School of Computer and Communication Sciences

Design Patterns of Timed Automotive Systems

Jagannath Aghav, Claude Petitpierre

October 2003
Technical Report IC/2003/62
http://ltiwww.epfl.ch/

EPFL- I & C CH-1015 Lausanne

Design Patterns of Timed Automotive Systems*

Jagannath Aghav, Claude Petitpierre
Computer Networking Laboratory
School of Computer and Communication Sciences
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne Switzerland
{jagannath.aghav, claude.petitpierre}@epfl.ch

Abstract

The functional complexity of hardware and software systems is growing exponentially. This demands an integration of system validation in the design phase to guarantee that a concrete implementation conforms to the modeled requirements. System validation process involves use of advanced systematic techniques and tools. In this report we present design patterns of timed automotive systems and a method for validating the real-time behavioral patterns of control intensive applications. The real-time behavioral patterns of embedded controllers are modeled using Statechart diagrams of Unified Modeling Language. Concrete implementation of the Statechart diagrams consists of active objects in Java programming language with synchronous communication. In the implementation using specified timed annotations, we validate the real-time constraints to be satisfied by embedded controllers. We shall illustrate the validation process with an example of a gear controller used in automobiles, and furthermore, provide an algorithmic solution.

Keywords

Unified Modeling Language, Statechart diagrams, Design Patterns, Embedded Systems, Reactive Systems, Validation and Verification.

*Full version of the paper [2] appeared in SVERTS'03.

Contents

1 Introduction

Growth in the technology of manufacturing microprocessors and decline in the prices have made a revolution of supplanting all electronic systems to single chip computers. The application-specific single chip computers or embedded systems [5] are often utilized in control intensive applications. Such systems or embedded controllers behaving with multiple functionality are fanned out in everyday life. The high cost of failures makes validation of system essential. These hardware-software co-designed systems are substantially increasing in complexity and size. Thus forming system validation a formidable challenge. The rapid development in this area makes it necessary to have systematic methods and tools for designing and validation.

Unified Modeling Language (UML) [8, 22] is used by software developers in the early stages of software development for expressing object oriented models and designs. The generalized solutions are mined in form of architectural patterns [11] and behavioral patterns [10] or broadly as design patterns [12]. Various real-time models [3] and their finite state formalisms [4, 15] are employed for validation of real time systems specified in different programming languages. We consider the design and validation of controllers for timed systems [1, 21] in UML. The integration of validation methods in development phase of UML will assist in devising the design patterns of complex systems with high reliability. In this report we present design

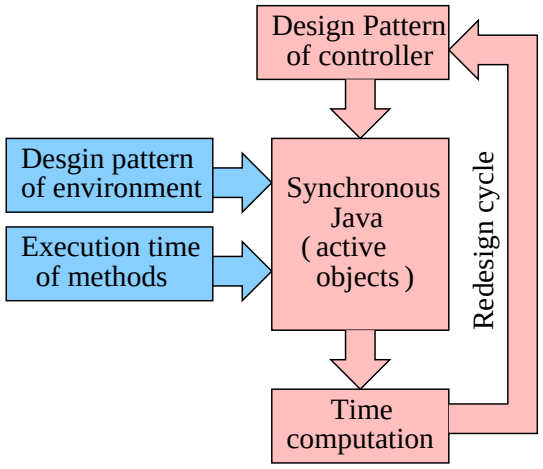

Figure 1: Validation Process Cycle for Designing Controllers.

patterns of timed automotive systems and describe a method for validating

real-time behavioral patterns in the design of embedded controllers. The reactive nature of controllers allow the classification of the behavioral patterns into two categories: behavioral patterns of program controllers and of components in environment. The specification of environment as model for validation originates from Sifakis [7] where non-computing regions of program are specified using event relations such as *cumulative, separate* and *coalescent*. We continue to apply the environment modeling for identifying the length of execution sequences in the UML framework. Figure 1 shows the general view of validation process of our approach which is a cyclic process like different play-in-scenarios in [14]. The steps in validation process are as follows:

(i) Model the behavioral pattern of the program controller in Statechart diagrams of UML including time constraints.

(ii) Model the behavioral pattern of corresponding component being controlled in the environment.

(iii) Implement the code from Statecharts diagrams as active objects having synchronous communication.

(iv) Read execution times in Java code. The time values are based on selected target architecture. Either the compiler supports the reading of time values or the designer supplies. Labels are annotated based on notations of *TimeC* in [17] or in [7, 23] for synchronous programming language ESTEREL [6].

(v) Construct a finite state automaton with time annotated labels on transitions from the concrete realization.

(vi) Compute the longest response time on each stimuli of the environment.

(vii) Check the real-time constraints in all possible execution paths for satisfaction. The longest time response of all paths is displayed on the transition of Statechart diagram of the controller.

The main contributions of the report are, a method for validating real-time behavioral patterns of embedded controllers and an algorithmic solution to the validation process. We provide an illustration of our method with an industrial case study of a gear controller as an example of timed automotive systems. The rest of the report is structured as follows: In the sequel we present an example of a gear controller used in automobiles. In Section 3

we describe the modeling of architectural, behavioral patterns of the example and demonstrate timed annotations in code implementation. Section 4 presents the validation of real-time constraints and an algorithmic solution for computing time responses. In Section 5 we present the conclusions on modeling and on the validation approach.

2 Gear Controller

We take an example of a gear controller (for more details see [18]) proposed by Mecel. The gear controller has five main components: Clutch, Gear Box, Engine, Interface and Controller apart from the linking and plug in components. The main controller is composed in parts or modules. These modules together with corresponding components forms the complete gear controller as shown in Figure 2. We categorize the composition of system into two

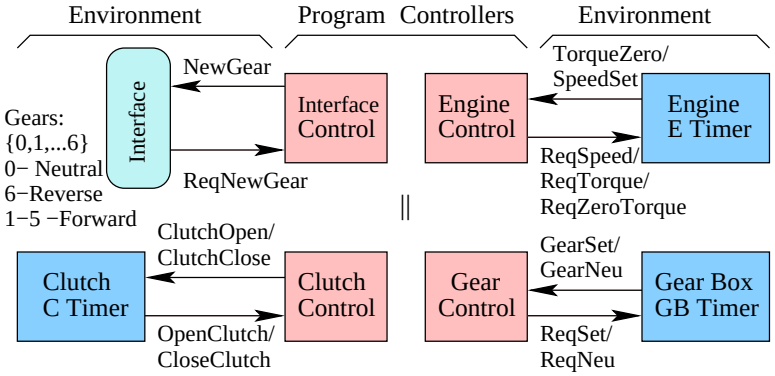

Figure 2: Composition of Mecel's Gear Controller.

parts: controller part as *program* and component part as *environment*.

The automated assembly has five forward gears (1-5), one reverse gear (6) and one neutral gear (0). The central control unit is the gear controller. The gear box changes gear as per the input gear request by gear controller. The interface component is the medium between the user and the gear controller. The user with gear stick or with control buttons requests new gear. The interface component keeps the information of current gear and the current status such as *idling* or *changing* of gear controller and sends new gear request to gear controller.

The engine operates in three control modes: (a) Normal torque mode (b) Zero torque mode and (c) Synchronous speed mode. In normal torque

mode, engine supplies requested engine torque over the transmission. In zero torque mode, engine finds zero torque difference over the transmission. In the last synchronous speed mode, the engine finds zero speed difference between engine and wheels. In these modes of operation if it is difficult to achieve zero torque transmission or synchronous speed in a given time bound then the clutch is operated to change of gear operation.

Next we shall describe sequence of operations with respect to the components that undergo for change of gear.

2.1 Gear Changing Algorithm

The gear controller executes the following steps to achieve the gear-change request:

1. (Wait until request for new gear.) The controller waits until new gear is requested. If new gear requested is different than the current gear of gear box then go to step 2.

2. (Obtain zero torque over transmission.) The gear controller sends `ReqZeroTorque` request of zero torque to engine. The engine obtains zero torque transmission and sends back the signal `TorqueZero` to the gear controller. This state of zero torque transmission is essential for releasing the earlier set gear. If the engine fails to find zero torque transmission then the clutch is operated.

3. (Bring gear box in neutral gear.) The gear controller requests the gear box for neutral gear by sending the signal `ReqNeu`. The gear box changes the current gear to neutral gear and sends back the signal `GearNeu` to the gear controller.

4. (Set the required speed of engine.) The gear controller obtains the synchronous speed by sending signal `ReqSpeed` and by receiving signal `SpeedSet` from the engine. If synchronous speed is not achieved in the given time bound then the clutch is requested.

5. (Set new gear.) The gear controller sets to new gear. It requests new gear by signal `ReqSet` to the gear box and gear box sets new gear and responds back with signal `GearSet`.

6. (Set the required torque and go to step 1.) The gear controller requests the engine with signal `ReqTorque`to increase the torque. In this case, the engine gives the requested torque and no signal is sent back to the

gear controller. This increase in torque is necessary so that the same wheel torque level is achieved before the gear change is effected. The gear controller operation goes to step 1.

In addition to accomplish the change of gear the timing requirements that are described in sequel on each component should also be satisfied.[†]

2.2 Timing Requirements

Following are the time requirements on each component for the correct functioning of gear controller and also on gear controller as a total integrated unit.

(a) Gear box sets a gear in 100 to 300 ms. If setting time is equal to or more than 300 ms then returns to error state.

(b) Gear box releases gear in 100 to 200 ms. If releasing time is more than 200 ms then it goes to error state.

(c) Clutch changes state from open to close or vice versa in 100 to 150 ms otherwise returns to error state.

(d) The maximum time bound to obtain a zero torque for engine is 400 ms.

(e) For engine the maximum time bound to obtain a synchronous speed is 500 ms otherwise engine enters into error state.

(f) A gear change should be completed within 1.5 seconds.

(g) A gear change under normal conditions should be happen within 1 sec.

Next we shall present the modeling and concrete realization of the example.

3 Implementation

UML provides stereotype of classes, protocols and Statechart diagrams to model event driven systems. Using capsule as a stereotype class, we model each independent control element of gear controller forming the architectural class diagram. The dynamic behavior of each element is modeled using a Statechart diagram. In the following we present architectural patterns, behavioral patterns and timed annotations of gear controller example.

[†]For validation purpose measurement is in time units instead of milliseconds. Based on target architecture exact times are computed.

3.1 Architectural Patterns

This is the static structure of the full gear controller that has a capsule for each control part and protocol stereotype for communication. For simple clutch controller, the capsules `clutchcontroller` and `Clutch` represent the clutch controlling part and the environment part of clutch component. Communication between these two elements is through the protocol `ControlClutch`. Figure 3 shows the architectural class diagram of the clutch controller. The architecture of gear controller is composed with

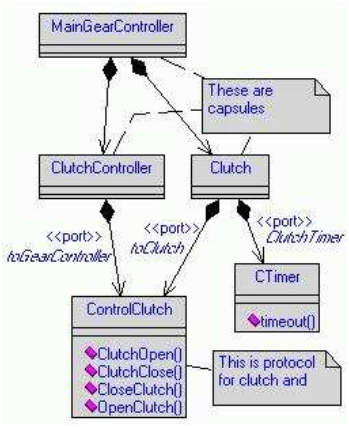

Figure 3: Class Diagram of Clutch Controller.

main capsule `MainGearController` which comprises capsules of all other components and controllers. Figure 4 shows architectural class diagram of gear controller.

3.2 Behavioral Patterns

The dynamic behavior of program controller part and its corresponding component is specified using Statechart diagrams [13]. We use UML variant of Statechart diagrams. The component or environment behavior is specified to assist the validation. Figure 5 shows the behavioral pattern of clutch controller or program part. Figure 6 shows behavioral pattern of clutch component or environment. The behavioral patterns of other components and program controllers are shown in Appendix A.1. The interactions between different program controllers are modeled using collaboration diagrams.

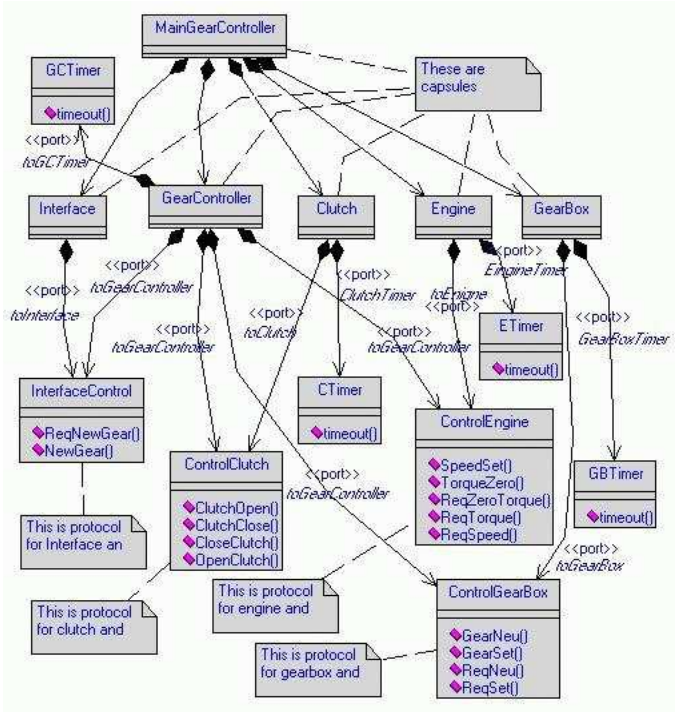

Figure 4: Class Diagram of Gear Controller.

3.3 Annotating Synchronous Active Objects

From the behavioral patterns the concrete implementation is generated as follows: For each controller and its corresponding component active objects are created. These active objects communicate synchronously and such objects are known as synchronous active objects [19, 20]. The communication between the controllers and its components is synchronized by having a common method name in Java. We annotate the syntax of synchronization calls with following label structure within the comments beginning with special symbol '!'.

$$//!\ \{\ \textit{Calling active object number, receiving active object number, method name, time units}\ \}$$

The structure of label is composed with the following information:

(i) *Caller* active object number.

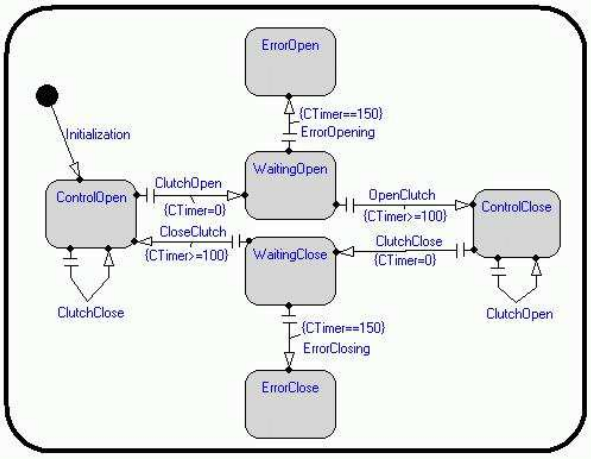

Figure 5: Statechart Diagram of Clutch Controller.

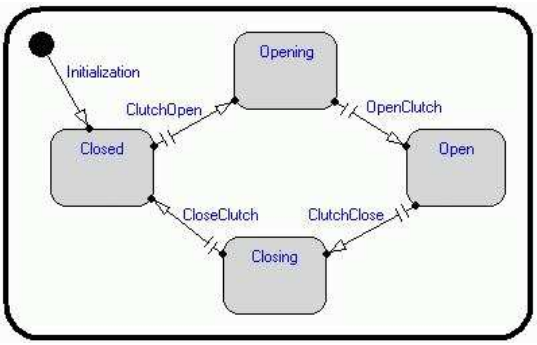

Figure 6: Statechart Diagram of Clutch Component

(ii) *Receiver* active object number.

(iii) The *name* of the method that is executed and

(iv) The *execution time* of the method.

The two categories of composition, the program controller and components of *environment* are specified in annotations as even and odd numbers respectively. For the example of simple clutch controller two active objects are created: `ClutchController` and the component `Clutch` with synchronous communication. The annotated source with the timing information appears

```
...
public class Gear {
...
active class ClutchController{
...
accept OpenClutch;
//! {1, 0, OpenClutch, 35}
...
accept CloseClutch;
//! {1, 0, CloseClutch, 10}
...
}
```

```
...
active class Clutch{
...
accept ClutchOpen;
//! {0, 1, ClutchOpen, 30}
...
accept ClutchClose;
//! {0, 1, ClutchClose, 25}
...
}
...
```

Table 1: Annotations in the Source Code of Clutch Controller.

as shown in Table 1. As the code depicts the clutch controller is numbered with 0 and clutch component with 1.

4 Validation

In this section we present the validation of the finite state model generated from the java source implementation and from the timed annotations on synchronous active objects. The generated finite state model is a directed graph that represents sequential execution of active objects. The time annotated labels on synchronization becomes the labels of the edges in the finite state model. For validation we detect the transitions that have communication from *environment* component to *Program* controller. Following algorithm computes time responses for the input finite state model.

4.1 Time Computation Algorithm

Let us denote *program* controller with P and *environment* with E. $(P \uparrow E)$ denotes a transition or edge that has communication from *program* controller to *environment* component. The input to the algorithm is a file [9, 16] containing the description of finite state model. Output is the sum of execution times for all possible paths that has starting transition $(E \uparrow P)$ and a terminating transition of either $(P \uparrow E)$ or $(P \uparrow P)$. Here $(P \uparrow P)$ denotes transition type within a program controller that has no outgoing edges. The steps for computing time are as follows: Read the description of labels, edges, and vertexes from the input file. Search the new edge that has

communication from environment to program controller, $(E \uparrow P)$. For the selected new edge, find the corresponding sinking vertex. From the sinking vertex find all possible paths ending on next new edge of type $(E \uparrow P)$. All the paths are terminating with either $(P \uparrow E)$ or $(P \uparrow P)$ type of edge. Compute the time on all the paths by summing up the execution times specified on the labels. Display the transition that takes longest time response into Statechart diagram of controller.

4.2 Clutch Controller

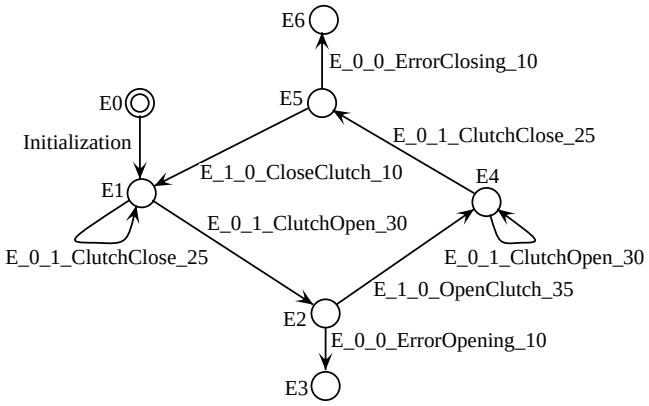

Figure 7: Finite State Model of Clutch Controller

Figure 7 shows the finite state model representing execution of two synchronous active objects with timed annotated labels for the simple clutch controller example. Composition with other controllers is not taken in account for this simple case. The behavioral patterns of this model are shown in Figure 5 and Figure 6. The result of time computation for the model is as following:

$E_4 \rightarrow E_6$: `Time(E_1_0_OpenClutch_35 + E_0_1_ClutchOpen_30 + E_0_1_ClutchClose_25 + E_0_0_ErrorClosing_10 ) = 100 units`

The longest time taken by the controller for $(E \uparrow P)$ transition having `OpenClutch` method of Java is 100 units. The result is displayed in the Statechart diagram of controller. The time computation for all transitions is listed in Appendix A.2.

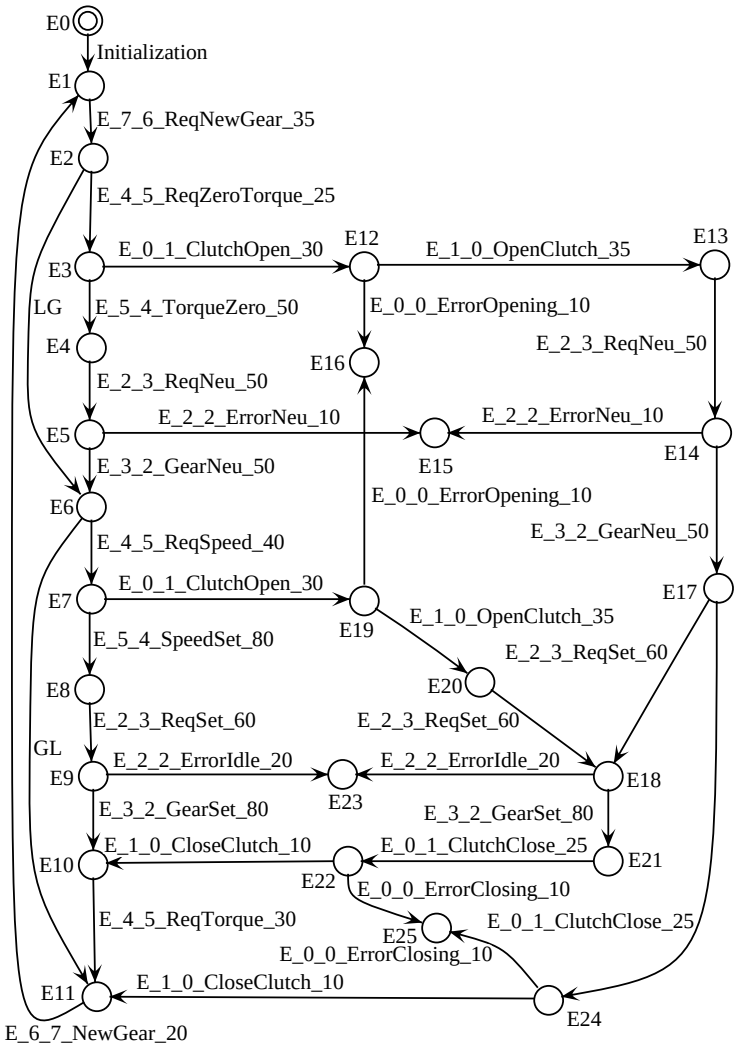

Figure 8: Finite State Model of Gear Controller.

4.3 Gear Controller

Now consider the finite state model of the composition for all four controllers.The environment statechart diagram of the respective components are provided for the validation process. The active objects for controllers of clutch, Gearbox, Engine and Interface are even numbered starting with

Sr.No.	Program	num	Environment	num
1.	Clutch Controller	0	Clutch component	1
2.	Gearbox Controller	2	Gearbox component	3
3.	Engine Controller	4	Engine component	5
4.	Interface Controller	6	Interface component	7

Table 2: Label numbering for program controllers and corresponding environment components.

0, whereas the corresponding environment components are odd numbered starting from 1. The synchronous active objects used in the composition are time annotated with numbers as shown in Table 2. Figure 8 shows the finite state model representing the execution of all synchronous active objects with time annotated labels for gear controller example. The result of time computation for the model is as following:

$E_8 \rightarrow E_{23}$: `Time(E_5_4_SpeedSet_80 + E_2_3_ReqSet_60 + E_2_2_ErrorIdle_20)=160 units`

The longest time taken by program controller for path is 160 units. There are two paths $E_{10} \rightarrow E_1$ and $E_{10}{}^* \rightarrow E_1$ showing the same sinking vertex E_{10} with two incoming $(E \uparrow P)$ type of transitions. The sum of execution time for the two paths is 130 units and 60 units respectively. The full result of time computation is given in Appendix A.3.

5 Conclusions

We have presented design patterns of automotive timed systems and a method for validating real-time behavioral patterns of embedded controllers specified in statecharts diagrams of UML. The validation process is illustrated with an example of a gear controller. The method aims for the integration into the design phase of UML to assist in predicting the run time of behavioral patterns for a given target architecture and also in simplifying the constructions of timed systems. The method validates only real-time behavioral patterns and not the architectural patterns or internal inconsistencies.

References

[1] E. Asarin, O. Maler, A. Pnueli and J. Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control.* Elsevier, 1998.

[2] Jagannath Aghav and Claude Petitpierre. Validating real-time behavioral patterns of embedded controllers. In *International Workshop on Specification and Validation of UML models for Real Time and Embedded Systems, SVERTS, at UML 2003*, San Francisco, CA, USA, October 2003.

[3] R. Alur and T.A. Henzinger. Logics and models of real time: a survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, LNCS 600, pages 74–106. Springer-Verlag, 1992.

[4] Rajeev Alur and David L Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach.* Kluwer Academic Publishers, April 1997.

[6] Gérard Berry and G Gonthier. The Esterel sychronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] Valérie Bertin, Michel Poize, and Joseph Sifakis. Towards validated real-time software. In *Proceedings of the 12 th Euromicro Conference on Real Time Systems*, pages 157–164, Stockholm, June 2000.

[8] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, Massachusetts, USA, 1 edition, 1999.

[9] Amar Bouali, Annie Ressouche, Valerie Roy, and Robert De Simone. The FCTOOLS user manual (version 1.0). Technical Report RT-0191, INRIA, Institut National de Recherche en Informatique et en Automatique, 1996.

[10] Bruce Powel Douglass. *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns.* Object Technology Series. Addison-Wesley, 1999.

[11] Bruce Powel Douglass. *Real-Time Design Patterns, Robust Scalable Architecture for Real-Time Systems.* Addison-Wesley, 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[13] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[14] David Harel. From play-in scenarios to code: An achievable dream. *Computer*, 34(1):53–60, January 2001.

[15] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[16] Eleftherios Koutsofios. Editing graphs with *dotty.* Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1994. This report, and the program, is included in the `graphviz` package, available for non-commercial use at http://www.research.att.com/sw/tools/graphviz.

[17] Allen Leung, Krishna V Palem, and Amir Pnueli. TimeC: A time constraint language for ILP processor compilation. In K A Hawick and H A James, editors, *The* 5^{th} *Australian Conference on Parallel and Real Time Systems, Australia*, pages 57–71. Springer Verlag, 1998.

[18] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. In *4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 281–297. Springer Verlag, March-April 1998.

[19] Synchronous Java. available on web site http://ltiwww.epfl.ch/sJava/.

[20] Claude Petitpierre. Synchronous C++, a Language for Interactive Applications. *IEEE Computer*, pages 65–72, September 1998.

[21] P. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–89, January 1989.

[22] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, USA, 1 edition, 1999.

[23] R K Shyamasundar and J V Aghav. Validating real-time constraints in embedded systems. In *Proceedings of IEEE Pacific Rim International Symposium on Dependable Computing, PRDC*, pages 347–355, Seoul, Korea, December 2001.

A Appendix

A.1 Statechart diagrams of Controllers and Components.

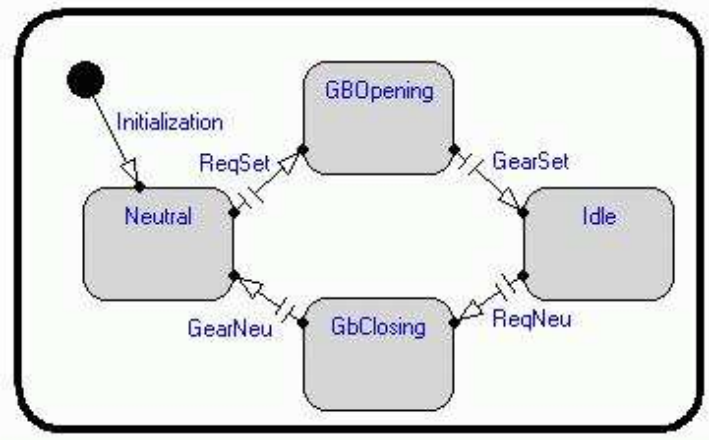

Figure 9: Statechart Diagram Gearbox Component.

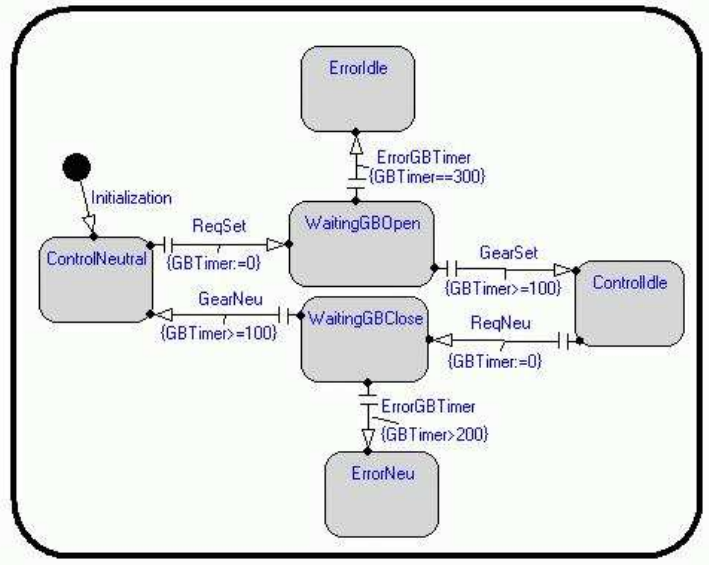

Figure 10: Statechart Diagram Gearbox Controller.

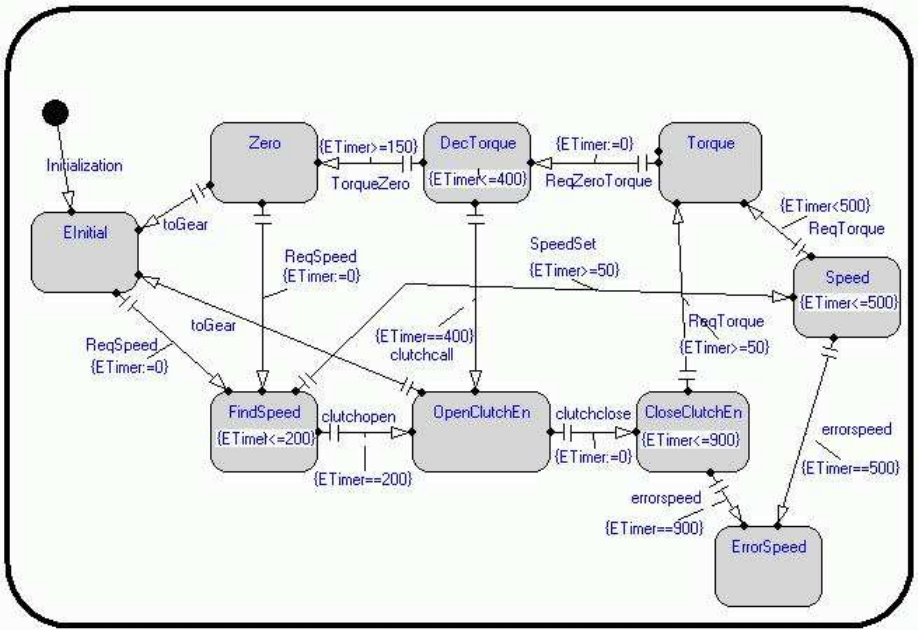

Figure 11: Statechart Diagram Engine Controller.

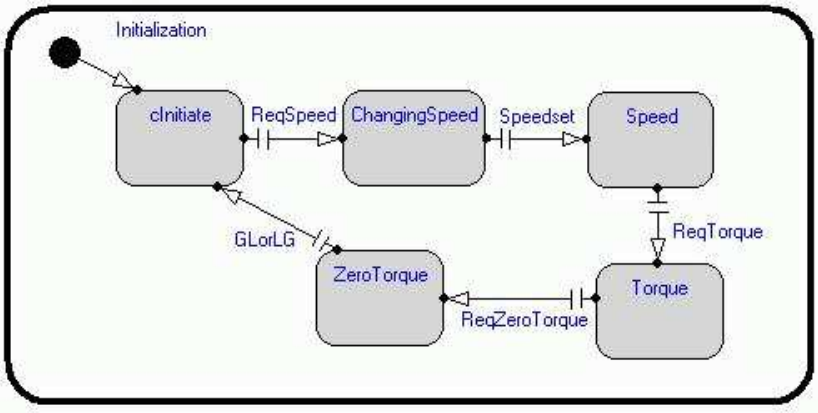

Figure 12: Statechart Diagram Engine Component.

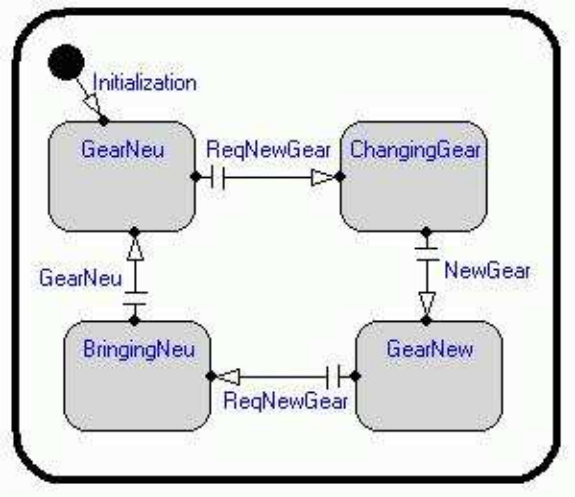

Figure 13: Statechart Diagram Interface Controller.

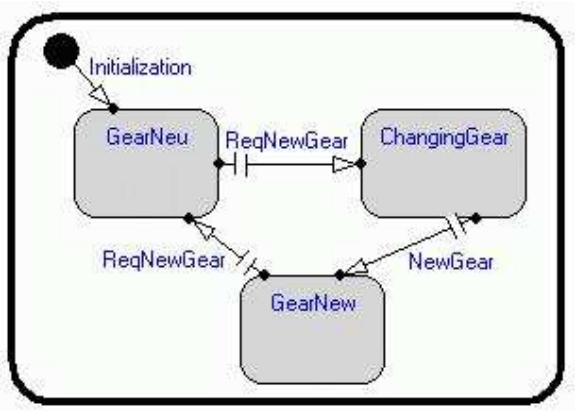

Figure 14: Statechart Diagram Interface Component.

A.2 Validation of Clutch Controller

$E_1 \rightarrow E_2$:
Time(E_1_0_CloseClutch_10 + E_0_1_ClutchOpen_30) = 40 units
Time(E_1_0_CloseClutch_10 + E_0_1_ClutchClose_25
+ E_0_1_ClutchOpen_30) = 65 units
$E_1 \rightarrow E_3$:
Time(E_1_0_CloseClutch_10 + E_0_1_ClutchClose_25

```
    + E_0_1_ClutchOpen_30 + E_0_0_ErrorOpening_10) = 75 units
E4→E5 :
Time(E_1_0_OpenClutch_35 + E_0_1_ClutchOpen_30
                         + E_0_1_ClutchClose_25 ) = 90 units
Time(E_1_0_OpenClutch_35 + E_0_1_ClutchClose_25) = 60 units
E4→E6 :
Time(E_1_0_OpenClutch_35 + E_0_1_ClutchOpen_30
  + E_0_1_ClutchClose_25 + E_0_0_ErrorClosing_10 ) = 100 units
```

A.3 Validation of Gear Controller

```
E2→E3 :
Time(E_7_6_ReqNewGear_35 + E_4_5_ReqZeroTorque_25) = 60 units
E2→E12 :
Time(E_7_6_ReqNewGear_35 + E_4_5_ReqZeroTorque_25
                             + E_0_1_ClutchOpen_30) = 90 units
E2→E16 :
Time(E_7_6_ReqNewGear_35 + E_4_5_ReqZeroTorque_25
   + E_0_1_ClutchOpen_30 + E_0_0_ErrorOpening_10) = 100 units
E2→E1 :
Time(E_7_6_ReqNewGear_35 + E_6_7_NewGear_20) = 55 units
E2→E19 :
Time(E_7_6_ReqNewGear_35 + E_4_5_ReqSpeed_40
                       + E_0_1_ClutchOpen_30) = 105 units
E2→E16 :
Time(E_7_6_ReqNewGear_35 + E_4_5_ReqSpeed_40
 + E_0_1_ClutchOpen_30 + E_0_0_ErrorOpening_10) = 115 units


E4→E5 :
Time(E_5_4_TorqueZero_50 + E_2_3_ReqNeu_50) = 100 units
E4→E15 :
Time(E_5_4_TorqueZero_50 + E_2_3_ReqNeu_50
                         + E_2_2_ErrorNeu_10 ) = 110 units


E6→E7 :
Time(E_3_2_GearNeu_50 + E_4_5_ReqSpeed_40) = 90 units
E6→E19 :
Time(E_3_2_GearNeu_50 + E_4_5_ReqSpeed_40
                      + E_0_1_ClutchOpen_30) = 120 units
E6→E16 :
Time(E_3_2_GearNeu_50 + E_4_5_ReqSpeed_40
     + E_0_1_ClutchOpen_30 + E_0_0_ErrorOpening_10 ) = 130 units
E6→E1 :
Time(E_3_2_GearNeu_50 + E_6_7_NewGear_20 ) = 70 units
```

$E_8 \rightarrow E_9$:
Time(E_5_4_SpeedSet_80 + E_2_3_ReqSet_60) = 140 units
$E_8 \rightarrow E_{23}$:
Time(E_5_4_SpeedSet_80 + E_2_3_ReqSet_60
+ E_2_2_ErrorIdle_20) = 160 units

$E_{10} \rightarrow E_1$:
Time(E_3_2_GearSet_80 + E_4_5_ReqTorque_30
+ E_6_7_NewGear_20) = 130 units
$E_{10}* \rightarrow E_1$:
Time(E_1_0_CloseClutch_10 + E_4_5_ReqTorque_30
+ E_6_7_NewGear_20) = 60 units

$E_{13} \rightarrow E_{14}$:
Time(E_1_0_OpenClutch_35 + E_2_3_ReqNeu_50) = 85 units
$E_{13} \rightarrow E_{15}$:
Time(E_1_0_OpenClutch_35 + E_2_3_ReqNeu_50
+ E_2_2_ErrorNeu_10) = 95 units

$E_{17} \rightarrow E_{18}$:
Time(E_3_2_GearNeu_50 + E_2_3_ReqSet_60) = 110 units
$E_{17} \rightarrow E_{23}$:
Time(E_3_2_GearNeu_50 + E_2_3_ReqSet_60
+ E_2_2_ErrorIdle_20) = 130 units
$E_{17} \rightarrow E_{24}$:
Time(E_3_2_GearNeu_50 + E_0_1_ClutchClose_25) = 75 units
$E_{17} \rightarrow E_{25}$:
Time(E_3_2_GearNeu_50 + E_0_1_ClutchClose_25
+ E_0_0_ErrorClosing_10) = 85 units

$E_{20} \rightarrow E_{18}$:
Time(E_1_0_OpenClutch_35 + E_2_3_ReqSet_60) = 95 units
$E_{20} \rightarrow E_{18}$:
Time(E_1_0_OpenClutch_35 + E_2_3_ReqSet_60
+ E_2_2_ErrorIdle_20) = 115 units

$E_{21} \rightarrow E_{22}$:
Time(E_3_2_GearSet_80 + E_0_1_ClutchClose_25) = 105 units
$E_{21} \rightarrow E_{25}$:
Time(E_3_2_GearSet_80 + E_0_1_ClutchClose_25
+ E_0_0_ErrorClosing_10) = 115 units

Index