

FPGA Based Tester Tool for Hybrid Real-Time Systems

Jan Krákora and Zdeněk Hanzálek
{krakorj, hanzalek}@fel.cvut.cz

Czech Technical University in Prague,
Faculty of Electrical Engineering, Department of Control Engineering,
Karlovo náměstí 13, Prague 2, 121 35, Czech Republic

August 11, 2008

Abstract

This paper presents a design methodology for a hybrid Hardware-in-the-Loop (HIL) tester tool, based on both discrete event system theory, given by timed automata, and continuous systems theory, given by difference equations. It is implemented using an FPGA platform that guarantees speed enhancement, time accuracy and extensibility with no performance loss. We have focused on the implementation of a discrete event system, specifically timed automata into FPGA, and we have linked them with continuous systems implemented as filters in fixed point arithmetic. The paper shows a methodology, which employs widely used tools (Matlab, UPPAAL) as a user interface, and which implements the FPGA based tester tool.

keyword Hardware-in-the-Loop, Real-Time System Testing, Model checking, Timed Automata, FPGA, Hybrid System, System Control

1 Introduction

Hardware-in-the-Loop (HIL) applications are used by design and test engineers to evaluate and validate, e.g. vehicle components (electronic control units, etc.), during the development of new systems. Rather than testing these components in complete system setups, HIL allows the testing of new components and prototypes, so called Implementation Under Test (IUT).

Replacing the rest of the system by a model implemented in a computer (Tester tool) increases the flexibility and the rate of test scenarios. The physical components being tested respond to the simulated signals as if they were operating in a real environment. Therefore, they can not distinguish between

the signals sent by other physical components and signals provided by models running on a computer.

This paper presents the design methodology of a tester tool. The objective of the tool is to check the behavior of the IUT while simulating the behavior of the controlled system. The tool has to be able to automatically analyze the behavior of the IUT and to vary the parameters of the system so that the IUT is forced to operate in different conditions. In most applications, the controlled system incorporates complex dynamics of physical nature, usually captured as a continuous change of continuous states on one hand and as complex dynamics of logic nature conveniently modeled by discrete states and events on other hand. Therefore, our tester tool is a hybrid, i.e. it is based on both the discrete event system theory, given by timed automata [3], and continuous systems theory, given by difference equations [23, 24]. In order to avoid implementation errors, a high level of specification is required, so that the application expert can easily implement the system model, test cases and their analyses. The choice of such high level specifications, which are widely supported, enables us to execute a preliminary analysis (using Matlab/Simulink or UPPAAL) without incorporating any specific hardware, which simplifies the implementation of the tester tool.

Figure 1 shows a setup of the tester tool. The system model block emulates environment interacting with IUT. It is given by the timed automata and by the difference equations. The tester block includes test cases and checks the behavior of the IUT. It is collection of timed automata executing a test and monitoring specified properties.

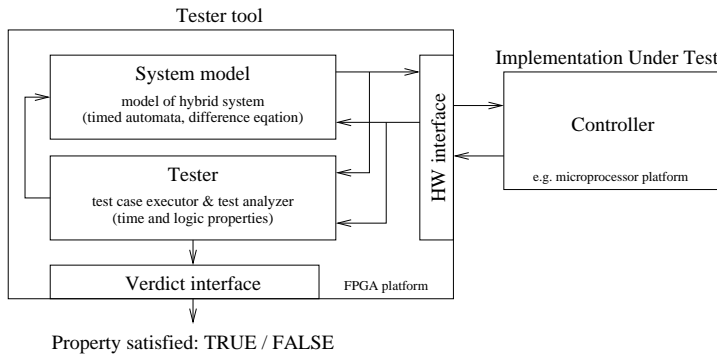


Figure 1: Hardware in the loop conception

Simulation tools for continuous systems (like Matlab-Simulink) and model checking tools for discrete event systems (like UPPAAL) are often used during the analysis and design phases of hardware and software designs (e.g. [17, 18]). Such model based designs usually lead to a modular structure, and the behavior of the modules is often analyzed separately in different tools, especially in the case of complex hybrid systems that are not tractable in polynomial time.

On the other hand, the testing phase of the model based design requires a compact solution in order to describe the complete system behavior. Therefore, in this article, we have followed this practice: we have assumed separate modules of the system, which are described and analyzed by appropriate tools (widely treated in the literature [7, 16]) and we have used these models as parts of the hybrid tester tool.

Our tester tool is implemented by using the FPGA platform that guarantees, not only speed performance, but also time accuracy, and has quite good extensibility with no performance loss as well. Compared to the operating system based platforms, the FPGA platform is able to achieve a much faster sampling frequency. Moreover, the FPGA platform is not affected by the rather complex behavior of the operating system services, interrupt handling, etc. In contrast to the typical sampling period of 10 μsec achieved in real-time operating systems like RTLinux [11] or OCERA [25], a sampling period of less than 100 $nsec$ can be achieved on the FPGA platform. More importantly, is the fact that, the FPGA platform has zero jitter since it is synchronous HW, and the separate parts do not influence each other. On the other hand, the operating system based platforms are well supported by widely used development tools.

1.1 Background

The hybrid tester tool presented in this article combines a discrete-event system, in the form of timed automata, and a continuous system in the form of difference equations.

Timed automata [3] are finite state automata, consisting of states (locations) and transitions, among these states. The transitions are extended by clocks that are used to specify quantitative time. Clocks are variables having non-negative real values, that can be reset. The difference between two clocks can be compared to a constant. In the initial system state, all clock values are zero, then all evolve at the same speed, synchronously with time.

The main feature of the timed automata approach, is that it can be verified using temporal logic [9]. It allows a user to check if any specification property of the model is satisfied or not. It is suitable in order to inspect the model due to deadlock, logical and timed conditions, for example.

The UPPAAL tool [5] used in this paper allows the user to design timed automata and to verify the automata using the mentioned temporal logic. Moreover, it offers an XML API for model export as well. The tool has been used as a design tool for timed automata that have been implemented into hardware.

Timed automata implementation into FPGA is inspired by [2]. The paper shows a way how to transform a timed automata (TA) into a program. The global time is represented by one running clock. For each clock of a timed automaton there is one integer variable (called digitized interpretation of TA). Such a variable is set to the value of the global time whenever the clock is reset. Consequently, the difference between this variable and global time represents the clock value. Each channel, used to synchronize the evolution of two timed automata, is replaced by one logic variable which is triggered synchronously with

all other channels in the model. A zeno behavior (a model has zeno behavior if it can take an infinite amount of actions in finite time) is not supported in our FPGA implementation.

The continuous part [24] of the tester tool is realized using difference equations that are able to represent a continuous system with periodic sampling (called discrete-time systems). The transfer function $G(z)$ of such a system influences the physical layout of the arithmetic and storage units in the FPGA.

The continuous part may have the following influence on the discrete part: when the continuous state variable reaches a certain threshold, a state in discrete part changes. The discrete part may have the following influence on the continuous part: when the state in discrete part changes, the parameters of continuous part are changed.

The tester tool was implemented in high performance FPGA, where several arithmetic operations can be executed within a period close to tens of *nsec* (Xilinx Virtex 4 XC4VFX12-F668-10C). As presented, a discrete-event system, as a set of timed automata, is designed in UPPAAL. A continuous system is designed as a transfer function using the Matlab control toolbox. Both a discrete-event and a continuous system are transformed into FPGA using the Xilinx System Generator, Matlab/Simulink toolbox (XSG). The toolbox can generate the final VHDL code for a target FPGA platform.

1.2 Related work

The related tools are briefly described in the following paragraphs, including comparison with our approach.

The TTG tool [14] is an off-line test generator (complete test scenarios and property results are computed a-priori and before test execution). The tool is based on the IF modeling language [8] and is able to generate a set of test cases for an IUT represented by a timed automata. Compared to our approach the TTG tool does not cover continuous systems.

The UPPAAL-TRON [12, 15, 20] is a tool for on-line testing (a single test primitive is generated from the model and at the time it is then immediately executed on the IUT) based on the discrete event system description using the timed automata. The tool both generates test cases to the IUT and checks test properties to be verified at the same time. The tool is executed on a UNIX platform, however, it can not guarantee the system time resolution of the FPGA. Moreover, it is not able to test an IUT using a hybrid system approach.

On the other hand, our tool only executes the test cases that are manually constructed, whereas TTG, UPPAAL-TRON and UPPAAL-COVER [12] generate the test cases automatically. In particular we assume the test cases to be based on the system requirements and to be written in the form of TA including fixed-point representation of continuous variables.

SoftCom [26] is an industrial HIL application that simulates industrial environment activities to react with an IUT, i.e. a programmable logic controller (called PLC). The application is based on discrete event systems based on state machines. However, SoftCom is an application developed for a Windows OS or

a non real-time Unix OS. The time resolution of these systems is approximately 100 *msec*, thus not very high. Moreover, it does not support hybrid system testing.

LabView FPGA [22] is a HIL application that allows a user to test the IUT running at a very high time resolution. The user can design their own HIL using the LabView environment supporting both discrete event and continuous systems. Such a HIL can be directly set into the FPGA. However, compared to our approach the tool misses the discrete-event system timed automata representation, and for that reason, an interconnection with model checking tool is not straightforward.

Another tool, CarMaker [13] is used for testing vehicle embedded control systems. The system is used for the development of vehicle embedded control systems interacting with a discrete event and a continuous system environment as well. Nevertheless, CarMaker, similarly to LabView FPGA, does not support timed automata.

1.3 Paper contributions

The main contributions of this paper are:

- (a) A methodology which incorporates widely used tools (Matlab used for controller design and simulation and UPPAAL for discrete event system design and model checking) as a user interface, and also implements the FPGA based tester tool. (see Subsection 2.1)
- (b) A novel approach to the implementation of timed automata into the FPGA. This approach enables the combination of formal model checking in UPPAAL with HIL on the FPGA.
- (c) Implementation of a high performance tester tool achieving a sampling period in the order of 100 *nsec*.

1.4 Outline

The paper's structure is as follows: Section 2 presents our hybrid HIL application methodology. The HIL discrete event system implementation using timed automata and continuous system implementation using difference equations are both presented there. Section 3 shows three case studies. The first study describes HIL implementation using the discrete event approach only. The second and third one show HIL implementation combining both discrete event systems and continuous systems. Section 4 deals with the conclusion of the paper and future work.

2 Tester tool architecture and implementation issues

HIL achieves a highly realistic simulation of equipment in an operational virtual environment. A typical HIL system includes an IUT and a tester tool. The structure of our HIL system is depicted in Figure 1. There are two main blocks, a tester tool and an IUT. The tester tool produces an output test signal to influence the IUT behavior and it reads the input signal to verify its reaction.

The tester tool consists of four parts.

System model: The *System model* specifies the system to be controlled by the IUT controller. It generates an output signal to the IUT (IUT input signals) and it reacts to the input signal from the implementation.

The system model implementation is given by the discrete event system description (e.g. timed automaton) or by the continuous system representation (e.g. difference equation). Both can be combined.

The timed automata models are designed using the UPPAAL tool [5] and transformed using the UPPAAL UTAP library [4]. The continuous system model, designed in Matlab/Simulink, is described by difference equations [17, 24].

Tester: The *Tester* block includes test cases and checks the behavior of the IUT. A test case generates a set of signals that are transmitted into a *System model*, influencing its states. Consequently, the tester analyzes both the input and output signals and verifies the properties of the IUT given by its IUT specification. The tester results are propagated via the *Verdict interface*.

The test case executor and test analyzer implementation are based on the discrete event system representation and on the continuous system as well. The test analyzer is able to affect both time and logic properties.

HW interface: The *HW interface* transmits data signals from the *System model* to the IUT and vice versa. It performs A/D and D/A conversion when appropriate, e.g. pulse width modulation (PWM).

Verdict interface: The *Verdict interface* is a block that interprets the test analyzer result for the user. It signals whether a given IUT property is satisfied or not.

2.1 Tester tool implementation

The tester tool has been implemented on the FPGA platform. The FPGA platform allows a designer to create its structure as needed. Moreover, it allows the user to implement complex real-time structures and implement parallel architectures. Thanks to the platform, the tester tool is fully configurable and time accurate.

The tester tool was designed using the Xilinx System Generator (XSG), an FPGA toolbox for the Matlab/Simulink IDE [28]. It makes it possible to design an FPGA infrastructure in a user friendly interface without deep knowledge of complex low level languages, like VHDL [29].

The tester tool implementation issues, like state machines, timed automata or channels using the XSG, are presented in the following text.

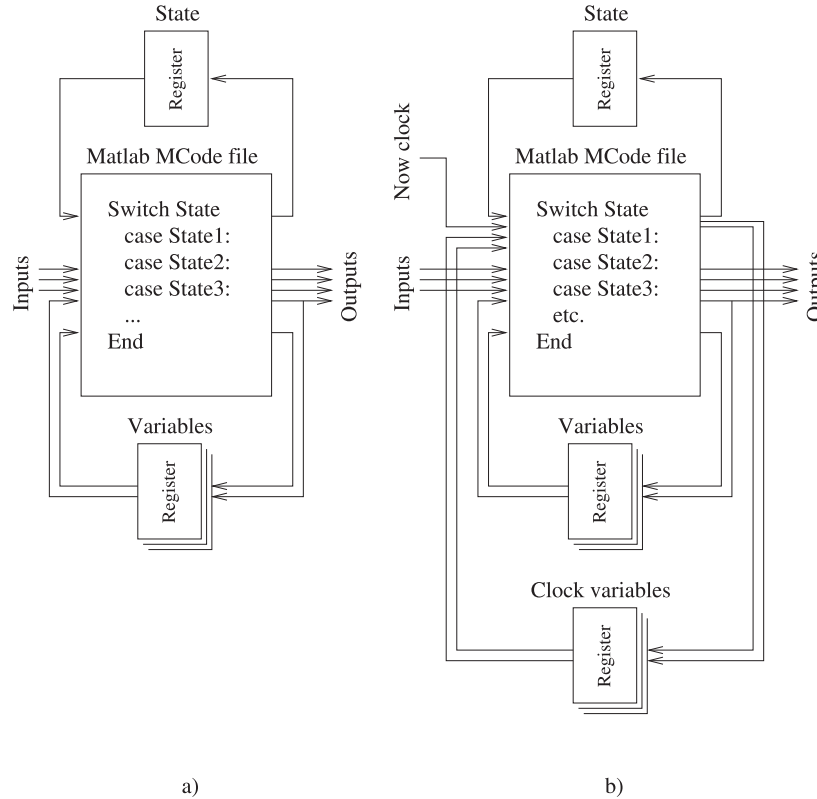


Figure 2: State machine and timed automaton implementation in XSG

2.1.1 State machine implementation

The state machine structure, implementable in XSG, is shown in Figure 2a. The automaton consists of the state register, the Matlab MCode file (MCode) and the set of local variables.

The state register stores the value of the actual automaton state. The state value is updated depending on the condition set in the MCode.

The MCode is a file containing a Matlab function with a *Switch-Case* statement in Matlab like notation. The statement structure is given by the structure of an accordant automaton.

```

1 function [state_new, channel_req, clk_new] =
2   TA1(state, now_clk, channel_ack, clk)

4 % Variable initialization
5 state_new = state;
6 channel_req = 0;
7 clk_new = clk;

9 % Timed automaton state machine
10 switch state
11   case 1 % State1 - the initial state
12     if( channel_ack == 1 )
13       channel_req = 0;
14       clk_new = now_clk;
15       state_new = 2;
16     else
17       channel_req = 1;
18     end
19   case 2 % State2
20     if( (now_clk - clk) == 2 )
21       state_new = 1;
22     end
23   otherwise
24     % an error occurred
25     state_new = -1;
26 end

```

Figure 3: MCode listing of timed automaton TA1 (see Figure 6)

Each *Case* statement represents the location and each *if* statement in the *Case* represents the state to state transition constraint.

The registers are used to store an eventual automaton variable since it can not be stored in the MCode. Because the variable register data type can differ from the required input data type of the MCode, such variables have to be converted to properly data type.

An MCode file content example is depicted in the listing in Figure 3.

2.1.2 Timed automata implementation

Our timed automata implementation is inspired by [2]. The paper shows the transformation of timed automata into a program. The global time is captured in the variable `now_clk`, the only running clock. Even though the clock is a real value in the timed automata, it may be represented by an integer value when assuming periodic sampling. For each clock of the timed automaton there is one static variable `clk`. Such a variable is set to the variable `now_clk` whenever the clock is reset. The difference between `now_clk` and `clk` represents the clock value. Each channel, used for synchronizing the evolution of two timed automata, is replaced by one logic variable which is triggered synchronously with all the other channels in the model.

The TA implementation into the FPGA is similar to the state machine implementation described in the previous section. Moreover, the `now_clk` generator and one clock register, per each timed automaton clock variable, were added.

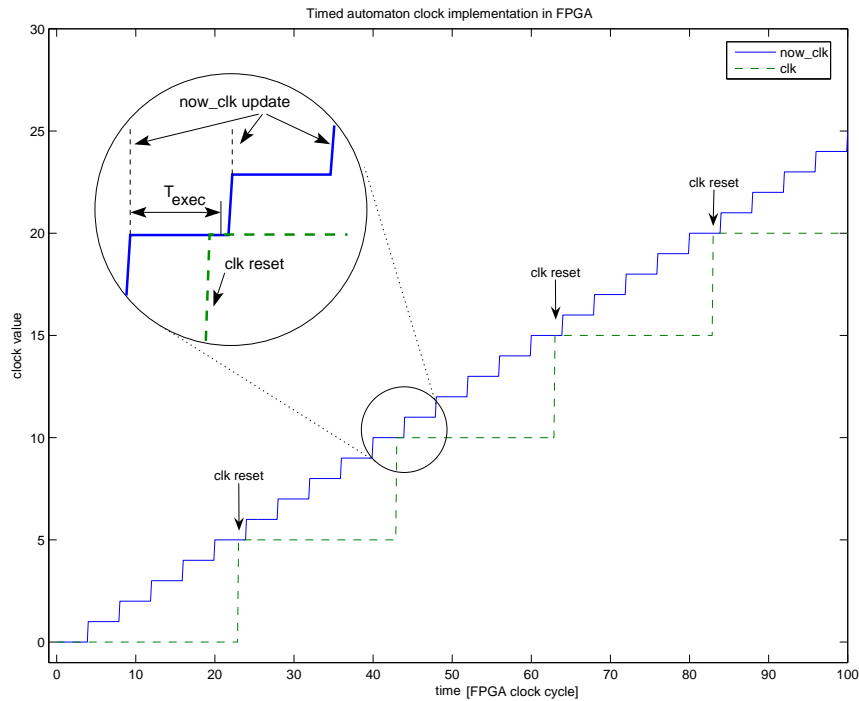


Figure 4: Timed automaton clock implementation

Each clock is updated depending on the MCode structure.

The clock evaluation example is depicted in Figure 4. It presents the `now_clk` and `clk` history of variables and it shows an example of the periodic reset of the variable `clk`. The `clk` sampling period is based on `now_clk`. Its reset time is given by timed automata constraints in the MCode.

The variable `now_clk` is incremented each sampling period. The sampling period is given by the length of the FPGA clock cycle and by the number of the FPGA clock cycles needed to evaluate the continuous part and the discrete part. The length of the FPGA clock cycle is given, namely, by the complexity of the arithmetic operations performed in the continuous part (see the chain of the multiplication and addition units in Figure 7) and the complexity of buses in the discrete parts. The shortest possible FPGA clock cycle is about 50 nsec , assuming the continuous system of the third order and 16-bit fixed point logics on the Xilinx Virtex 4 XC4VFX12-F668-10C.

The existence of the urgent locations and channels in the timed automaton requires repetitive calls of the corresponding timed automaton function (e.g. Figure 3). For example, when a sequence of three urgent locations is assumed,

then at least four repetitive calls are needed. Therefore, the sampling period has to be greater than T_{exec} , the time needed to update all FPGA internal processes. For example, the sampling period of the timed automata is set to 4 clock cycles in Figure 4.

The implementation of the timed automaton location-to-location transition is bounded by a time constraint interval and can be resolved by an approach based on the principle that only one sample from the interval is chosen. In timed automata, while using invariants and guards, a transition can be taken within the given time interval. This non-deterministic feature, related to the system parameter for which only the lower and upper bounds are known, can be handled in the tester tool while using a stochastic approach. In Figure 3, the stochastic approach can be implemented by using a function `randTI()` that generates a random value given by a probability distribution function on the time interval. In the case of a normal distribution on the interval $[2, 4]$ (invariant $clk \leq 4$ and guard $clk \geq 2$ in Figure 6a, code line 20 should be changed into `if((now_clk-clk) == randTI(normal,2,4))`. Such function with normal probability distribution function can be realized by a linear feedback shift register [21] using [27].

Model-checking by the timed automata is an integer based formalism. For that both clocks and timed automata variables are interpreted as integers. The continuous variables have to be represented in appropriate units, so that their integer representations offer required precisions.

Channel implementation A primitive implementation of channels for the synchronization of state automata is shown in Figure 5. The figure presents two automata using one channel to coordinate their activities.

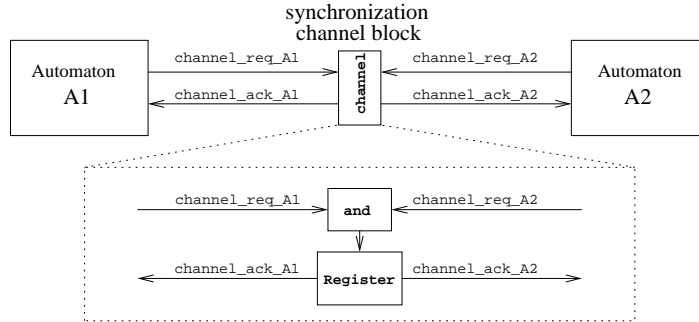


Figure 5: Channel implementation

Each automaton uses two variables. The first variable, `channel_req_Ax`, activates the synchronization process, the second one, `channel_ack_Ax` acknowledges to the automaton that the synchronization process is completed.

The channel structure is depicted in the dotted box in Figure 5. The structure uses the one logic operand AND and one register to avoid an algebraic loop.

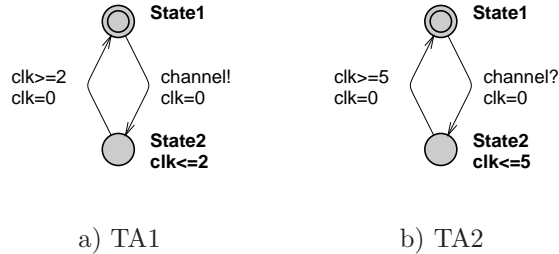


Figure 6: Timed automata example in UPPAAL like notation

Timed automata example An example of the timed automata transformation into XSG is presented in the following paragraphs.

Figure 6 shows two timed automata in UPPAAL like notation. Each automaton uses a local clock variable `clk`. Both automata are synchronized by the synchronization channel `channel`.

The automaton *TA1* has two locations **State1** and **State2**. *TA1* waits for *TA2* in the initial location **State1** using `channel`. If the synchronization event occurs the `clk` is reset to zero and *TA1* passes into location **State2**. The automaton waits here for 2 time units. It is given by the constraint in the invariant and transition **State1** \rightarrow **State2** guard. After that, the clock `clk` is reset and *TA1* returns to **State1**.

In a similar way, *TA2* starts in the initial location **State1**. After synchronization, it waits in the **State2** location for 5 time units and returns to the **State1** location.

The transformation from *TA1* into MCode is listed in Figure 3.

The *TA2* transformation into MCode is similar to *TA1*. It differs only in code line 20 where `(now_clk - clk)` is compared to 5 sampling periods.

In order to retrieve a sequence of states leading to a given state, a diagnostic trace can be implemented by using a circular buffer that stores a time stamp, state information and variable at each state turn.

2.1.3 Discrete-time system implementation

The continuous system model is implemented as a digital filter [17, 23]. The general form of the filter transfer function between the output $Y(z)$ and the input $X(z)$ is given by:

$$G(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + \dots + b_m z^{-m}}{1 + a_1 z^{-1} + \dots + a_n z^{-n}} \quad (1)$$

where m is the maximal order of the numerator $Y(z)$ and n is the maximal order of the denominator $X(z)$. Without loss of generality, we suppose the system to be pure, i.e. $n \geq m$. With respect to the transfer function of z^{-1} that represents a delay of one sampling period, the equation can be modified into:

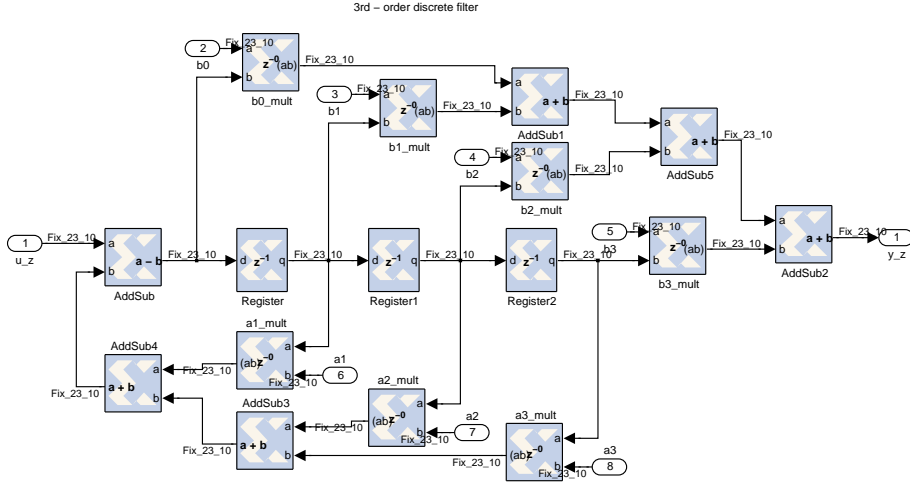


Figure 7: Transfer function of the 3^{rd} order system implemented in Matlab XSG

$$y(k) = b_0x(k) + b_1x(k-1) + \dots + b_mx(k-m) - a_1y(k-1) - \dots - a_ny(k-n) \quad (2)$$

The $y(k)$ is the k -th sample of the output and $x(k)$ is k -th sample of the input. It can be directly implemented into an FPGA platform using delay elements, adders and multipliers.

For example, a 3^{rd} order transfer function can be implemented by using XSG as shown in Figure 7.

An example of the transfer function response to the input step function is depicted in Figure 8. The figure shows the $y_s(k)$ of the transfer function implemented in the standard Matlab/Simulink, $y_x(k)$ of the transfer function implemented in the XSG toolbox and $y_f(k)$, the response of the transfer function implemented in the FPGA. The XSG $y_x(k)$ deviation is given by the fixed point logic accuracy, preset in the toolbox arithmetic. The XSG simulation is bit-exact to the FPGA implementation, which means the simulated system behavior is identical to the system behavior finally implemented in the target FPGA platform (samples of $y_x(k)$ and $y_f(k)$ are completely identical in Figure 7).

3 Case studies

This section shows three case studies. The first case study is composed using the timed automata. The second case study combines both timed automata and difference equation implementations. The third case study is more complex while focusing on scalability and flexible reuse of the components of the second case study.

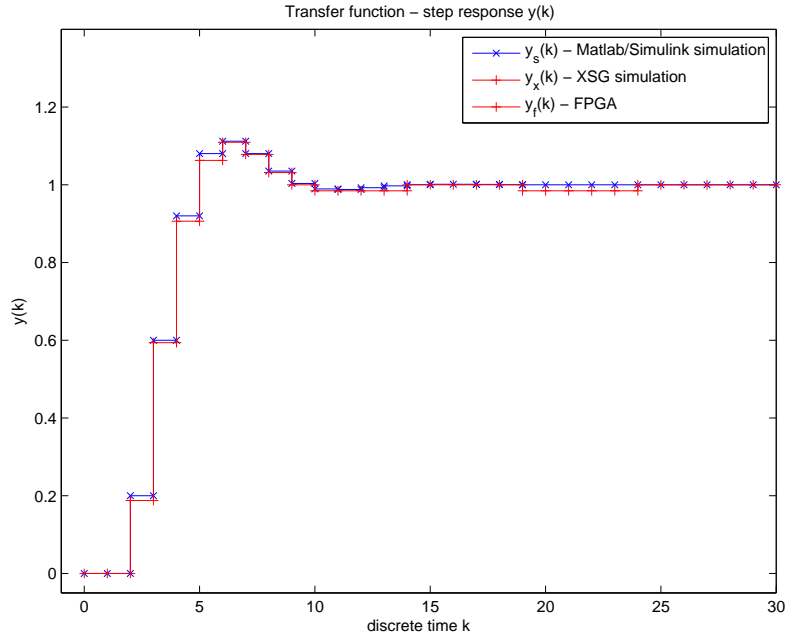


Figure 8: Output function of the system model as a reaction to the step function

3.1 Case study 1 - Airbag control

This case study deals with the on-line testing of an electronic control unit (ECU) that is responsible for airbag control. An airbag is a flexible membrane or envelope, inflatable to contain air or some other gas. Airbags are most commonly used for cushioning, in particular after very rapid deceleration, in the case of an automobile collision. The airbag system consists of three basic parts, crash *sensor*, a *controller*, and an *inflator device*. It is a highly safety critical application.

The case study structure is illustrated in Figure 9. The *sensor* detects a car impact. If it detects the impact (a deceleration crosses over a safety limit) the *controller* should execute the airbag deployment by the *inflator*. The *controller* has the responsibility to activate the inflator within a specified time after the *sensor* alarm occurs.

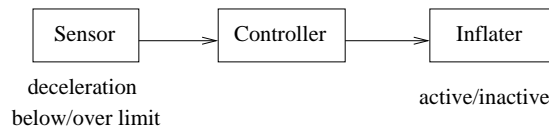


Figure 9: Case study 1: Airbag control system

A very simple *controller* algorithm, the *sensor* functionality and the *inflator*

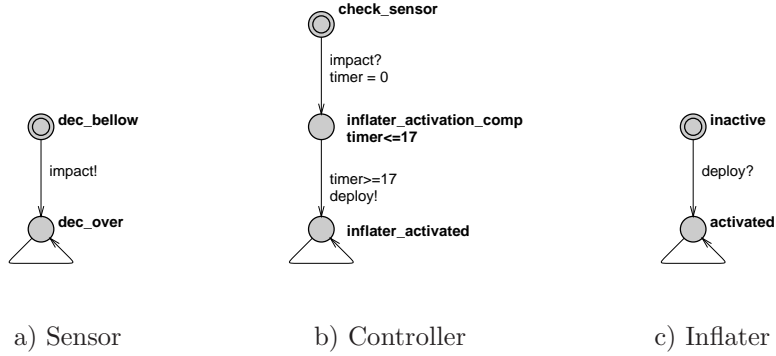


Figure 10: Case study 1: Acceleration sensor, Controller and Inflation device timed automata

functionality are presented in Figure 10.

The goal is to validate the controller whenever it activates the *inflator* within a specified time after the *sensor* event.

The case study implementation scheme of HIL is presented in Figure 11. It consists of a system model simulating the *sensor* and *inflator* activity, the *controller* under test and tester containing three property blocks and the test case presented below.

The system model is composed of timed automata representing the *sensor* and the *inflator* (see Figure 10a and 10c). The controller under test including its simple dynamics depicted by timed automaton (**inflator_activated** state is reached at most 17 sampling periods after impact) is shown in Figure 10b. These timed automata are transformed into XSG using the methodology depicted in Subsection 2.1 including the synchronization channels **impact** and **deploy**.

The tester consists of the test case executor and the set of properties to be verified. The test executor activates the system model block and all property blocks as presented below. The tester and the properties are given by timed automata, which have a sampling period of 100 *nsec*.

There is just one test case implemented for this case study. The test case activates the system model *sensor* at 2 μsec . All property blocks are activated at that time, the property results are analyzed at 5 μsec .

There are three properties to be verified. Each property is transformed into the timed automata model as explained in Subsection 2.1. The list of properties is as follows:

Property 1 “*The controller always activates the inflator within 2.0 μsec from the moment when the sensor is activated.*”. For the property timed automaton see Figure 12a.

Property 2 “*The controller always activates the inflator within 1.5 μsec from when the sensor is activated.*”. For the property timed automaton see Figure 12b.

Property 3 “Does the state in which the inflater is activated exist?”. The accordant automaton is depicted in Figure 12c.

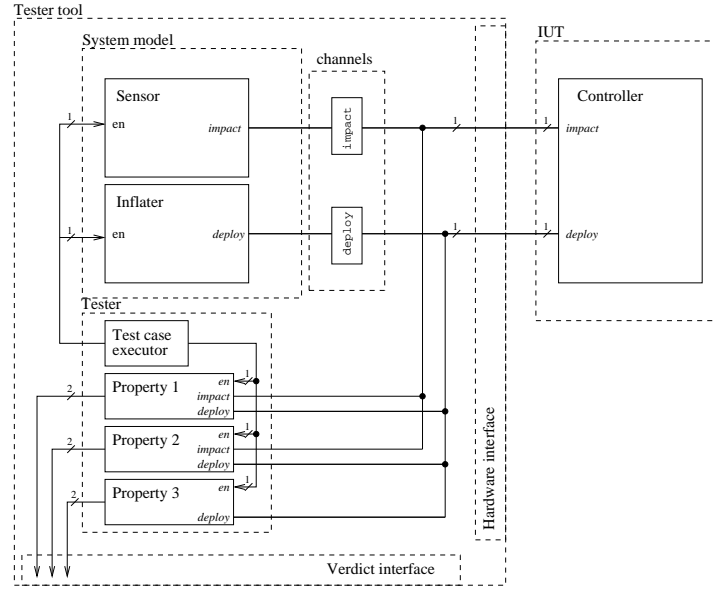


Figure 11: Case study 1: Implementation scheme of the case study HIL

The *controller* under test is implemented using the timed automata approach (the controller timed automaton `now_clk` update time is $0.1 \mu\text{sec}$). Its structure is depicted in Figure 10b. When the *sensor* is activated, the *controller* receives the synchronization signal `impact` (consequently, the transition `check_sensor` \rightarrow `inflater_activation_comp` in Figure 10b is activated). Furthermore, it computes a reaction and it activates the *inflater* by the signal `deploy` within 17 time units (`inflater_activation_comp` \rightarrow `inflater_activated`).

Figure 13 shows the state evolution of the *controller* and the *sensor* in the upper plot and evolution of each verified property in the bottom one.

The *sensor* is activated at $2 \mu\text{sec}$. The *controller* evolves at that time in `check_sensor` \rightarrow `inflater_activation_comp` and switches to the `inflater_activated` after $1.7 \mu\text{sec}$ (computation time of the controller). The bottom plot of Figure 13 shows the following results:

- Property 1** The property is satisfied. The inflater is activated at $1.7 \mu\text{sec}$.
- Property 2** The property is not satisfied. The inflater is not activated within $1.5 \mu\text{sec}$.
- Property 3** The property is satisfied. The `inflater_activated` state occurs at $1.7 \mu\text{sec}$.

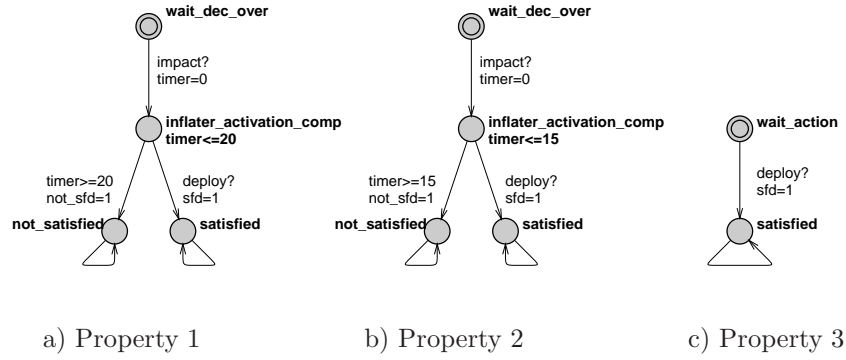


Figure 12: Case study 1: Properties to be verified

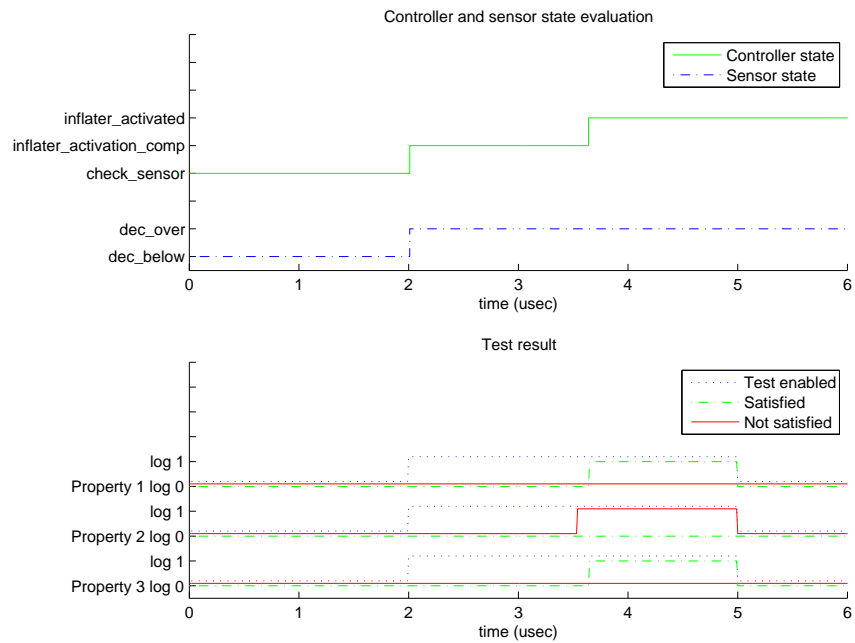


Figure 13: Case study 1: Tester output

Therefore the tests of the airbag controller have shown its ability to deploy the inflater within $2 \mu\text{sec}$ as required.

3.2 Case study 2 - Motor control

This case study deals with testing of controllers given by continuous system representation. It presents the combination of a continuous system (system model, controller) with a discrete event system (property blocks, test executor).

The study tests several properties of the servo-system with or without friction to the shaft. The system consists of a motor and a controller. The controller operates the motor to a reference angular velocity ω_{ref} .

The motor behavior can be described by the following equations:

$$\begin{aligned}\frac{\partial i}{\partial t} &= -\frac{R}{L}i - \frac{\xi}{L}\omega + \frac{1}{L}u \\ \frac{\partial \omega}{\partial t} &= \frac{\xi}{J}i - \frac{1}{J}m_z\end{aligned}\quad (3)$$

In this system, R is the terminal resistance, L is the terminal inductance, ξ is the torque constant, ω is the angular velocity, u is the applied armature voltage, i is the armature current, J is the equivalent moment of inertia of the motor and load referred to the motor and m_z is the torque load given by the friction to the shaft, for example.

Using the motor Maxon A-max 26 [19], the transfer function $\frac{\omega}{u}$, with $m_z = 0 \text{ Nm}$ and $T_s = 1 \text{ msec}$, is:

$$G(z) = \frac{2.652z + 0.3143}{z^2 - 0.9202z + 0.0001003} \quad (4)$$

when the servo-system has the friction to the shaft $m_z = 0.1 * 10^{-3} \text{ Nm}$ the transfer function, with $T_s = 1 \text{ msec}$, is:

$$G_f(z) = \frac{2.559z + 0.2924}{z^2 - 0.8483z + 9.249 * 10^{-5}} \quad (5)$$

There are two controllers tested separately in this case study. The first controller is given by the transfer function:

$$G_{C1}(z) = \frac{0.1055z - 0.0939}{z - 1} \quad (6)$$

with sampling period $T_s = 1 \text{ msec}$. The transfer function of the second controller is:

$$G_{C2}(z) = 0.2 \quad (7)$$

The goal is to test, whether a given controller is able to operate both systems given by the transfer Functions 4 and 5. Each controller has to be able to satisfy the below mentioned properties in both operating conditions, i.e. with or without friction at the shaft.

It is required to verify the following:

- The motor angular velocity should be stabilized in the interval (9, 11) $Rad s^{-1}$ within 15 msec when the operating conditions are changed (i.e. when the reference velocity is changed from speed 0 $Rad s^{-1}$ to a non zero velocity or when the friction m_z has changed).
- The motor angular velocity should not exceed 12, when the test is enabled. $Rad s^{-1}$.

The structure of the HIL is depicted in Figure 14. All the tester tool parts except the interfaces have been implemented using XSG. The speed of the continuous part is degraded by the performance of ADC and DAC. The problem with DAC can be partially solved while implementing Pulse Width Modulation (PWM) in FPGA for a specific controlled system (e.g. in the servo-motor case study, the motor behaves like a low pass filter with a time constant which determines the period of the PWM). The problem with ADC, in the servo-motor case study, can be solved by implementation of the IRC speed detector in FPGA.

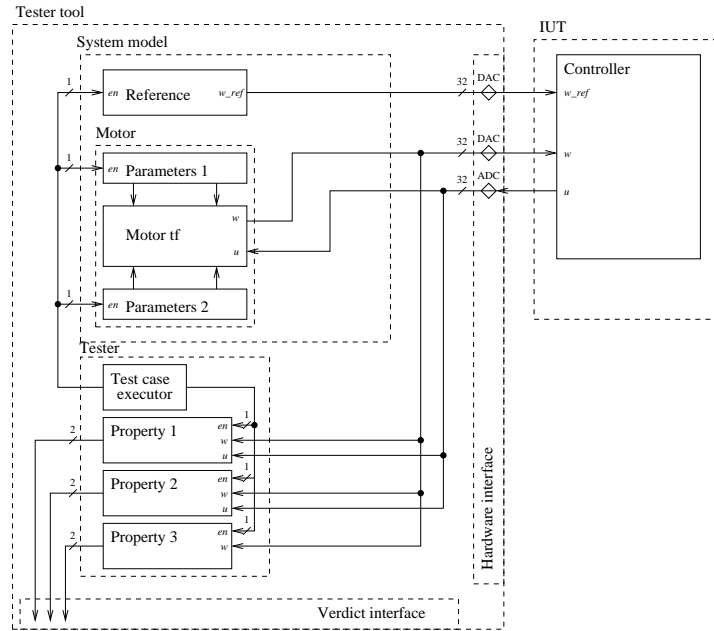
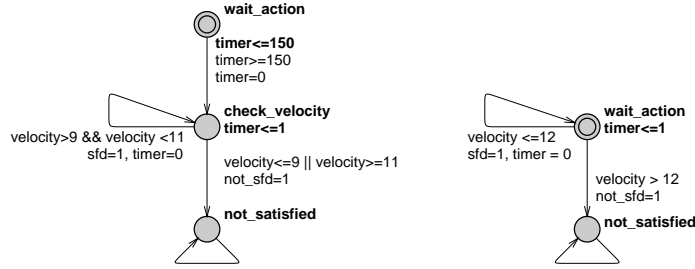


Figure 14: Case study 2: Tester structure

The controllers are implemented using the Matlab/Simulink discrete transfer function blocks. They are directly connected to the tester tool via XSG in/out interfaces.

The *tester tool* consists of the parts depicted in Section 2. All the necessary details of each part are given in the following paragraphs.

The *System model* includes the *Reference* block and *Motor* block. The *Reference* block sets up the required motor angular velocity ω_{ref} . The *Motor* block



a) Properties 1 and 2

b) Property 3

Figure 15: Case study 2: Properties timed automata

influences the tested controller. It is given as a system with time varying parameters. Based on the test executor structure (may be conveniently depicted by timed automaton), the test executor changes parameters of the transfer function from (4) to (5). The *Motor* transfer function sampling period is set up to 1 μsec (using XSG up sampling and down sampling blocks).

The *Tester* consists of a *Test case executor* and three *Property* blocks to be verified. The *Test case executor* simulates a situation when the *motor* is started from 0 Rad s^{-1} to $\omega_{ref} = 10 \text{ Rad s}^{-1}$ at 10 msec (see Figure 16, the upper chart). The motor brakes are applied so the friction at the motor shaft is $m_z = 10^{-4} \text{ Nm}$ at 60 msec . Testing is finished at 110 msec . The motor behavior within the first interval [10,60] msec is given by Equation 4, the behavior in the interval [60,110] msec is given by Equation 5. Otherwise the behavior is given by $G(z) = 1$.

For that purpose, the *Test case executor* simultaneously enables the *Reference* block to propagate the ω_{ref} to the tested controller, it enables *Parameters 1* and it activates the *Property 1* and *Property 3* blocks as well (see Figure 16, the bottom chart). At 60 msec the *Test case executor* switches the *motor* parameters from *Parameters 1* to *Parameters 2*, and consequently the *Property 2* block is activated.

The properties to be verified are derived from the case study requirements. Each property is implemented using timed automata as presented in Subsection 2.1. The *Property* block sampling period is 100 μsec . The description of each property and equivalent timed automata are the following:

Property 1 “The motor angular velocity is always stabilized in the interval (9, 11) Rad s^{-1} within 15 msec .” Figure 15a shows the property automaton. The property test is activated at 10 msec and it is analyzed at 60 msec .

Property 2 “The motor angular velocity is always stabilized in the interval (9, 11) Rad s^{-1} within 15 msec .” The property timed automaton is identical with the property 1. The property test is activated at 60 msec and it is analyzed at 110 msec .

Property 3 “The motor angular velocity does not exceed 12 Rad s^{-1} .” The automaton is shown in Figure 15b. The property test is activated at 10 msec and it is analyzed at 110 msec .

The controllers under test are given by the transfer functions $G_{C1}(z)$ and $G_{C2}(z)$ shown in Equation 6 and 7, respectively.

The test results for controller 1 are depicted in Figure 16. The upper plot shows the motor input voltage u given by the controller operation, the motor angular velocity output ω and reference velocity ω_{ref} . The bottom plot depicts the *Verdict interface* output.

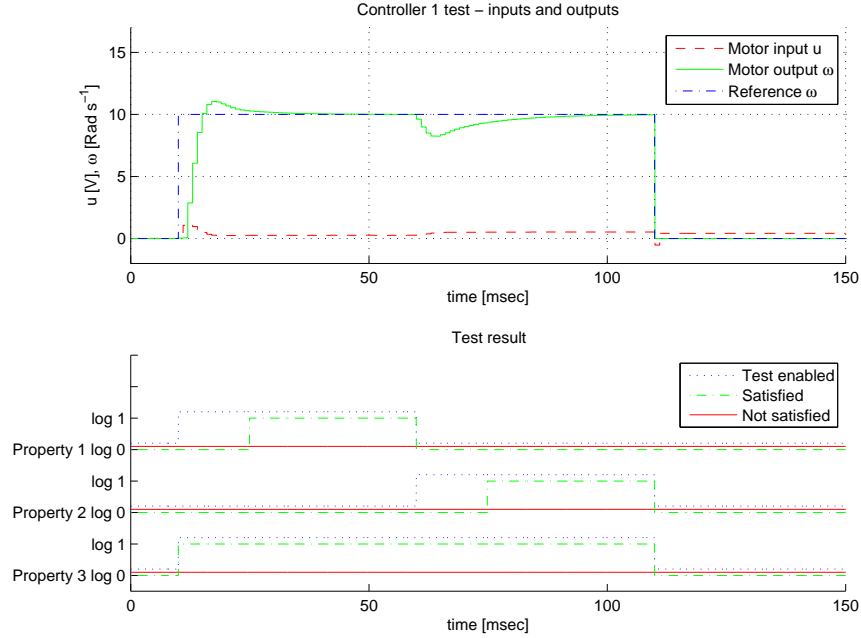


Figure 16: Case study 2: HIL inputs and outputs for controller 1

The model checking results are as follows:

Property 1 The property is satisfied. As presented in Figure 16, the controller controls the velocity to the required reference within 15 msec from the beginning of the test. The test of property 1, enabled within the time interval $[10, 60) \text{ msec}$, shows the property is satisfied, i.e. the location **not_satisfied** in Figure 15 is not reached.

Property 2 The property is satisfied. Although the motor angular velocity is influenced by friction to the shaft at 60 msec , the controller is able to keep the velocity at the required value. The test of property 2, enabled within the time interval $[60, 110) \text{ msec}$ shows that the property is satisfied.

Property 3 The property is satisfied as well. During the test interval $[10, 110] msec$ the motor velocity does not overshoot the $12 Rad s^{-1}$ boundary.

The test results for controller 2 are presented in Figure 17. The property results are the following:

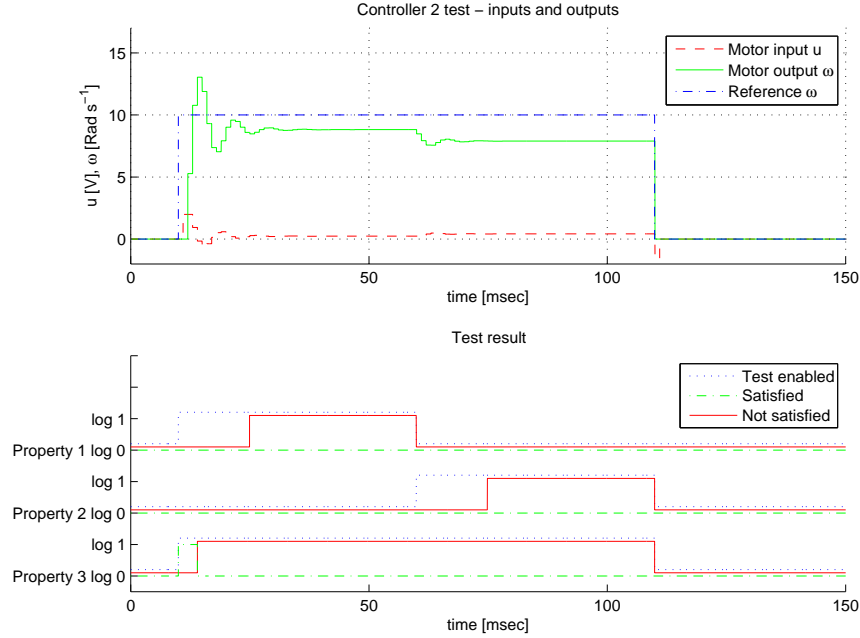


Figure 17: Case study 2: HIL inputs and outputs for controller 2

Property 1 The property is not satisfied. As presented in Figure 17, the controller is not able to stabilize the required velocity. The test of Property 1, enabled within the time interval $[10, 60] msec$, shows the property is not satisfied.

Property 2 The property is not satisfied. The controller is not able to operate the velocity in the required interval. The test of Property 2, enabled within the time interval $[60, 110] msec$ shows the property is not satisfied because the value is below the required value interval $(9, 11) Rad s^{-1}$.

Property 3 The property is not satisfied as well. During the test interval, the velocity overshoots $12 Rad s^{-1}$ boundary at $4 msec$ from the beginning of the test.

3.3 Paper machine case study

This subsection deals with a case study focusing on the scalability and flexible reuse of the components in the second case study.

The Fourdrinier Machine [6] is the basis for the most modern paper making, and it accomplishes all the steps needed to transform a source of wood pulp into a final paper product. Figure 18 shows a simplified structure of all major sections of the machine.

Our case study is related to the dryer section which dries the wood pulp as it turns into pure paper. The dryer section includes several tens of steam cans. The whole machine is actuated by several different motors. The line speed is almost always above 3 meters per second, and may exceed 5 meters per second.

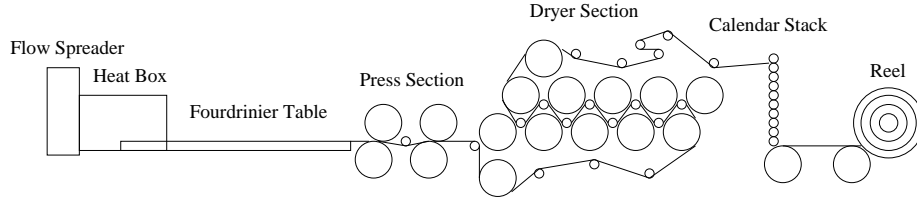


Figure 18: Fourdrinier paper machine

The HIL tester case study configuration can be kept in the same manner as the one depicted in Figure 14. It consists of a system model including 10 motors and a set of property observers as presented previously. The main task is to verify a set of controllers of angular velocity. Controllers are set the same as controller 1 in case study 2. The motors can be divided into 3 classes depending on their mechanical and electrical parameters, namely on J , the equivalent moment of inertia of the motor, and a load referred to the motor shaft.

It is necessary to verify the same properties as in the second case study. In addition we need to verify:

- The motor angular velocity should not exceed an angular velocity of any other motor by more than $\pm 0.1 \text{ Rad s}^{-1}$ (± 5.73 degrees).

Therefore:

Properties 1, 2 and 3 are the same as properties 1, 2, and 3 in the second case study.

Property 4 “*Angular velocity difference of the motor x and of the motor y does not exceed range $\pm 0.1 \text{ Rad s}^{-1}$ ”.* The test begins when the friction m_z is changed (60 ms).

The timed automata observers for properties 1, 2 and 3 were shown in Figure 15. A general timed automaton of property 4 is depicted in Figure 19. Based on the three motor classes, three property 4 observers have been designed.

The tester results are depicted in Figure 20 and 21. Figure 20 shows the step response for the three classes of the motors in the upper chart. The results of properties 1,2, and 3 in the chart below are grouped due to the same tester

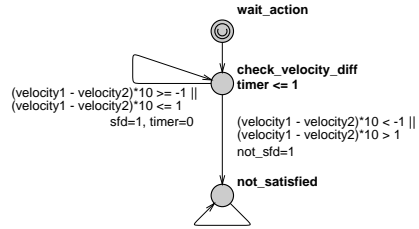


Figure 19: A general timed automaton of the property

results for all motor classes. Properties 1, 2, and 3 are satisfied for all tested controllers.

Figure 20 presents the results of property 4. The upper chart presents velocity difference $\Delta_{\omega_{x-y}}$ between motor class x and motor class y . The lower chart shows results of property 4 for motor class 1 and 2 (denoted Property 4_1-2), 2 and 3 (denoted Property 4_2-3) and finally 3 and 1 (denoted Property 4_3-1).

Property 4_1-2 for motor class 1 and 2 is satisfied. Angular velocity difference is less than 0.1 Rad s^{-1} .

Property 4_2-3 for motor class 2 and 3 is satisfied. Angular velocity difference is less than 0.1 Rad s^{-1} .

Property 4_3-1 for motor class 3 and 1 is not satisfied. Angular velocity difference exceeds 0.1 Rad s^{-1} at 63 msec .

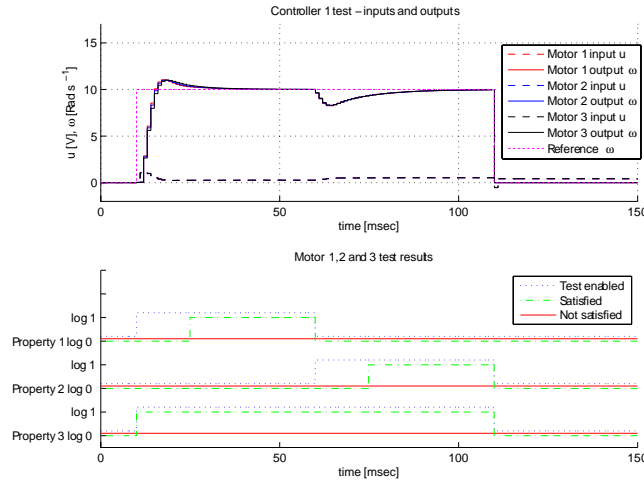


Figure 20: Controller 1 test, property 1, 2 and 3

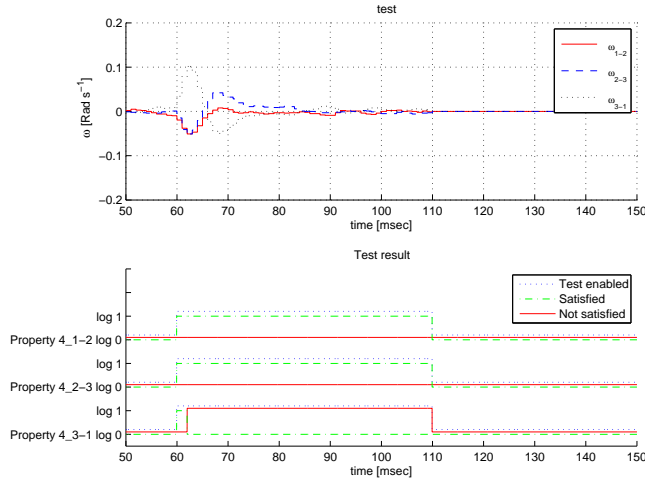


Figure 21: Motor class Δ test, property 4_1-2, 4_2-3 and 4_3-1

3.3.1 Implementation notes

This subsection summarizes the FPGA utilization for the proposed motor case studies while using the following design tools: Matlab 7.1.0.246 (R14) Service Pack 3, Xilinx System Generator v8.2 and Xilinx ISE 8.2.03i.

Utilization of the FPGA resources (Slices, Flip Flops, Look-Up-Tables) is shown in Table 1. Column CS2 corresponds to the complete HIL tester for the second case study. Column CS2⁺ extends the previous case by two additional motors. Column CS3 corresponds to the complete HIL tester for the third case study including 10 motors with controllers and observers of 4 properties.

	CS2	CS2 ⁺	CS3
Slices	4258 (10%)	11289 (26%)	36296 (86%)
Slice Flip Flops	444 (1%)	778 (1%)	1947 (2%)
4 input LUTs	5456 (6%)	13991 (16%)	43931 (52%)

Table 1: FPGA utilization summary (in percents of XC4VFX100 resources)

CS2 fits into an XC4VFX12 chip, on the other hand, CS3 requires an XC4VLX100 or XC4VFX100 chip family.

The resources of the FPGA for the continuous system model (as shown in Figure 7) are consumed in linear proportion to the system order including the FPGA interconnections relating each delay block (z^{-1}) with one previous block, one subsequent block, system output and the feedback. Also in the discrete part (as shown in Figure 3), the resources of the FPGA are consumed in linear proportion to the number of states and variables in the automaton.

Table 1 (summarizing sparsely interconnected systems) shows that, due to the modular structure, the FPGA utilization grows proportionally to the num-

ber of systems (both continuous and discrete event ones). The number of interconnections relates to the structure of the system, which is under designer's control. The width of the interconnection data buses can be set according to the application needs using bit-exact simulation in Matlab/XSG (considering influence of 8/16/24/32 bit arithmetic on the precision of the FPGA implementation) and verification in UPPAAL (gives the possibility to find out the minimum/maximum value of the particular integer variable).

4 Conclusion

The tester tool shown in this paper enables one to prove the quality of the developed controllers without them actually being assembled in the final product. Testing the considered class of applications requires a high sampling frequency, low jitter and scalability. Therefore, we have chosen an FPGA platform, synchronous logic hardware, capable of achieving a high degree of parallelism.

Due to the hybrid nature of industrial applications, the presented methodology combines discrete event systems and continuous systems using timed automata and transfer function representation.

The sampling period of the tester tool is given by the length of the FPGA clock cycle and by the number of the FPGA clock cycles needed to evaluate the continuous part and the discrete part. The shortest possible FPGA clock cycle is about 50 *nsec*, assuming a continuous system of the third order and a 24-bit fixed point arithmetic unit on Xilinx Virtex 4. The existence of the urgent locations and channels in the timed automaton requires repetitive calls of the corresponding timed automaton function, which gives the number of the FPGA clock cycles needed to evaluate the state of the timed automaton. In normal applications (free of timed automata with long sequences of urgent locations), one can easily approach the sampling period in hundreds of *nsec*. If, an even shorter sampling period is required, then it is necessary to change the quantization of the continuous values to a smaller set of discrete values, that can be handled by smaller arithmetic units leading to a shorter clock cycle.

Moreover, using FPGA, used as the synchronous logics, jitter is less than one clock cycle. The scalability of our tester tool is given by the size of the FPGA only. Due to the physical parallelism, the blocks do not influence each other.

On the other hand, when time and continuous variables require wide data buses, the number of FPGA interconnections increase and becomes a limiting factor (both for system model description and tested properties) of the presented method.

In the future, we will focus on the interfaces, namely improvement of ADC and DAC. Further work will also deal with a dynamic reconfiguration of the FPGA that can be considered for on-line execution of test cases and automation of the testing process. Since we are using the same formalism in the discrete-event system part, one may also consider to join the automatic test generation by UPPAAL-COVER and the test execution in our tool running at a very high

time resolution.

Currently, we are using a Linux OS running on the PowerPC hard core hosted on the same chip. This enables us to increase the flexibility and divide the problem into a HW (set of FPGA coprocessors) and SW part.

We are also considering using Allen's interval algebra [1] for the generation of the test properties. It proposes thirteen basic relations between the time intervals and the operations on them. The basic relations are *precedes*, *meets*, *overlaps*, *finished by*, *contains*, *starts*, *equals*, *started by*, *during*, *finished*, *overlapped by*, *met by* and *preceded by*. Operations on relations are *complement*, *composition*, *converse*, *intersection* and *union*. The test generation approach based on this algebra has been presented in [10], where it is used for SysML temporal observer design.

Acknowledgements

This work was supported by the Ministry of Education of the Czech Republic under research programme MSM6840770038 and by the Ministry of Industry and Trade of the Czech Republic under project FT-TA3/044. The authors would like to thank the anonymous referees for providing many invaluable comments and suggestions that led to significant improvement of this paper.

References

- [1] James F. Allen. Maintaining knowledge about temporal intervals. *Communication of the ACM*, 11:832–843, 1983.
- [2] Karine Altisen and Stavros Tripakis. Implementation of timed automata: An issue of semantics or modeling? In *FORMATS 2005*, pages 273–288, 2005.
- [3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [4] Gerd Behrmann. *The UPPAAL Timed Automata Parser Library - LIBU-TAP*, 2007.
- [5] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal, formal methods for the design of real-time systems. In *SFM-RT 2004*, pages 200–236. Springer-Verlag, 2004.
- [6] Blumenbecker Prag s.r.o. *Paper Manufacturing Reference – Cellulose Drier*, 2007. <http://www.blumenbecker.cz>.
- [7] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an audio-control protocol. In *FTRTFT'94, Lecture Notes in Computer Science*, volume 863, 1994.

- [8] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. If: a validation environment for timed asynchronous systems. In E.A. Emerson and A.P. Sistla, editors, *CAV2000 proc.*, volume 1855 of LNCS, pages 543–547. Springer Verlag, 2000.
- [9] E. Allen Emerson. Temporal and modal logic. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072, 1990.
- [10] Benjamin Fontan, L. Apvrille, P. de Saqui-Sannes, and J. P. Courtiat. Real-time and embedded system verification based on formal requirements. In *IEEE symposium on Industrial Embedded Systems - IES2006*, 2006.
- [11] FsmLabs ltd. *RTLlinux*, 2005.
- [12] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. *Lecture Notes in Computer Science, Testing Real-time systems using UPPAAL*, pages 77–117. Springer-Verlag, 2007.
- [13] IPG Automotive GmbH. *CarMaker*, 2006.
- [14] Moez Krichen and Stavros Tripakis. Black-box conformance testing - for real-time systems for real-time systems. Technical report, VERIMAG, www-verimag.imag.fr, 2004.
- [15] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *EMSOFT '05*, pages 299–306, Jersey City, NJ, USA, September 18 - 22 2005. ACM Press New York, NY.
- [16] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. *Springer International Journal of Software Tools for Technology Transfer*, 3:353–368, 2001.
- [17] M. D. Lutovac, D. V. Tomic, and B. L. Evans. *Filter Design for Signal Processing Using MATLAB and Mathematica*. Prentice Hall, 2000.
- [18] Petr Matoušek, Aleš Smrčka, and Tomáš Vojnar. High-level modelling, analysis, and verification on FPGA-based hardware design. In *IFIP CHARME 2005*, pages 371 – 375. LNCS, LNCS 3725, 2005.
- [19] Maxon ltd. *A-Max 26, Graphite Brushes, 11 Watt Servomotor Data-Sheet*, April 2005.
- [20] M. Mikucionis, K. Larsen, and B. Nielsen. Online on-the-fly testing of real-time systems. Technical Report RS-03-49, (BRICS), 2003.
- [21] Andrew J. Miller. *Linear Feedback Shift Register in a Programmable Gate Array – United States Patent 6181164*. Xilinx, Inc., 2001.

- [22] National Instruments corp. *LabVIEW FPGA in Hardware-In-the-Loop Simulation Application*, 2003.
- [23] Katsuhiko Ogata. *Discrete-time control systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.
- [24] Katsuhiko Ogata. *Modern Control Engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2001.
- [25] Ismael Ripoll, A. Crespo, G. Lipari, and A. Matellanes. Ocera: An architecture based on components for the development of real-time embedded application. In *International Workshop on Advanced Real-Time Operating System Services*, pages 17–30, 2003.
- [26] H. Schludermann, T. Kirchmair, and M. Vorderwinkler. Soft-Commissioning: Hardware-in-the-loop based verification of controller software. In PROFACTOR GmbH, editor, *Proceeding of the 2000 Winter Simulation Conference*, 2000.
- [27] Xilinx, Inc. *Linear Feedback Shift Register v3.0*, March 28 2003.
- [28] Xilinx, Inc. *Xilinx System Generator v7.1 User Guide*, 2005.
- [29] Sudhakar Yalamanchili. *VHDL Starter's Guide*. Prentice Hall, 1998. 269 pages.