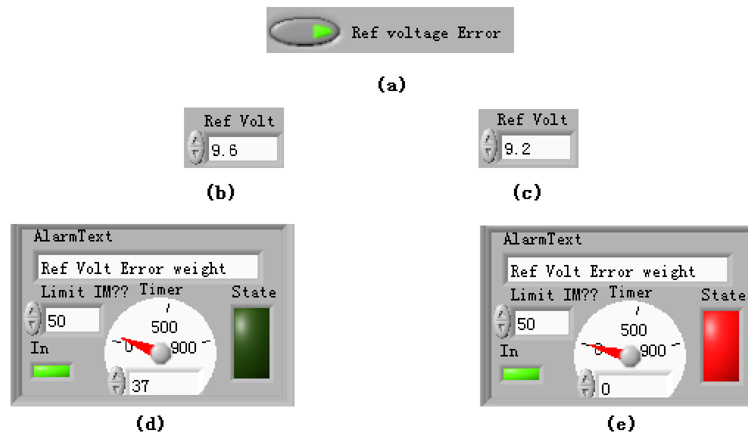


Figure 5.12: The updated Dol99B of the model2

(e)). That is the action in the **Dub99**. However, when we refer to the SKOV document [18], the generation of that alarm is not specified. So that, we are not sure whether is also one of the expected actions of the feeding system.

5.5 Summary

This chapter presents you the model, established according to the calibration function of the feeding system. It is also modeled into the ENV part and the SUT part. After going through all the templates int the model, we propose some of its important properties that need to be satisfied. Then we modify the model according to the Dub99. A test case is produced from the modified model, and is manually teste with the Dub99. At the end of this chapter, the ambiguity in calibration is presented.

Chapter 6

Model3 - Weighing

6.1 Model purpose and structure

The model3 mainly contains 8 templates in Figure 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8. The system is separated into **ENV** and **SUT** parts, the main components of each part are shown in Figure 6.1.

It is used to mimic the weighing function of the feed system. When the **Viper1** sense feed demand from the feed bin, it will be started to fulfill the demand by means of correctly activating and stoping the objects in **Dol99B**, e.g. the **Siloauger** in Figure 6.7, and the **drum** in Figure 6.6.

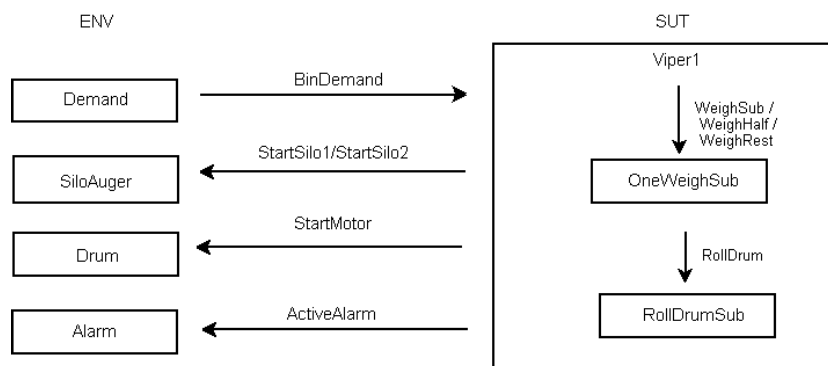


Figure 6.1: Structure of the model3

Figure 6.1 is the construction of the model3 system. The ENV is consisted of 4 templates, and its **SiloAuger** will later be instantiated according to two input signal **StartSilo1** and the **StartSilo2**. SUT has 3 components. The **OneWeighSub** has 3 instances, mainly distinguished by 3 input signals: the **WeighSub**, the **WeighHalf**, and the **WeighRest**. The whole systems contains 19 variants, 6 invariant, 26 synchronized channels, and 3 clock variables. Figure 6.1 presents you the main communication channels, they are:

BinDemand This signal is used to notify the **Viper1** about the current feed demand status of the feed bin: the Bin1 demand feed, the Bin2 demand feed or no feed demand at all. This transition will also help to ask for the weighing scale as well.

StartSilo1/StartSilo2 Different feed bin has its own feed resource. When there is feed demand from the feed bin1(2), the controller **Viper1** will ask the silo auger1(2) for desired amount of feed, so that the channel **StartSilo1**(**StartSilo2**) will be issued.

WeighSub/WeighHalf/WeighRest The feed demand from the bin may be the same as drum maximum fill size (called "one portion"), or two times of the fill size, or maybe between one and two portions. Different amount of the feed demand use distinguishing weighing strategies by means of activating 3 templates. The three template are the instances of the template **OneWeighSub**. Each template is activated by one specific signal. There are 3 signals: the **WeighRest** will be issued when the demand is less than or equals one portion; the **WeighHalf** is responsible for the situation when the feed demand is more than one portion, but less than 2 portions; and the **WeighSub** will be enabled when feed demand is more than 2 times of the portion.

RollDrum/StartMotor The drum control process will be started by the **OneweighSub** through the signal **RollDrum**. The enabling of the signal **StartMotor**, indicates the really start of the drum rolling function. In the model3 system, two cases may need that functionality , i.e. before the silo auger transporting the feed to the drum, it is required in its top plate. Otherwise, it will be forced to rotate to that position. And it is also rolling, when the feed in the drum is transported to the feed bin.

ActiveAlarm If time out happens when the drum is rolling, or the silo auger is activating, alarm will be stimulated by means of issuing the signal **ActiveAlarm**.

6.2 Model description

6.2.1 The Demand template

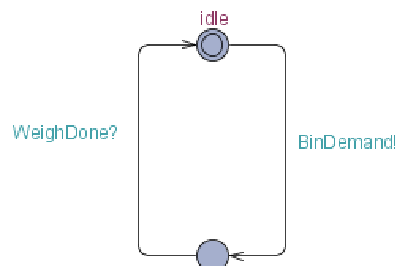


Figure 6.2: The Demand template

The feed demand template(see Figure 6.2) intends to create the feed demand from either feed Bin. It enables the channel **BinDemand** synchronized with the **Viper1**(see Figure 6.3) template. That sends the amount of the feed demand to the controller **Viper1**, and ask for the weighing scale as well. Unless receiving the **WeighDone** notification, it will go back to its initial location **idle**.

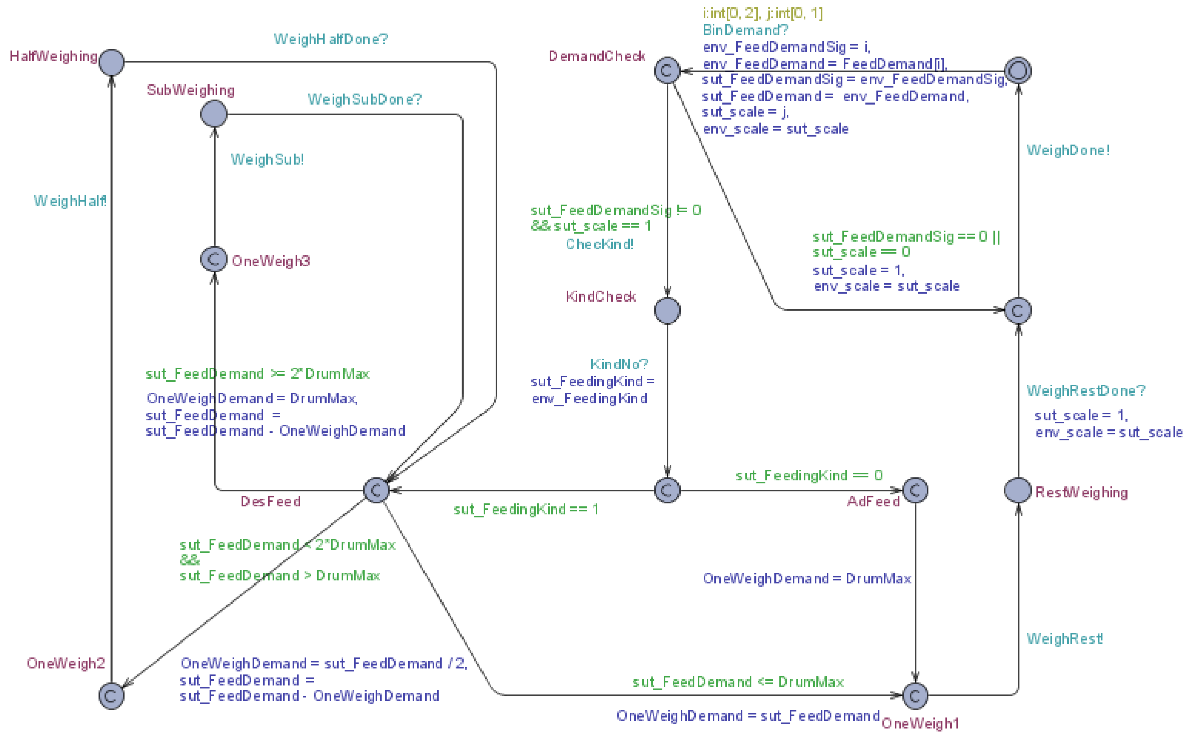


Figure 6.3: The Viper1 template

6.2.2 The Viper1 template

The **Viper1** template in Figure 6.3 controls the weighing process. It is not only responsible for granting the scale to the weighing request, but also deciding the manner(kind) for the weighing. It may make further choice among the three weighing strategies regarding the feed demand, if the second manner(**DesFeed**) is chosen.

When the **Demand** template in Figure 6.2 issues the synchronized signal **BinDemand**, the **Viper1** in Figure 6.3 will be activated. It does its first step, checking the feed demand. The **env_FeedDemandSig** will nondeterministically choose one integer among the integers 1, 2, 3, and gives the corresponding amount of the demand to the variant **env_FeedDemand**. Then the ENV values will be passed to its SUT variables **sut_FeedDemandSig**, and the **sut_FeedDemand**. During that transition, the **Viper1** also decides whether to grant the weighing scale or not. That transition will also lead the template moving to the location **DemandCheck**. The system will return its start state if the **sut_FeedDemandSig** equals "0", or the **sut_scale** equals "0".

Otherwise, the **CheckKind** channel will be enabled. That transition will synchronize with the **CheckSignal** template in Figure 6.8, uses to determine which kind of weighing will be chosen. The result will be returned through the signal **KindNo** using "sut_FeedingKind = env_FeedindKind". If **sut_FeedingKind** == 0, it decides to go to the location **AdFeed**. The value of the variant **DrumMax** will be immediately assigned to the **OneWeighDemand**. The **OneWeighDemand** is the variant, indicating how much feed demand will be satisfied each time. And the system will be transited to the location **OneWeigh1**. And the **WeighRest** channel will

be enabled to active one instance of the **OneWeighSub** called **Rest**. After the **Rest** returns the signal **WeighRestDone**, the **Viper1** will go back to its start state.

Otherwise, the **FeedingKind** == 1, it will be transited to the location **DesFeed** the **Viper1** may have three choice to take. If feed demand is between the **DrumMax**(one portion) and the **2** $*$ **DrumMax**, the **OneWeighDemand** equals half amount of the **sut_FeedDemand**, and the rest of the feed demand will be $sut_FeedDemand - OneWeighDemand$. And it immediately takes the transition labeled **WeighHalf** to stimulated the instance of the **OneWeighSub** template in Figure 6.4 named **Half**. Until gets the signal **WeighHalfDone**, it will be back to the location **DesFeed**.

If the system feed demand **sut_FeedDemand** $\geq$ **2** $*$ **DrumMax**, the **OneWeighDemand** will be the **DrumMax**. And the rest of the demand will be the result of the **sut_FeedDemand** minus **OneWeighDemand**. Then the **WeighSub** will be issued without any delay, and template **Sub** which is also an instance of the **OneWeighSub** will be called. The signal **WeighSubDone** indicates its completion, and enables the channel leading it to the location **DesFeed**.

However, if SUT feed demand is less than or equal to the **DrumMax**, it will directly fulfill the feed demand by using the **Rest** template with the **OneWeighDemand** equaling **sut_FeedDemand**.

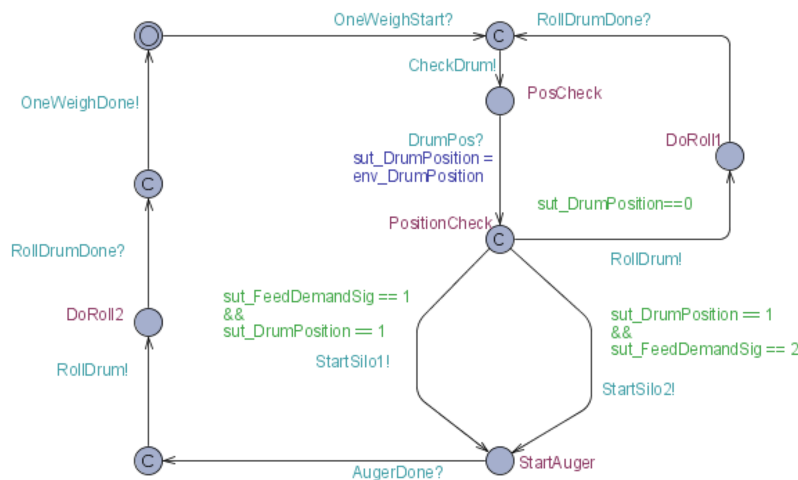


Figure 6.4: One Weigh Sub template

6.2.3 The OneWeighSub template

The **OneWeighSub** template(see Figure 6.4) is subject to the **Viper1**. It simulates the process, the demand feed is transported from the silo auger to the bin by means of correctly activating the rolling drum. It starts when the channel **OneWeighStart** is enabled. And then it immediately checks the drum position by means of enabling the synchronized transition **CheckDrum** with the template in Figure 6.8. The synchronized channel **DrumPos** brings back the current position of the drum using "sut_DrumPosition = env_DrumPosition". If the drum is not in the required

position (**sut_DrumPosition** == 0), the drum will be controlled by means of issuing the channel **RollDrum** which is synchronized with the drum template (see Figure 6.5).

If the **sut_DrumPosition** == 1, it will have two ways to go according to **sut_FeedDemandSig**. The **sut_FeedDemandSig** equals "1" will enables the transition labeled **StartSilo1** synchronized with one instance of the template **SiloAuger** in Figure 6.7. Otherwise, the **sut_FeedDemandSig** == 2 will issue the synchronized signal **StartSilo2**. Both of those transition will end in the location **StartAuger**. After being informed that the silo auger action is finished, the input channel **AugerDone** will be enabled. And the roll drum control will start again by the **RollDrum** signal in order to transit the feed from the drum to the bin. In the end, it takes the channel **OneWeighDone**, and returns to the start state.

6.2.4 The RollDrumSub template

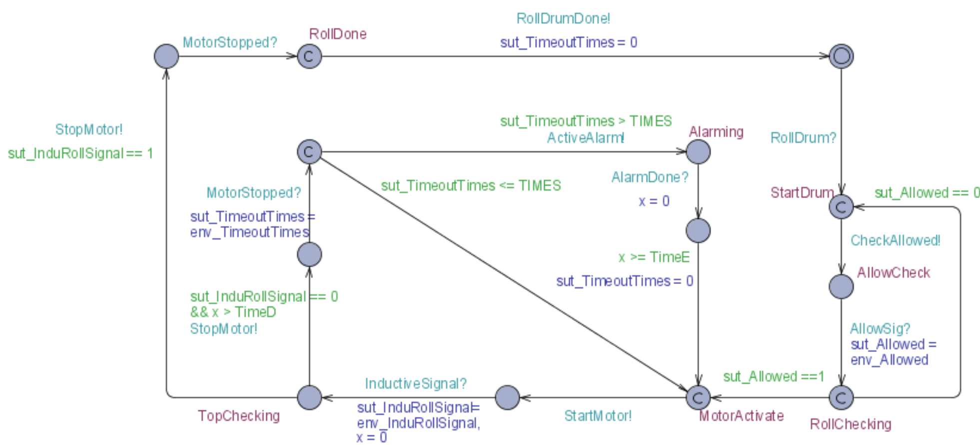


Figure 6.5: The RollDrumSub template

The template in Figure 6.5, is used to control the start and the stop of the drum motor. During the process, it may generated an alarm in terms of invalid starting the drum motor. It waits until it receives the roll drum requirement **RollDrum**, then goes to the location **StartDrum** from the start location. It issues the synchronized signal **CheckAllowed** without any delay. The checking will continue unless it returns **sut_Allowed** == 1 through the channel **AllowSig**. And it reaches the location **MotorActivate**, the **sut_InductiveSignal** is detected using the transition **StartMotor** to synchronized with one of the instance of the **CheckSignal** template as soon as possible. The Signal **InductiveSignal** brings back the amount of the inductive signal. During that transition the clock variable **x** is reset as well. If the **sut_InduRollSignal** == 0 and the clock **x** > **TimeD**(4 mtu), it will enables the **StartMotor** with the template in the Figure 6.6, in order to the **TimeoutTimes** counter. The current time out times will be sent back through **MotorStopped** Then it arrives a committed location with the label "C". It will return to the location **MotorActivate** while the **sut_TimeoutTimes** $\leq$ **TIMES**(equals "5"). Whereas, the **sut_TimeoutTimes** > the **TIMES**, an alarm will be activated by the signal **ActiveAlarm**, and the location is moved to the **Alarming**. Until the transition **AlarmDone** is enabled, the clock

x will be reset. It waits until clock **x** $\geq$ **TimeE**(12 mtu), and reset the **TimeoutTimes** counter, returns to the **MotorActivate**.

Otherwise, the **sut_InduRollSignal** == 1, it will enable the transition **StopMotor** to do the real drum rolling. When the rolling is ended in the committed location **RollDone**, it immediately issues the signal **RollDrumDone** to notify its completeness to the template in Figure 6.4, and reset the counter **sut_TimeoutTimes** too.

6.2.5 The Drum template

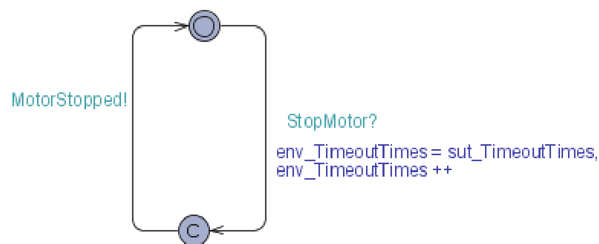


Figure 6.6: The Drum template

The **Drum** template in Figure 6.6, acts as a counter. It is activated by the signal **StopMotor**. And signal also brings the system variant **sut_TimeoutTimes** to its environment. The **env_TimeoutTimes** is increased by "1" through "env_TimeoutTimes ++". Then it immediately issues and synchronized channel **MotorStopped** to inform the result of the times counter to the **RollDrumSub** in the SUT.

6.2.6 The SiloAuger template

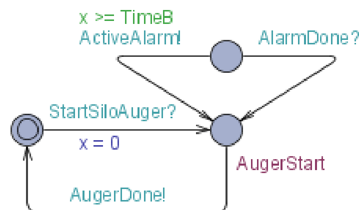


Figure 6.7: The Silo auger template

The silo auger in Figure 6.7 is subject to the controller. It is an template in the SUT, controls the actions of the silo auger in the ENV. It is enabled by the synchronized signal **StartSiloAuger**. During that transition the clock **x** will be reset. Then it reaches the location **AugerStart**. And checks the clock **x** there, if **x** $\geq$ **TimeB**(60 mtu), the alarm template in Figure4.7 will be activated by means of the synchronized signal **ActiveAlarm**. Otherwise, the synchronized channel **AugerDone** will be issued.

6.2.7 The CheckSignal template

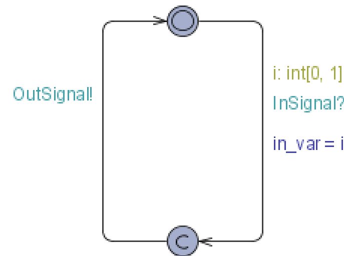


Figure 6.8: The Check signal template

The **CheckSignal** will be instantiated into **FeedKind**, **DrumPosition**, **Permit**, **InductiveSig** template through different parameters. It is used to nondeterministically selecting values for some specified signals.

6.3 Model checking

Due to the time limit, we only propose some properties, which seems more important.

- Due to "deadlock free" property is important for model3 system as it is in the former model systems. The query can be written as "A[]not deadlock". Then, it is applied in the UPPAAL Verifier, and is demonstrated to be satisfied.

- The query "A[]Viper1.KindCheck imply (sut_scale == 1 and sut_FeedDemandSig != 0)" indicates that, the **Viper1** can only start deciding the kind for the weighing when there are two state formulae being satisfied. That are, the weighing process must get the scale, and there must be feed demand in either viper.

- The query "A[]Viper1.OneWeigh2 imply env_FeedDemand == FeedDemand[2]" describes that, whenever the **OneWeigh2** weighing strategy is chosen, is because the feed demand equals FeedDemand[2]. Well, the FeedDemand[2] is set to be 7 in our model, which is between the one portion and two portions.

- "A[] RollDrumSub.Alarming imply sut_TimeoutTimes >= 5" means, it is always true that, the alarm is activated due to the timeout times is more than 5.

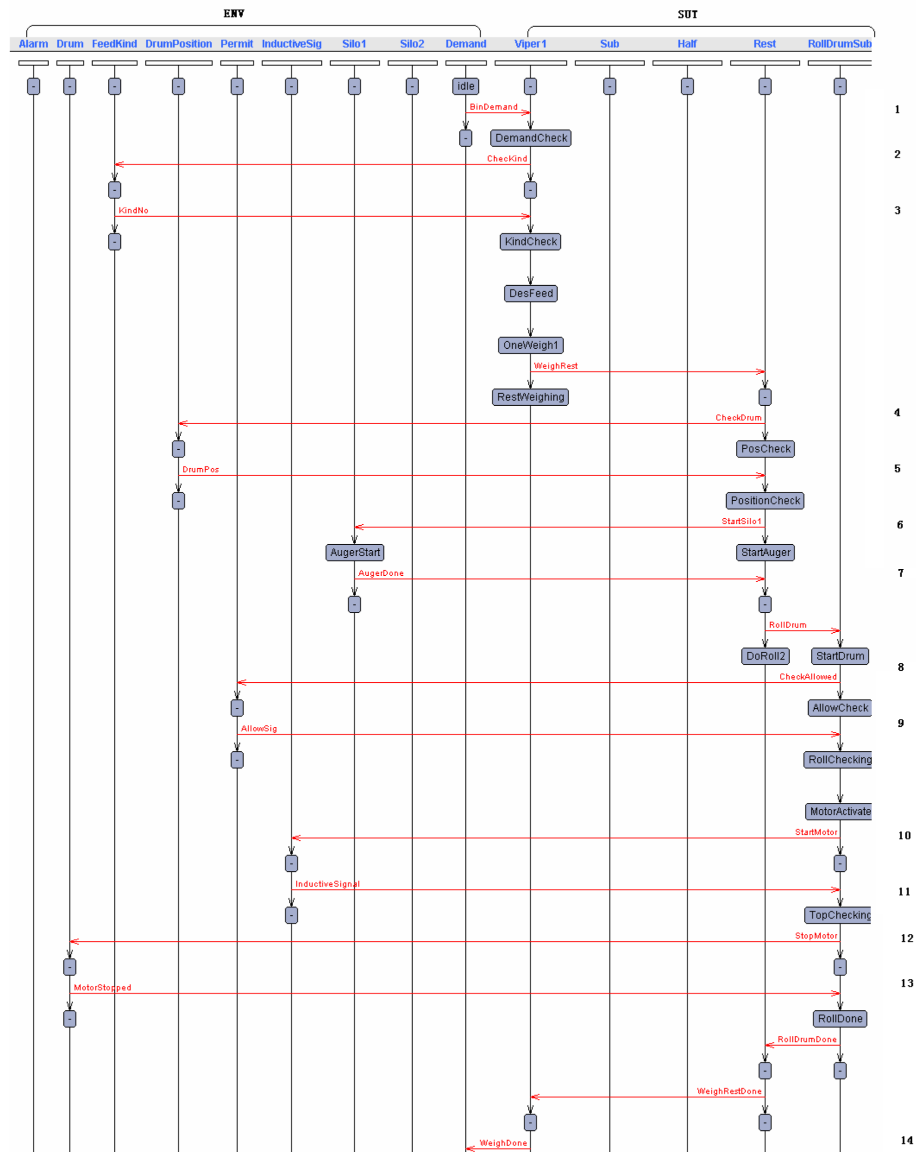


Figure 6.9: The MSC of 1st test case for the model3(when Viper1 has feed demand)

6.4 Manual testing

6.4.1 Test case 1

Figure 6.9 shows the case when the feed bin1 has the feed demand. The ENV will issues an input signal **BinDemand** to the **Viper1** in the SUT part. The **Viper1** assigns the scale to the weighing process, whenever it senses the demand from Bin1. Then it outputs the signals **CheckKind** to the ENV, asking for the way of the weighing. Then the real weighing process begins. It chooses the one weigh rest strategy by means of issuing the channel **WeighRest** which will be synchronized with the **Rest** template. The **Rest** will then output the **CheckDrum** and the **StartSilo1** requirement in order to fill the drum with required amount of the feed. After

silo auger finished its process, it will activate the process transporting the food from the drum to the feed bin by means of enabling the **RollDrum** signal. Then the **WeighRestDone** is issued by the **Rest** template to inform the **Viper1** about the completion. Due to the limit of the **Dub99**,e.g some of the signal or item is not presented, we explain the manually test by skip some steps.

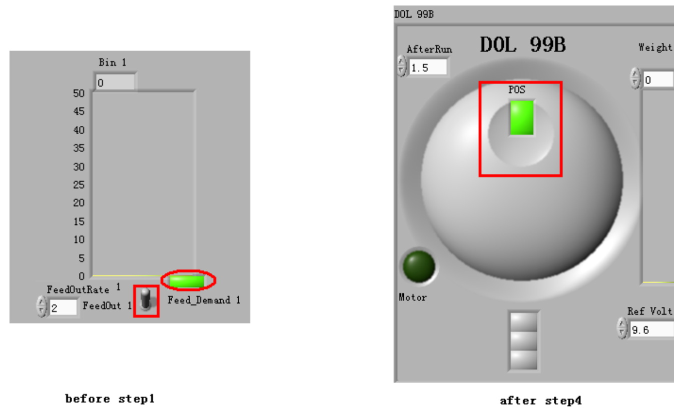


Figure 6.10: The snapshot of the feed bin and the Dol99B

Step1,,Step4 The trace in Figure 6.9 starts when the **BinDemand** is sent from the ENV to the **Viper1**. And the **Viper1** grants the weighing scale, and asks for the kind of the weighing(see Figure 6.9 step2). As soon as the ENV returns the **KindNo** signal, the **CheckDrum** transition is enabled from the SUT to the ENV checking for the current drum position. The implementation (IUT) conforms those steps. The sub-figure before step1 shows the feed demand of the bin1, which is sensed by the **Viper1**. And output is the scale signal from the **Viper1** is sent to the **Dol99B**(see Figure 6.10 before step1, red rectangle). And then the **Viper1** asks for selecting weighing kind, and checking for the current drum position. The drum position is then displayed in the red rectangle of the sun-figure after step4 in Figure 6.10.

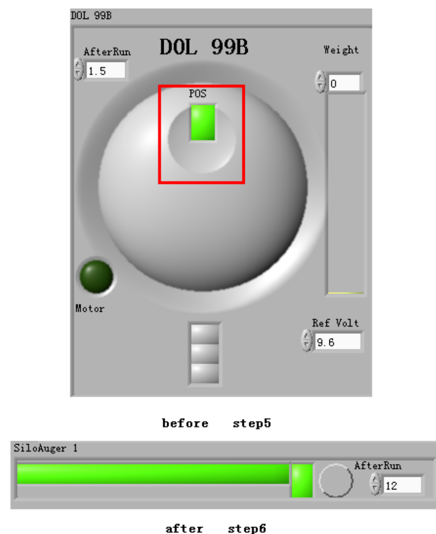


Figure 6.11: The snapshot of the Dol99B and the silo auger

Step5 & Step6 The step5 in Figure 6.9 passes the current drum position(top position) to the the SUT. And the **Viper1** in the SUT activate the **StartSilo1** signal to start the silo1 according to the bin1 demand. In the IUT, the red rectangle in the sub-figure before step5 shows indicates the drum now is in its top position. That will notify the master **Viper1** in the **Dub99**. And the master will control to start the silo auger1. The green box in the sub-figure after step6 in Figure 6.11, indicates being activated.

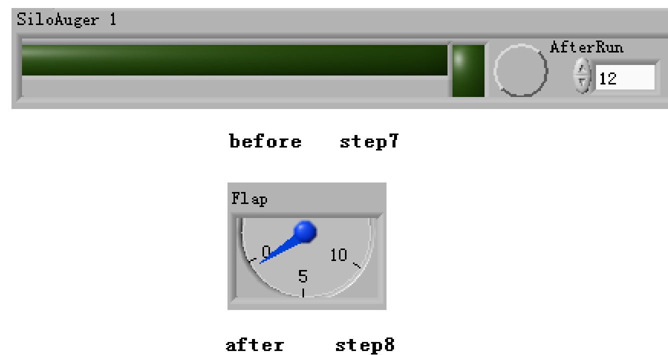


Figure 6.12: The snapshot of the silo auger and the flap

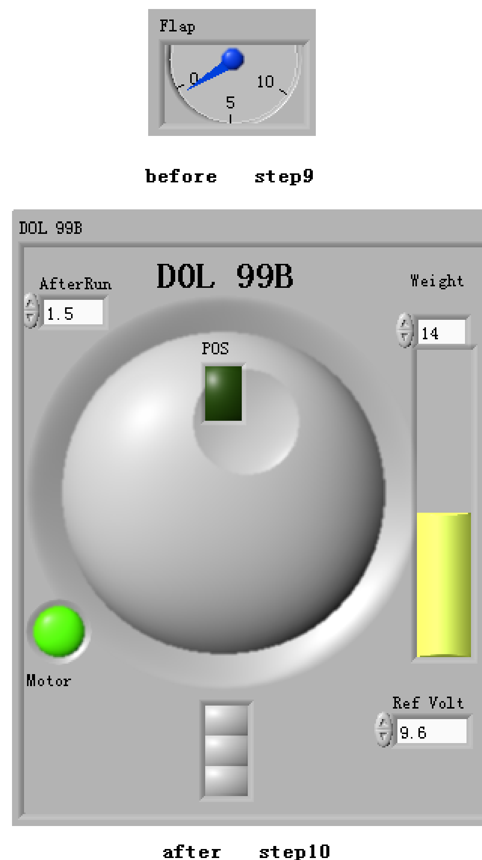


Figure 6.13: The snapshot of the flap and the Dol99B

Step7 & Step8 After completing transport the desired feed to the drum, the silo1 stopes. And inform the SUT through the signal **AugerDone**(see the step7 in the Figure 6.9). The controller in the **SUT** asks whether it is allowed to transit the feed to the feed bin1 now

using the signal **CheckAllowed**(see the step8 in Figure 6.9). And the **Dub99** conforms that pair of actions. As it is shown in Figure 6.12, the dark green silo1 box (in Figure 6.12 before step7) indicates the completion of the feed transporting. That will be inform the master. The sub-Figure after step8 (in Figure 6.12) is the result of the **CheckAllowed** event. It presents the flap is in the legal position with regard to the feed demand of bin1.

Step9 & Step10 The step9 does send the allow signal to the controller in the SUT through the transition **AllowSig** . And the drum motor is activated in order to transit the feed to the bin1(step10 in Figure 6.9). In the **Dub99**, the input signal with the result(see Figure 6.13 before step9) of the allowable checking, will be sent to the master **Viper1**. The master issues an event to start the drum motor. Then motor status in the after step10 of Figure 6.13 indicates being activated by the controller **Viper1**.

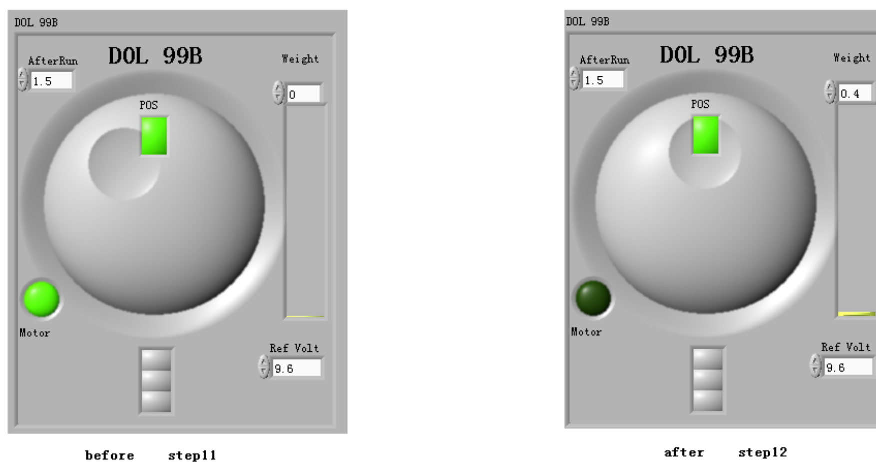


Figure 6.14: The snapshot of the Dol99B

Step11 & Step12 The moving status of the drum is presented using the inductive signal, and it is returned to the SUT through the channel **InductiveSignal**(see Figure 6.9, step11). The controller in the SUT issues the **StopMotor** notify the drum about the time to stop(see the step12). In the IUT **Dub99**, when the input signal arrives in the **Viper1** with the inductive signal (from low to high, as it is shown in the before step11 of Figure 6.14), the **Viper1** will enable an output to stop the drum by means of stopping the drum motor. That is displayed as dark green motor box in the sub-figure after step12 in Figure 6.14.

Step13 & Step14 The quiescence status of the drum is sent to the SUT through the **MotorStopped** channel(see Figure 6.9 step13). That means the weighing process is completed. Then the **WeighDone** signal will be sent to the ENV(step14). As it is shown in Figure 6.15, the **Dub99** conforms to those input and output actions. The sub-figure after step14 indicates the finish of this test case.

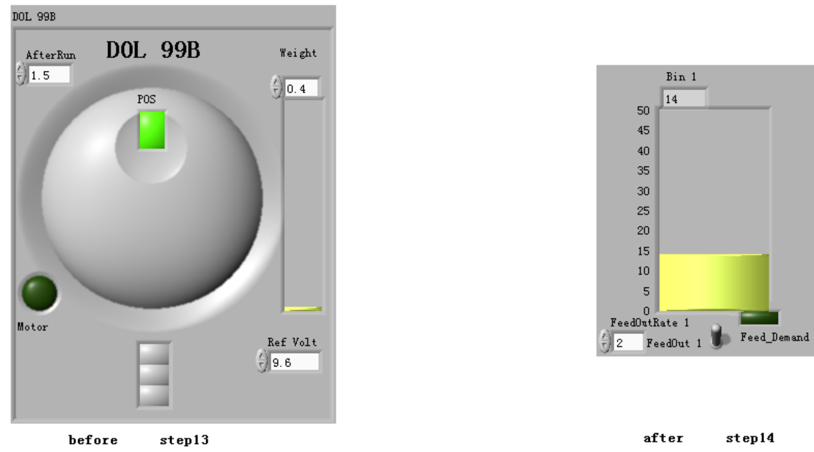


Figure 6.15: The snapshot of the Dol99B and the feed bin

6.5 Ambiguity

We also propose two ambiguities for the model3 system due to the reason specified in the section 4.5.

- The silo auger is not used to mix kinds of food, as it is shown on the LabView, when feed bin1 required food, the food will be pushed into the drum from silo auger 1, otherwise if the bin2 require food, silo auger2 will be issued. See the snapshot of the **Dol99B** in Figure 6.16. Our model is then be modified according to this feature.

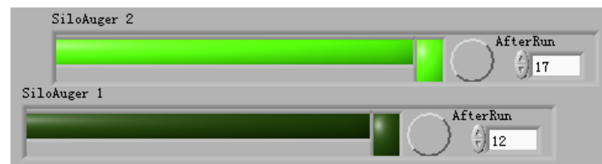


Figure 6.16: The snapshot of the silo auger in the Dub99 LabView

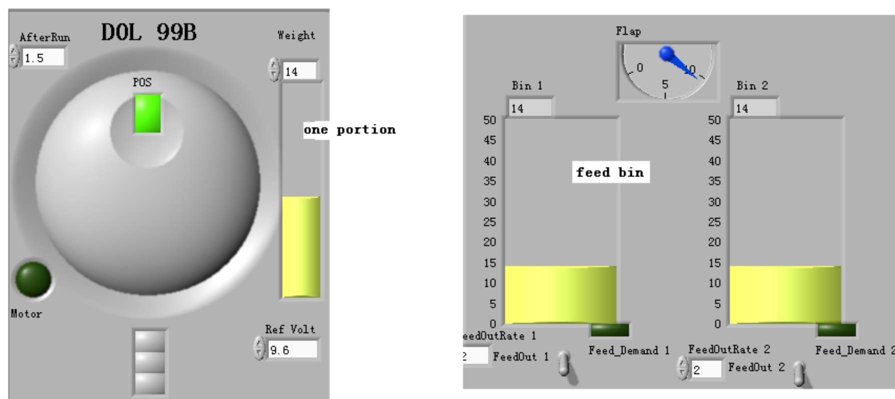


Figure 6.17: The snapshot of the drum and the feed bin in the Dub99 LabView

- Figure 6.17 is one of the snapshots of the Dub99 LabView. It indicates the initial state of the system. The demand of the feed bin1 and the feed bin2 are set to be the same amount. And the size of the weigh drum (or one portion) is the same as either feed bin. In our model, due to the three sub-weighing strategies, we assign the amount of the demand from the two feed bins with different values. We assumes that the demand amount of the feed bin1 is 3, the demand of the feed bin2 is 7, whereas the weight amount of one portion is 5.

6.6 Summary

This chapter explains the model. It carries out the weighing function of the feeding system. The system are modeled into the ENV and the SUT parts. The model is described template by template. Some of the system's properties are also verified. We manually generate one test case from the model, and manually execute it in the Dub99. The result of it is pass.

Chapter 7 Conclusion & Future work

7.1 Summary

This report has been presented here with two purposes: (1) to demonstrate that UPPAAL-TRON is properly tool for model-based online testing, (2) to prove the Skov's product works correctly according to its specification. Throughout the report:

First, we have displayed you the basic theory for the testing: the semantics of the Timed Input/Output transition systems, the Timed Automaton(TA) and the theory of the relativized input/output conformance relation. We also offer you a simple algorithm used by our model-based online testing.

Besides, we have investigated an existing small system called smartlamp. We separately explain its system specification and the environment assumption which are written in UPPAAL modeling language. And the testing have been performed with its java developed Implementation Under Test(IUT) part according to the former presented testing algorithm. By doing so, we concluded that UPPAAL-TRON is a really suitable tool for this kind of testing purpose.

After that, we translated one of the SKOV product, the Climate controller , into UPPAAL model with respect to the model-based online testing purpose. That is, separately established the specification of "Climate Controller" and its working environment. Due to lack of the IUT, we did not fulfill the online testing purpose.

Then, we planed to perform the testing with the SKOV feeding system. We modeled the feeding system into three sub-models according to three main required functions. They are called the demand feed model, the calibration model and the weighing model. Each model system is established into ENV and the SUT parts for the testing purpose. We also verified some important properties for them. Because of the limit time and resources, we only applied those models for manually offline testing using the emulate feeding system called **Dub99**.

During the modeling, we also do effort to reduce the state space explosion. In the modeling of the climate controller, we introduce a new method, that makes it flexible to select only the meaningful elements of an array instead of covering all its elements. In the modeling of the feeding system, we refer to as much as the committed location to reduce the state space explosion.

7.2 Conclusion

In this master project, we use the UPPAAL and UPPAAL TRON platforms to do model-based black-box conformance testing of embedded real-time software systems. We use UPPAAL timed automata to capture the behavior of the software system. Based on this models, we manually derive some test cases, which can be applied on the emulate IUT established using LabView. Some particular behaviors of system can be exercised by using the test cases. Therefore we conclude that, we can use UPPAAL to generate meaningful test cases.

We can conclude that, it is trick and difficult to model the behavior of the system. The reasons include: (1) we should be faithful to the requirements of the SKOV company, but it is nearly impossible to capture all the requirements. (2) the requirements of the company are changed over the time, this adds further difficulty of the modeling. (3) it is usually the case that, the requirements of the company are partially specified, and they may sometimes be contradictory.

Due to lacking of the IUT, we can not use online testing method. So that, we can only depend on the offline testing. Because of the limit time, we do not make a program to do the automate generation of the test cases. We manually derive several test cases. And they are demonstrated to be passed. By observing the execution trace of the test cases, we may conclude that, provided the IUT and the adapter, we can do the model-based online testing using UPPAAL TRON.

7.3 Future work

During this long term project, we have only accomplished some of its missions, due to some reasons. There are still many tasks waiting for us.

First, as we've known that, the UPPAAL TRON and the IUT can not communicate with each other directly, an adapter acting as a translator between those two parts also required to be established. And in the last step, online model-based black-box testing will be carried out to prove how decent UPPAAL TRON is.

Provided the IUT and the adapter, the system model can be applied on TRON for the model-based online testing. The conformance of the IUT in terms of the system specification will be validated. The usability of TRON will be further discussed.

Appendix

Appendix A Climate Controller

A.1 Cooling Control System

Figure A.1, shows the complete process of the Cooling Control Systems. The main objective of the **cooling system** is to take care of the heat. It is used in the livestock buildings when the ventilation is insufficient to reduce the temperature inside. Compared to ventilation, cooling can not only lower the temperature, but also increase the air humidity. The conjunction of the high temperature and the high humidity is believed that, will do harm to the animal's lives. Since cooling will increase the humidity, this product is designed to automatically disconnect when the humidity exceeds its setpoint of the buildings. This climate controller is established to automatically control the inside temperature ensuring that it will never exceed the **temperature setpoint**.

A.1.1 Control Process

The operation of the **cooling control system** is shown in Figure A.1. First, the system figures out the current day's **temperature** according to the **temperature Curve**. That curve represents the temperature of the each day during certain period, where 50 days is the fixed temperature period. Then it picks up another parameter **useroffset** set manually by user, goes though the function **interpolation** which basically performs "**temperature** $+$ **useroffset**" action. Then it adds the **Cool setpoint** which is also set by user with the result of the function **interpolation**, and gets the temperature setpoint T_{set}.

This **cooling control system** then follows some actions to calculate the **cool demand**. It compares the result of the difference between T_{set} and the T_{mol} (which represents the real measured temperature of the current day) to 0, and chooses the smaller one. The **cooling demand** is the result of the smaller value be divided by a given constant K.

This **cooling demand** is only available for guide the **cooling relay when the cooling control system** gets the required **control signal**. According to the **Internal Cooling Rule, the control signal** is a boolean consequence of logic operations with 4 system parameters: **HumidityCool-Blocking**, **Vent Max**, **Vent AbsMax**, **VentLimitation**.

If the **control signal** is "False", the **cooling demand** will be passed to the **Cooing Relay**. Otherwise, if the **control signal** is "True" means no need to perform "cooling", 0 will be passed.

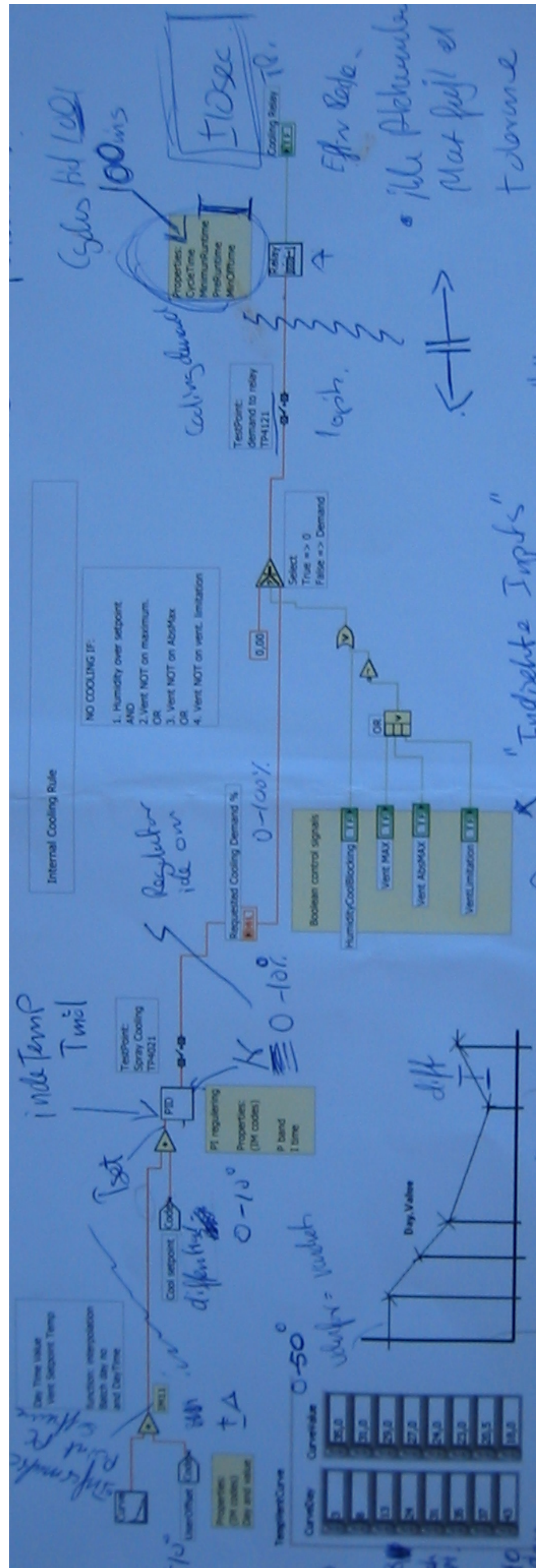


Figure A.1: Workflow of the Cooling Control System

A.1.2 Model Structure

We will model the Cooling Control System as a network of UPPAAL Timed Automata. Due to testing purpose, we propose separately modeling the specification of the system and its environment. Figure A.2 depicted the abstract input/output actions.

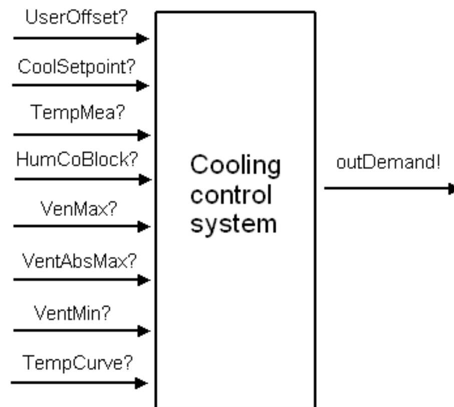


Figure A.2: Model Inputs and Outputs

The model consists of 5 concurrent components (UPPAAL timed automaton), 2 clock variables, 25 discrete integer variables and 12 channels. Figure A.3 shows the main components and their input/output communications. The following subsections will be organized in this way, first the constants, variant, channels used in the model will be shown, and then the behavior will be described, and all those will be done template by template.

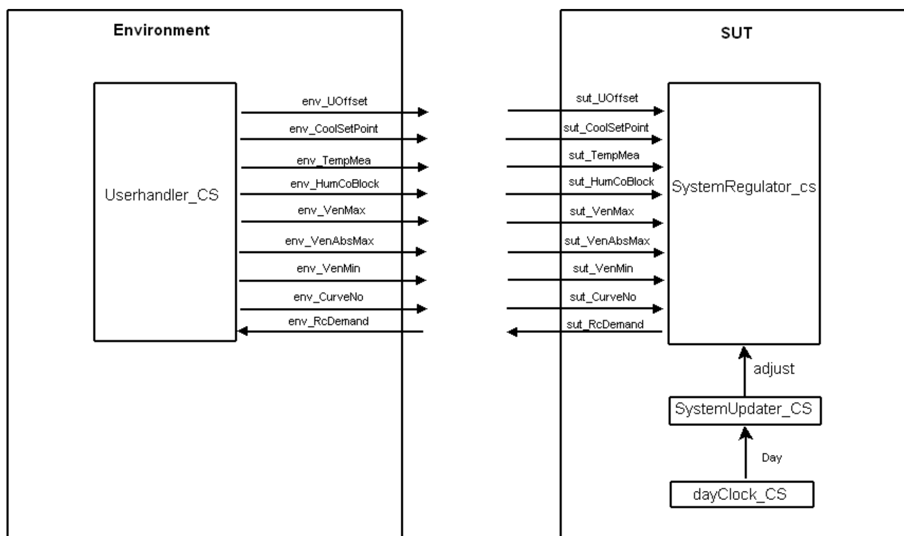


Figure A.3: Main model components

A.2 Model description

A.2.1 Environment

The automaton named **dayClock_CS** illustrated on Figure A.4 aiming at simulating the 24-hour per day. The variables used in this automaton are showed below. The local clock variable **x** is designed with intention to accumulate time units, when it reaches the value **hour24**, the global integer variable **Day** will be increased by 1. And this **Day** is designed for storing current day's number, that will be used to pick up current day's temperature from the **temperature Curve** later.

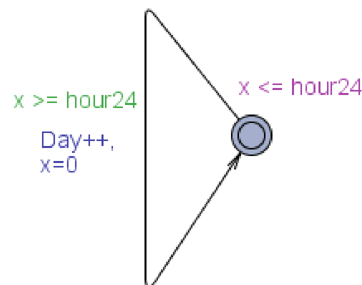


Figure A.4: Day Updater of the Cooling Control System

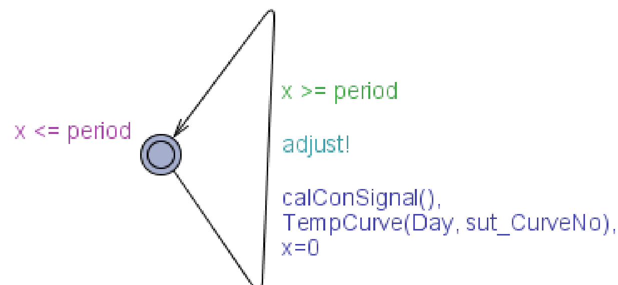


Figure A.5: Period Updater of the Cooling Control System

```
108 //−−−−−−−−−−−Global declaration for the "dayClock_CS"−−−−−−−−−−−−−−−−−
109
110 const int hour24 = 24*60*60; // in the model, we assume that each
111                              // time unit unit represents one second,
112                              // so here number of seconds in 24 hours
113 int Day = 1;
114
115 //−−−−−−−−−−−Local declaration for the "dayClock_CS"−−−−−−−−−−−−−−−−−
116
117 clock x;
```

Likewise, in order to periodically activate the cooling control system , the template in Figure A.5 named **SystemUpdater_CS** is designed. It uses local clock variant **x** to be the time units

accumulator, when **x** equals to **period** as it's shown in the following code section, it will synchronize with the automaton shown in Figure A.8 through channel **adjust**. When this transition is enabled, it will calculate the control signal **ConSignal** by means of **ConSignal()** function, and get the current day's temperature according to the Curve using **TempCurve(int day, int Number)** with two parameters, where **day**'s value passes from global variable **Day**, and **Number** is from **sut_CurveNo** represents which curve is chosen for this iteration of the cooling control system regulation.

```
119 //−−−−−−−−−−−Global declaration for the "SystemUpdater_CS"−−−−−−−−−−−−−−−−−
120
121 const int period = 10; chan adjust ;
122
123 //−−−−−−−−−−−Local declaration for the "SystemUpdater_CS"−−−−−−−−−−−−−−−−−
124 clock x;
125
126
127 void calConSignal () {
128
129 ConSignal = (env_HumCoBlock && ((!env_VenMax) || (!env_VenAbsMax)
130 || (! env_VenMin)));
131
132 }
133
134
135 void TempCurve(int day, int Number){
136
137 // in order to simplify the design of the model at the very beginning
138 //we assume the function to calculate the temperature of a Day equals to a constant e.g. y = 18,
139 // later we will propose the c code for the describing the temperature Curve
140 int y;
141
142 if (Number == 1) {
143   y = 18;
144   T_curve = y;
145 }
146
147 if (Number == 2) {
148   y = 15;
149   T_curve = y;
150 }
151
152 if (Number == 3) {
153   y = 12;
154   T_curve = y;
155 }
156
157 if (Number == 4) {
158   y = 9;
159   T_curve = y;
160 } return; }
```

As shown on Figure A.6, the automaton **Userhandler_CS** is used to simulate the users' way of setting the system parameters. Each time, it nondeterministically chooses one transition to take. Action in each transition plays in the same manner. Considering the synchronized channel labeled **HumCoBlock** as an example. Each time it is enabled, it will pick out one integer between its upper bound and its lower bound based on the UPPAAL select feature. In this transition, it has only two choices, either 0 or 1, then assigns the value to the **env_HumCoBlock** which is environment variable according to "separately model the environment part from SUT part" principle.

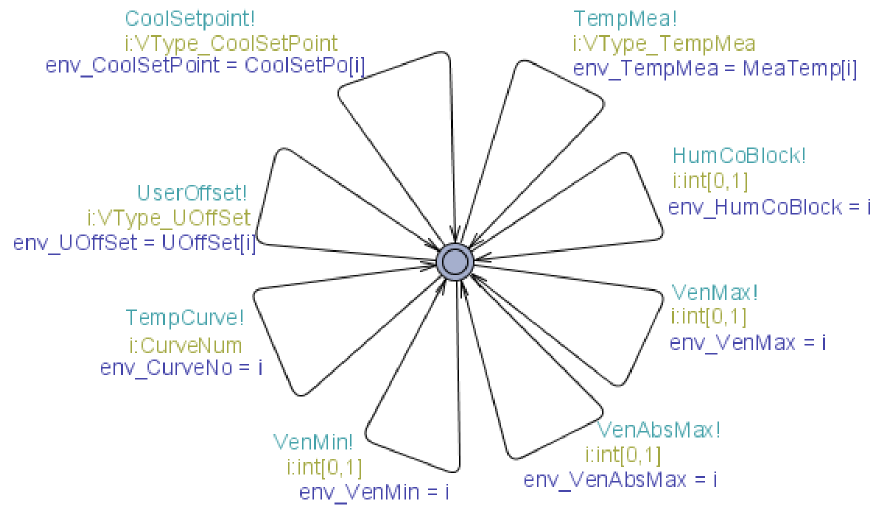


Figure A.6: Template for setting the system parameters

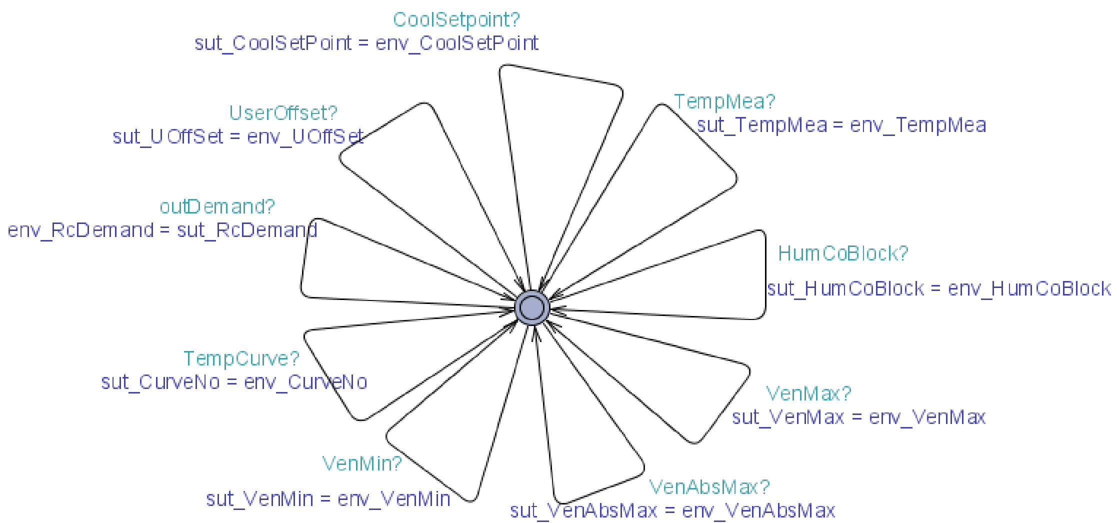


Figure A.7: Template for passing value of the system parameter

When the channel **HumCoBlock** is stimulated, as mentioned above, the automaton in Figure A.6 will synchronize with the automaton **IntUpdater_CS** in Figure A.7 through the channel with the same label. **IntUpdater_CS** performs the action of passing the value, either passes the value from the environment to the SUT, or in the inverse order. As it is shown in Figure A.7, after **env_HumCoBlock** being enabled, it sends **env_HumCoBlock**'s value to **sut_HumCoBlock**

according to the description of the update action.

```
161 //−−−−−−−−−−−Global declaration for the "Userhandler_CS"−−−−−−−−−−−−−−−−−
162
163 //−−−−userOffSet−−−−−
164
165 typedef int[−10, 10]   Type_UOffset;
166
167 // typedef  int [0,  20] VType_UOffSet;
168 typedef int [0,  4] VType_UOffSet;
169
170 const Type_UOffset   UOffSet[21]={−10, −9, −8, −7, −6, −5, −4, −3,
171                             −2, −1, 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10};
172 // const  Type_UOffset  UOffSet[5]={−10, −5, 0,  5,  10};
173
174 // In  the  model we add the  prefix  "env_"  to  all  the  environment
175 // variables ,  in  the  same manner, we add "sut_"  to  all  the
176 // SUT variables
177
178 Type_UOffset   env_UOffSet = 0  ;  Type_UOffset   sut_UOffSet = 0  ;
179
180
181 //−−−−measured temperature−−−−−
182
183 typedef int [0,  40]   Type_TempMea;
184
185 // typedef  int [0,  40] VType_TempMea;
186 typedef int [0,  2] VType_TempMea;
187
188 const Type_TempMea  MeaTemp[41]={0, 1, 2, 3, 4,  5,  6,  7,  8,  9,  10,
189                             11,  12,  13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,  27,
190                             28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,  40};
191
192 // const  Type_TempMea  MeaTemp[3]={0, 20, 40};
193
194 Type_TempMea  env_TempMea = 0;
195
196 Type_TempMea  sut_TempMea = 0;
197
198
199 //−−−−cool setpoint−−−−−−
200
201 typedef int [0,  20]  Type_CoolSetPoint;
202
203 // typedef  int [0,  20] VType_CoolSetPoint;
204 typedef int [0,2]  VType_CoolSetPoint;
205
206 const Type_CoolSetPoint   CoolSetPo[21]={0,  1,  2,  3,  4,  5,  6,  7,  8,
207                                   9,  10,  11,  12,  13,  14,  15,  16,  17,  18,  19,  20};
208 // const  Type_CoolSetPoint  CoolSetPo[3]={0,  10,  20};
209
210 Type_CoolSetPoint   env_CoolSetPoint  =  0;
211
212 Type_CoolSetPoint    sut_CoolSetPoint  =  0;
```

```
213
214
215 //---------temperature curve----------
216
217 typedef int [1, 4] CurveNum;
218
219 CurveNum env_CurveNo = 1;
220
221 CurveNum sut_CurveNo = 1;
222
223
224 //----------variant related to control signals-----------
225 int env_HumCoBlock = 0;        int sut_HumCoBlock = 0;
226
227 int env_VenMax = 0;            int sut_VenMax = 0;
228
229 int env_VenAbsMax = 0;         int sut_VenAbsMax = 0;
230
231 int env_VenMin = 0;            int sut_VenMin = 0;
232
233 int ConSignal = 0;
234
235 //----------------chan----------------------------------
236 chan TempMea; chan UserOffset; chan CoolSetpoint;
237
238 chan ControlSignal; chan HumCoBlock; chan VenMax;
239
240 chan VenAbsMax; chan VenMin; chan TempCurve;
```

A.2.2 SUT

The **SystemRegulator_CS** is shown in Figure A.8. It is used to calculate the amount of the cooling demand, to determine whether it is necessary to activate the cooling process via relay. This automaton is synchronized with the automaton shown in Figure A.5 through the channel called **adjust**. As we mentioned before, the automaton **SystemUpdater_CS** in Figure A.5 is activated periodically. Whenever this transition is enabled, it will also pass two results of the automaton in Figure A.5: the global variant **ConSignal** and the temperature **T_curve** from the Curve.

After a "committed" location which is signed "c", it calculates the temperature setpoint of the SUT, and the setpoint will later be stored in the global variable **T_set**. The calculation is performed through two functions. **InterpolationTemp(T_curve, sut_UOffSet)** which basically adds the value of the useroffset **sut_UOffSet** to the temperature **T_curve** gets from the Curve, obtains a result which is temporarily assigned to **T_set**. Then, it simply pluses the cool setpoint of the SUT with the **T_set** by means of **getTemSet(T_set, sut_CoolSetPoint)** and gets **T_set**.

After checking the value of the **ConSignal** it will be led to different status of the relay. if **ConSignal** == 1 means the system gets the command to turn off the relay, during that transition it also assigns the value of the **sut_RcDemand** == 0. Then it is led to a new location called **TurnOffRelay**. After outputting a synchronize signal **Roff** to the template **Relay_CS** in Figure A.9, until receiving the input signal **realDone** which comes from the **Relay_CS**, it will go back to the initial location.

Whereas, if the **ConSignal** equals 0, means the "NO COOLING" condition is not satisfied according to the system specification, the system calculates the demand of the cooling **sut_RcDemand** by means of the function **getRcDemand(T_set, sut_TempMea)**, and reaches to the location **TurnOnRelay**. If the **sut_RcDemand** == 0, it will transit to location **TurnOffRelay** as well. Otherwise, if the **sut_RcDemand** does not equal 0, it will send the synchronized signal **Ron** to trigger the relay, and waits until getting the signal **realDone** from the relay, and then goes back to the initial location.

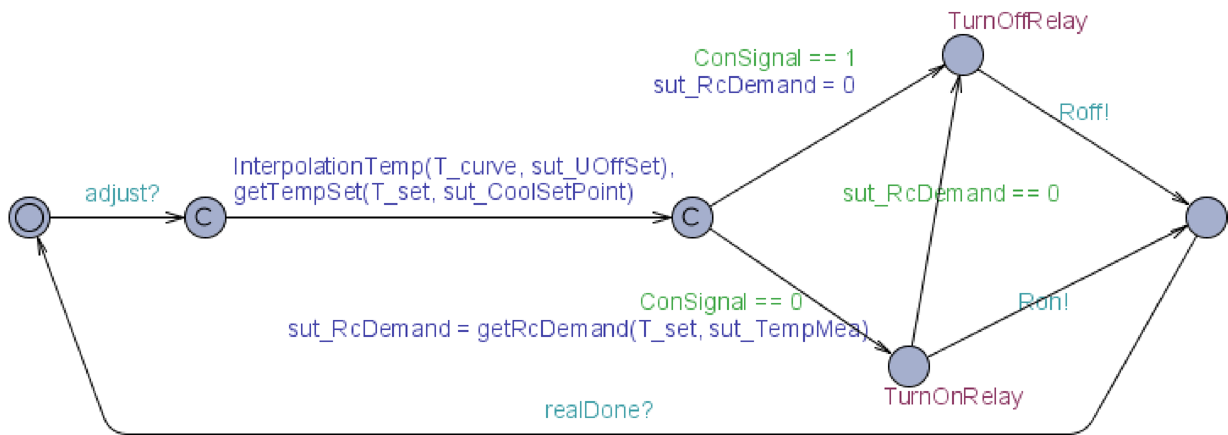


Figure A.8: Template for simulating the internal cooling system

```
242 //−−−−−−−−−−−Global declaration for the "SystemRegulator_CS"−−−−−−−−−−−−−−−−−
243
244 int T_curve = 0; int T_set = 0;
245
246 const int K=1;
247
248 //−−−−−−−cooling demand;−−−−−−−−−−−
249 int sut_RcDemand = 0;                    int env_RcDemand = 0;
250
251
252 //−−−−−−−channel−−−−−−−−−−−−−−−−
253 chan Roff;
254
255 chan On;
256
257 //−−−−−−−−−−−Local declaration for the "SystemRegulator_CS"−−−−−−−−−−−−−−−−−
258
259 void InterpolationTemp(int Tcurve, int uoffset){
260
```

```
261             T_set = 0;
262             T_set = Tcurve +  uoffset ;
263             return;
264 }
265
266 void getTempSet(int Tset, int coolsetpoint ){
267
268              T_set = Tset + coolsetpoint ;
269              return ;
270
271 }
272
273 int getRcDemand(int Tset, int Tmea){
274             int MinTemp;
275             int RcDemand;
276             T_set = Tset - Tmea;
277
278             if (T_set >= 0){
279                                  MinTemp = 0;
280                                 } else MinTemp = (0 - T_set) ;
281
282             RcDemand = MinTemp/K;
283             return RcDemand;
284 }
```

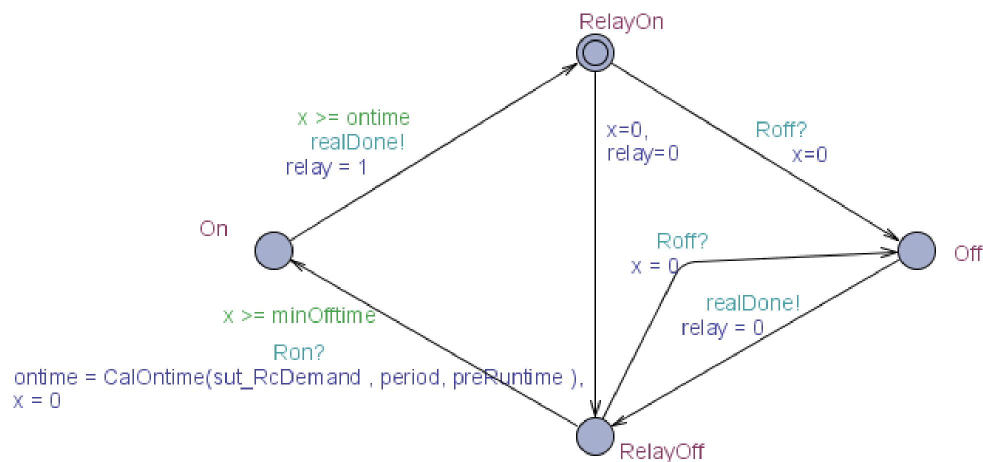


Figure A.9: Template for simulating the relay

The relay template in Figure A.9, is used to simulate the inner working process of the relay. It starts from the location **RelayOn**, if it gets the synchronized input signal **Roff**, it will be led to the location **Off**, during the transition it will also reset local clock x to 0. After being asked to turn **Off**, it will perform turning off the relay and pass the signal to the **SystemRegulator_CS** in Figure A.8 by means of the synchronized channel **realDone**, and arrives at the location **RelayOff**.

Otherwise, if it needs to be turned on, it requires to be turned off at first. The transition between the location **RelayOn** and the location **RelayOff** performs with intention to turn off the relay

before the **On** trigger being activated. It must wait for more than **minOfftime**, then can receive the synchronized input signal **Ron** in order to stimulate the **On** trigger. During that transition, it calculates **ontime** by means of the function **CalOntime** with three parameters **sut_RcDemand**, **period**, **preRuntime**, and resets the local clock **x** as well. After being turned on more than the required on time **ontime**, it sends **realDone** to the **SystemRegulator_CS**, and assigns relay has being turned on (**relay = 1**), and then goes back to **RelayOn**.

However, it may also receive the **Roff** signal to activate the **Off** trigger in the location **RelayOff**.

```
285 //-----------Global declaration for the "Relay_CS"-----------------
286 const int preRuntime = 1; const int minOfftime = 1;
287
288 int relay = 1;       //0 represents the "off" status of the relay
289                      //1 represents the "on" status of the relay
290 int ontime = 0;      // relay ontime
291
292 chan realDone;
293
294 //-----------Local declaration for the "Relay_CS"-----------------
295 clock x;
296
297 int CalOntime(int demand, int cycletime, int pretime) {
298     ontime = 0;
299     ontime = (demand * cycletime) + pretime;
300     return ontime;
301 }
```

A.3 Temperature Curve

As is described in the system specification, the temperature curve helps to calculate the current day's temperature setpoint. It is written in C.

The following we present the C code. It describes the way to draw temperature curve based on the eight previously given temperature points.

```
302
303 int main() {
304
305     int i, j, points=8;
306     int day;
307     float tempCurve[8][2]={{3, 35}, {8, 31}, {13, 29}, {24, 27}, {31, 24}, {35, 23}, {37, 21}, {43,
            18}};
308     float x0, y0, x1, y1, day, temp;                //"temp" represents temperature
309
310     /*
311     //get the points manually
312
```

```
313         printf ("The eight given points are:\n");
314
315         for(i=0; i<points; i++)
316         {
317
318             printf ("tempCurve[%d][0]=", i);
319             scanf("%f", &tempCurve[i][0]);
320
321             printf ("tempCurve[%d][1]=", i);
322             scanf("%f", &tempCurve[i][1]);
323
324         }
325         */
326         /* print out all the points */
327
328         printf ("The following is the first 100 points in the curve:\n");
329
330         for(i=0; i<100; i++)
331         {
332
333             day = i+1;
334             printf ("Day: %d", day);
335             printf ("     ");
336
337             if(day <= tempCurve[0][0]){
338
339                 temp = tempCurve[0][1];
340                 printf ("Temperature: %f\n", temp);
341
342             }
343
344             if(day >= tempCurve[7][0]){
345
346                 temp = tempCurve[7][1];
347                 printf ("Temperature: %f\n", temp);
348
349             }
350
351             if((day > tempCurve[0][0]) && (day < tempCurve[7][0])){
352
353                 j=0;
354
355                 while(day > tempCurve[j][0]){
356
357                     j++;
358
359                 }
360
361
362                 x0 = tempCurve[j-1][0];
363                 y0 = tempCurve[j-1][1];
364
365                 x1 = tempCurve[j][0];
```

```
366                 y1 = tempCurve[j ][1];
367
368                 temp = (y1−y0)∗day/(x1−x0) + (x1∗y0−y1∗x0)/(x1−x0);
369                  printf ("Temperature:␣%f\n", temp);
370
371             }
372
373         }
374
375         getch () ;
376         return 0;
377 }
```

A.4 Model checking

The model will be used for testing. Its properties need to be verified. As it is mentioned above, this model contains many nondeterministic actions. For example, one of the transitions with the channel named **UserOffset** in the template **Userhandler_CS** as shown in Figure A.6, each time when it's enabled, **env_UOffSet** nondeterministically chooses one integer value from its domain which starts from -10 to 10. So many nondeterministic reasons will cause magnifying of the explored state space undoubtedly, which may lead to unexpected result like unlimited consuming time and so on.

With the testing purpose, we design the model in flexible way. See Figure A.6, for instance, the channel labeled **UserOffset**. The selection action during this transition does not directly pick up integer from the domain of the variable of user offset.

As shown in Figure A.6, in the **UserOffset** transition, the variable **i** supposes to make the selection directly from -10 to 10, which is the domain of the user offset. Instead of that , we develop a new way. See the code section below, first, we define a new type called **Type_UOffset** represents the integer with the domain -10 to 10. And then we define another new type **VType_UOffSet** which is also a integer type but with the range between 0 to 20. An array **UOffSet** is bound with the type **Type_UOffset**. I.e, $UOffSet[0] = -10$, $UOffSet[20] = 10$.

So when performing the selecting action, the variable **i** is nondeterministically assigned with one of the integers according to the declaration of the **VType_UOffSet**. And it turns to decide which element in the array **UOffSet** to take by means of the **UOffSet[i]**, then the system assigns it to the environment variable **env_UOffSet**.

This design is suitable for the testing and verification purpose. It is really flexible to reduce the state space through changing the size of the **VType_UOffSet** into a small amount e.g. from 0 to 2 with 3 elements, so that the size of the array is lowered down as well. I.e, $UOffSet[0] = -10$, $UOffSet[4] = -10$.

```
378 //−−−−−−−−−−−Global declaration for the "Userhandler_CS"−−−−−−−−−−−−−−−−−
379
380
381 //−−−−userOffSet−−−−−
382
383 typedef int[−10, 10]   Type_UOffset;
384
385 typedef int [0, 4] VType_UOffSet;
386
387 // typedef  int [0, 20] VType_UOffSet;
388
389 // const Type_UOffset   UOffSet[21]={−10, −9, −8, −7, −6, −5, −4, −3,
390                                      −2, −1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
391 const Type_UOffset UOffSet[5]={−10, −5, 0, 5, 10};
392
393 Type_UOffset   env_UOffSet ;                  Type_UOffset  sut_UOffSet ;
394
395
396 //−−−−measured temperature−−−−−
397
398 typedef int [0, 2] VType_TempMea;
399
400 typedef int [0, 40]   Type_TempMea;
401
402 // typedef  int [0, 40] VType_TempMea;
403
404 // const Type_TempMea  MeaTemp[41]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
405                                    11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
406                                    26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40};
407
408 const Type_TempMea  MeaTemp[3]={0, 20, 40};
409
410 Type_TempMea  env_TempMea = 0;
411
412 Type_TempMea  sut_TempMea = 0;
413
414
415 //−−−−cool setpoint−−−−−−
416
417 typedef int [0, 20] Type_CoolSetPoint;
418
419 typedef int [0,2]  VType_CoolSetPoint;
420
421 // typedef  int [0, 20] VType_CoolSetPoint;
422
423 // const Type_CoolSetPoint   CoolSetPo[21]={0, 1, 2, 3, 4, 5, 6, 7, 8,
424                                           9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20};
425
426 const Type_CoolSetPoint CoolSetPo[3]={0, 10, 20};
427
428 Type_CoolSetPoint   env_CoolSetPoint;
429
```

```
430 Type_CoolSetPoint     sut_CoolSetPoint ;
```

The first query formula we proposed is "A[]not deadlock". It assumes that deadlock should never happen. For the memory reason, we run it in the cs.aau.dk server. But until now that query can not finish within one hours.

Bibliography

[1] Software testing. http://www2.umassd.edu/CISW3/coursepages/pages/cis311/ LectureMat/test/test1.html.

[2] C.Rankin. The software testing automation framework. *IBM Systems Journal*, 41(1), 2002.

[3] Daniela Damian. Practical software engineering. http://sern.ucalgary.ca/courses/CPSC/451/W00/Testing.html.

[4] R. de Vries, J. Tretmans, A. Belinfante, J. Feenstra, L. Feijs, S. Mauw, N. Goga, L. Heerink, and A. de Heer. Côte de Resyste in PROGRESS. In STW Technology Foundation, editor, PROGRESS *2000 – Workshop on Embedded Systems*, pages 141–148, Utrecht, The Netherlands, October 13 2000.

[5] Jean-Claude Fernandez, Claude Jard, Thierry Jeron, and Cesar Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146, 1997.

[6] Performance testing. http://en.wikipedia.org/wiki/Performance_testing.

[7] Stress testing. http://en.wikipedia.org/wiki/Stress_testing.

[8] Jan Tretmans and Ed Brinksma. Côte de resyste - automated model based testing.

[9] Peter A. Lindsay. A tutorial introduction to formal methods.

[10] A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment, 1999.

[11] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal: Status and future work. In Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans, editors, *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http://drops.dagstuhl.de/opus/volltexte/2005/326> [date of citation: 2005-01-01].

[12] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[13] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306, New York, NY, USA, 2005. ACM Press.

[14] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[15] Allard Robbert Kakebeen. Extension and formal verification of a distributed lift system using uppaal. Radbout Universiteit Nijmegen, August 2005.

[16] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99 - Concurrency Theory: 10th International Conference, Eindhoven, The Netherlands, August 1999. Proceedings*, volume Volume 1664/1999 of *Lecture Notes in Computer Science*, page 46. Springer Berlin / Heidelberg, February 1999.

[17] Marius Mikucionis. Uppaal tron. http://www.cs.aau.dk/ marius/tron/.

[18] Skov confidential document.