Formal Design and Analysis of a Gear Controller: an Industrial Case Study using Uppaal*

Magnus Lindahl[1] *Paul Pettersson*[2] *Wang Yi*[2]

[1] Mecel AB, Göteborg, Sweden. E-mail: `magnus.lindahl@mecel.se`
[2] Department of Computer Systems, Uppsala University, Sweden.
E-mail: `{paupet,yi}@docs.uu.se`

Abstract. In this paper, we report on an application of the validation and verification tool kit Uppaal in the design and analysis of a prototype gear controller, carried out in a joint project between industry and academia. The gear controller is a component in the control system operating in a modern vehicle, implementing the gear change algorithm. We give a detailed description of the formal model of the gear controller and its surrounding environment, and its correctness formalized in 46 logical formulas according to the informal requirements delivered by our industrial partner of the project. The second contribution of this paper is a solution to the problem we met in this case study, namely how to use a tool like Uppaal, which only provides reachability analysis to verify bounded response time properties e.g. *if* f_1 *(a request) becomes true at a certain time point, then* f_2 *(a response) must be guaranteed to hold within a given time bound.* We present a logic and a method to characterize and model–check such properties for networks of timed automata by syntactical transformation and reachability analysis. The advantage of this approach is that we need no additional implementation work to extend the existing model–checker, but simple manual syntactical manipulation on the system description. The method has been demonstrated in verifying the correctness of the gear controller design. It takes 2.99 seconds to check the 46 logical formulas by Uppaal installed on a Pentium 75MHz PC equipped with 24 MB of primary memory.

1 Introduction

Over the past few years, a number of modeling and verification tools for real-time systems [HHWT95, DOTY95, BLL+96] have been developed based on the theory of timed automata [AD94]. They have been successfully applied in various case-studies [BGK+96, JLS96, SMF97]. However, the tools have been mainly used in the academic community, namely by the tool developers. It has been a challenge to apply these tools to real-sized industrial case-studies. In this paper we report on an application of the verification tool-kit Uppaal to a prototype gear controller developed in a joint project between industry and academia. The project has been carried out in collaboration between Mecel AB and Uppsala University.

The gear controller is a component in the real-time embedded system that operates in a modern vehicle. The gear-requests from the driver are delivered over a communication network to the gear controller. The controller implements the actual gear change by actuating the lower level components of the system, such as the clutch, the engine and the gear-box. Obviously, the behavior of the gear controller is critical to the safety of the vehicle. Simulation and testing have been the traditional ways to ensure that the behavior of the controller satisfies certain safety requirements. However these methods are by no means complete in finding errors though they are useful and practical. As a complement, formal techniques have been a promising approach to ensuring the correctness of embedded systems. The project is to use formal modeling techniques in the early design stages to describe design sketches, and to use symbolic simulators and model checkers as debugging and verification tools to ensure that the predicted behavior of the designed controller at each design phase, satisfies certain requirements

* This work has been supported by ASTEC (Advanced Software TEChnology), NUTEK (Swedish Board for Technical Development) and TFR (Swedish Technical Research Council).

under given assumptions on the environment where the gear controller is supposed to operate. The requirements on the controller and assumptions on the environment have been described by Mecel AB in an informal document, and then formalized in the UPPAAL model and a simple linear–time logic based on the UPPAAL logic to deduce the design of the gear controller.

We shall give a detailed description of the formal model of the gear controller and its surrounding environment, and its correctness according to the informal requirements delivered by Mecel AB. Another contribution of this paper is a lesson we learnt in this case study, namely how to use a tool like UPPAAL, which only provides reachability analysis to verify bounded response time properties e.g. *if* f_1 *(a request) becomes true at a certain time point,* f_2 *(a response) must be guaranteed to be true within a time bound.* We present a logic and a method to characterize and model–check response time properties. The advantage of this approach is that we need no additional implementation work to extend the existing model–checker, but simple manual syntactical manipulation on the system description.

UPPAAL[3] is a tool suite for validation and symbolic model-checking of real-time systems. It consists of a number of tools including a graphical editor for system descriptions (based on Autograph), a graphical simulator, and a symbolic model–checker. In the design phase the symbolic simulator of UPPAAL is applied intensively to validate the dynamic behavior of each design sketch, in particular for fault detection, derivation of time constraints (e.g. the time bounds for which a gear change is guaranteed) and later also for debugging using diagnostic traces (i.e. counter examples) generated by the model–checker. The correctness of the gear controller design has been established by automatic proofs of 46 logical formulas derived from the informal requirements specified by Mecel AB. The verification was performed in a few seconds on a Pentium PC [4] running UPPAAL version 2.12.2.

The paper is organised as follows: Next section is a brief summary of various definitions for the semantics and models of real–time systems. In section 3, we present a simple logic to characterize safety and response time properties and a method to model–check such properties. In Section 4 and 5 the gear controller system and its requirements are informally and formally described. In Section 6 the formal description of the system and its requirements are transformed using the technique developed in section 3 for verification by reachability analysis. Section 7 concludes the paper. Finally, we enclose the formal description of the surrounding environment of the gear controller in the appendix.

2 Preliminaries

In this section, we briefly introduce all the necessary definitions for the basis of the UPPAAL modelling language. For details, we refer to [YPD94, LPY97].

2.1 Timed Transition Systems and Timed Traces

A timed transition system is a labeled transition system with two types of labels: atomic actions and delay actions (i.e. positive reals), representing discrete and continuous changes of real-time systems.

Let *Act* be a finite set of actions and $\mathcal{P}$ be a set of atomic propositions. We use $\mathbf{R}$ to stand for the set of non-negative real numbers, D for the set of delay actions $\{\epsilon(d) \mid d \in \mathbf{R}\}$, and Σ for the union $Act \cup D$ ranged over by $\alpha, \alpha_1, \alpha_2$ etc.

Definition 1. *A timed transition system over Act and* $\mathcal{P}$ *is a tuple* $\mathcal{S} = \langle S, s_0, \longrightarrow, V \rangle$*, where* S *is a set of states,* s_0 *is the initial state,* $\longrightarrow \subseteq S \times \Sigma \times S$ *is a transition relation, and* $V : S \to 2^{\mathcal{P}}$ *is a proposition assignment function.* □

A trace σ of a timed transition system is an *infinite* sequence of transitions in the form:

$$\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots s_n \xrightarrow{\alpha_n} s_{n+1} \ldots$$

[3] Installation and documentation is available at the UPPAAL home page http://www.docs.uu.se/docs/rtmv/uppaal/.

[4] 2.99 seconds on a Pentium 75MHz equipped with 24 MB of primary memory.

where $\alpha_i \in \Sigma$.

A position i of σ is a natural number. We use $\sigma[i]$ to stand for the ith state of σ, and $\sigma(i)$ for the ith transition of σ, i.e. $\sigma[i] = s_i$ and $\sigma(i) = s_i \xrightarrow{\alpha_i} s_{i+1}$.

We use $\delta(s \xrightarrow{\alpha} s')$ to denote the duration of the transition, defined by $\delta(s \xrightarrow{\alpha} s') = 0$ if $\alpha \in Act$ or d if $\alpha = \epsilon(d)$. Given positions i, k with $i \leq k$, we use $\Delta(\sigma, i, k)$ to stand for the accumulated delay of σ between the positions i, k, defined by $\Delta(\sigma, i, k) = \sum_{i \leq j < k} \delta(\sigma(j))$. We shall only consider *non–zeno* traces.

Definition 2. *A trace σ is non–zeno if for all natural number T there exists a position k such that $D(\sigma, 0, k) > T$. For a timed transition system $\mathcal{S}$, we denote by $Tr(\mathcal{S})$ all non–zeno traces of $\mathcal{S}$ starting from the initial state s_0 of $\mathcal{S}$.* □

2.2 Timed Automata with Data Variables

We study the class of timed transition systems that can be syntactically described by timed automata extended with data variables ranging over finite data domains.

Assume a finite set of clock variables $\mathcal{C}$ ranged over by x etc and a finite set of data variables $\mathcal{D}$ ranged over by i etc. We use $\mathcal{V}$ to denote the union of $\mathcal{C}$ and $\mathcal{D}$, ranged over by v. We use $\mathcal{G}(\mathcal{V})$ to stand for the set of formulas ranged over by g, generated by the following syntax: $g ::= \ c \mid g \wedge g$ where c is a constraint of the form: $x \sim n$ or $i \sim n$ for $x \in \mathcal{C}$, $i \in \mathcal{D}$, $\sim \in \{\leq, \geq, =\}$ and n being a natural number. We shall call elements of $\mathcal{G}(\mathcal{V})$ *guards*.

To manipulate clock and data variables, we use reset–operations of the form: $v := \exp$ where v is a clock or data variable and exp is an expression. A reset-operation on a clock variable should be in the form $x := 0$; A reset-operation on an integer variable should be in the form: $i := k * i + k'$ where k, k' are integer constants. We call a set of such reset–operations a *reset-set*. A reset–set is *proper* when the variables are assinged a value at most once. We use $\mathcal{R}$ to denote the set of all proper reset-sets, ranged over by r, r' etc.

Definition 3. *A timed automaton A over actions Act, atomic propositions $\mathcal{P}$, and $\mathcal{V}$, is a tuple $\langle L, l_0, E, I, V \rangle$, where L is a finite set of nodes (or locations), l_0 is the initial node, and $E \subseteq L \times \mathcal{G}(\mathcal{V}) \times Act \times \mathcal{R} \times L$ corresponds to the set of edges. In the case, $\langle l, g, a, r, l' \rangle \in E$ we shall write, $l \xrightarrow{g,a,r} l'$. $I : L \to \mathcal{G}(\mathcal{V})$ is a function which for each node assigns an invariant condition, and $V : L \to 2^{\mathcal{P}}$ is a proposition assignment function which for each node gives a subset of atomic propositions true in the node. We shall use $P(A)$ to stand for the union of the subsets of propositions true in all the nodes L of A, i.e. $P(A) = \bigcup_{l \in L} V(l)$.* □

Informally, a process modelled by an automaton starts at its initial location l_0 with all its variables initialized to 0. The values of the clocks increase synchronously with time at location l. At any time, the process can change location by following an edge $l \xrightarrow{g,a,r} l'$ provided the current values of the variables satisfy the enabling condition g. With this transition, the variables are updated by r.

A *variable assignment* is a mapping which maps clock variables $\mathcal{C}$ to the non-negative reals and data variables $\mathcal{D}$ to integers. For a variable assignment u and a delay d, $v \oplus d$ denotes the variable assignment such that $(v \oplus d)(x) = v(x) + d$ for any clock variable x and $(v \oplus d)(i) = v(i)$ for any integer variable i. This definition of $\oplus$ reflects that all clocks operate with the same speed and that data variables are time-insensitive. For a reset-operation r (a set of assignment-operations) we use $r(u)$ to denote the variable assignment u' with $u'(v) = V(\exp, u)$ whenever $v := \exp \in r$ and $u'(v') = u(v')$ otherwise, where $V(\exp, u)$ denotes the value of exp in u. Given a guard $g \in \mathcal{G}(\mathcal{V})$ and a variable assignment u, $g(u)$ is a boolean value describing whether g is satisfied by u or not.

2.3 Networks of Automata

To model concurrency and synchronization, we introduce a CCS-like parallel composition operator for automata. Assume automata $A_1 ... A_n$. We use $\overline{A}$ to denote their parallel composition $A_1 || \ldots || A_n$. The intuitive meaning of $\overline{A}$ is similar to the CCS parallel composition of $A_1 ... A_n$ with *all* actions being restricted, that is, $(A_1|...|A_n)\backslash Act$. Thus only internal synchronization between the components A_i is possible. We shall call $\overline{A}$ a network of automata [5]. We simply view $\overline{A}$ as a vector and use A_i to denote its ith component.

A *control vector* $\bar{l}$ of a network $\overline{A}$ is a vector of locations where l_i is a location of A_i. We shall write $\bar{l}[l_i'/l_i]$ to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l_i'.

A *state* of a network $\overline{A}$ is a configuration $\langle \bar{l}, u\rangle$ where $\bar{l}$ is a control vector of $\overline{A}$ and u is a variable assignment. The initial state of $\overline{A}$ is $\langle \bar{l}_0, u_0\rangle$ where $\bar{l}_0$ is the initial control vector whose elements are the initial locations of A_i's and u_0 is the initial variable assignment that maps all variables to 0.

The *semantics of a network* of automata $\overline{A}$ is defined in terms of a timed transition system $\overline{\mathcal{S}} = \langle S, s_0, \longrightarrow, V\rangle$ with the set S of states being the set of configurations, s_0 being the initial state i.e. $\langle \bar{l}_0, u_0\rangle$, the proposition assignment function V is defined by $V(\langle \bar{l}, u\rangle) = \bigcup_{l_i \in \bar{l}} V_i(l_i)$, and the transition relation defined as follows:

- $\langle \bar{l}, u\rangle \xrightarrow{\tau} \langle \bar{l}[l_i'/l_i], r_i(u)\rangle$ if there exist $l_i \in \bar{l}, g_i, r_i$ such that $l_i \overset{g_i,\tau,r_i}{\longrightarrow} l_i'$ and $g_i(u)$
- $\langle \bar{l}, u\rangle \xrightarrow{\tau} \langle \bar{l}[l_i'/l_i, l_j'/l_j], (r_i \cup r_j)(u)\rangle$ if there exist $l_i, l_j \in \bar{l}, g_i, g_j, \alpha, r_i$ and r_j such that $i \neq j, l_i \overset{g_i,\alpha!,r_i}{\longrightarrow} l_i'$, $l_j \overset{g_j,\alpha?,r_j}{\longrightarrow} l_j'$, $g_i(u)$, $g_j(u)$, and $r_i \cup r_j \in \mathcal{R}$
- $\langle \bar{l}, u\rangle \xrightarrow{\epsilon(d)} \langle \bar{l}, u \oplus d\rangle$ if $I(l_i)(u+d)$ for all $l_i \in \bar{l}$.

Note that the timed transition system defined above can also be represented finitely as a timed automaton. In fact, one may effectively construct the product automaton of $A_1 \dots A_n$ such that its timed transition system is bisimilar to $\overline{\mathcal{S}}$. The nodes of the product automaton is simply the product of A_i's nodes, the invariant conditions on the product nodes are the conjunctions of the conditions on all A_i's nodes, the set of clocks is the (disjoint) union of A_i's clocks, and the edges are based on synchronizable A_i's edges with enabling conditions conjuncted and reset-sets unioned.

Thus theoretically, there is no difference between the notions of a timed automaton and a network of such. However, for efficient verification, it is often not necessary to construct the product automaton. We shall distinguish them only in discussing verification methods, not when semantics aspects are concerned.

Finally, we denote by $Tr(\overline{A})$ all non–zeno traces of the timed transition system $\overline{\mathcal{S}}$ i.e. $Tr(\overline{A}) = Tr(\overline{\mathcal{S}})$.

3 A Logic for Safety and Bounded Response Time Properties

At the start of the project, we found that it was not so obvious how to formalize (in the Uppaal logic) the pages of informal requirements delivered by the design engineers. One of the reasons was that our logic is too simple, which can express essentially only invariant properties. After a while, it became obvious that these requirements could be described in a simple logic, which can be model–checked by reachability analysis in combination with a certain syntactical manipulation on the model of the system to be verified. We also noticed that though the logic is so simple, it characterizes the class of logical properties verified in all previous case studies where Uppaal is applied (see e.g. [BGK+96, JLS96, SMF97]).

3.1 Syntax and Semantics

The logic may be seen as a timed variant of a fragement of the linear temporal logic LTL, which does not allow nested applications of modal operators. It is to express invariant and bounded response time properties.

[5] We shall require that $P(A_i) \cap P(A_j) = \emptyset$ for all $i \neq j$, that is, no atomic proposition can be true in more than one components automata.

$(l,u) \models g$ *iff* $g(u)$
$(l,u) \models p$ *iff* $p \in V(l)$
$(l,u) \models \neg \mathsf{f}$ *iff* $(l,u) \not\models \mathsf{f}$
$(l,u) \models \mathsf{f}_1 \wedge \mathsf{f}_2$ *iff* $(l,u) \models \mathsf{f}_1$ and $(l,u) \models \mathsf{f}_2$
$\sigma \models Inv(\mathsf{f})$ *iff* $\forall i : \sigma[i] \models \mathsf{f}$
$\sigma \models \mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$ *iff* $\forall i : (\sigma[i] \models \mathsf{f}_1 \Rightarrow \exists k \geq i : (\sigma[k] \models \mathsf{f}_2$ and $D(\sigma, i, k) \leq T))$

Table 1. Definition of satisfiability.

Definition 4. *Assume that P is a finite set of propositions ranged over by p, q etc. Let $\mathcal{F}_s$ denote the set of boolean expressions over $\mathcal{GV} \cup P$ ranged over by $\mathsf{f}, \mathsf{f}_1, \mathsf{f}_2$ etc, defined as follows:*

$$\mathsf{f} ::= g \mid p \mid \neg \mathsf{f} \mid \mathsf{f}_1 \wedge \mathsf{f}_2$$

where $g \in \mathcal{GV}$ is a constraint. and $p \in P$ is an atomic proposition. We call $\mathcal{F}_s$ state–formulas, meaning that they will be true of states. □

As usual, we use $\mathsf{f}_1 \vee \mathsf{f}_2$ to stand for $\neg(\neg \mathsf{f}_1 \wedge \neg \mathsf{f}_2)$, and $\mathtt{tt}$ and $\mathtt{ff}$ for $\neg \mathsf{f} \vee \mathsf{f}$ and $\neg \mathsf{f} \wedge \mathsf{f}$ respectively. Further, we use $\mathsf{f}_1 \Rightarrow \mathsf{f}_2$ to denote $\neg \mathsf{f}_1 \vee \mathsf{f}_2$.

Definition 5. *The set $\mathcal{F}_t$ ranged over by $\mathsf{f}, \mathsf{f}_1, \mathsf{f}_2$ of trace–formulas over $\mathcal{F}_s$ is defined as follows:*

$$\varphi ::= Inv(\mathsf{f}) \mid \mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$$

where T is a natural number.

If f_1 and f_2 are boolean combinations of atomic propositions, we call $\mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$ a bounded response time formula. □

$Inv(\mathsf{f})$ states that f is an invariant property; a system satisfies $Inv(\mathsf{f})$ if all its reachable states satisfy f. It is useful to express safety properties, that is, *bad* things (e.g. deadlocks) should never happen, in other words, the system should always behave safely. $\mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$ is similar to the strong Until-operator in LTL, but with an explict time bound. In addition to the time bound, it is also an invariant formula. It means that as soon as f_1 is true of a state, f_2 must be true within T time units. However it is not necessary that f_1 must be true continously before f_2 becomes true as required by the traditional Until-operator.

We shall call formulas of the form $\mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$ *a bounded response time formula*. Intuitively, f_1 may be considered as a *request* and f_2 as a *response*; thus $\mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$ specifies the bound for the response time to be T.

We interpret $\mathcal{F}_s$ and $\mathcal{F}_t$ in terms of states and (infinite and non–zeno) traces of timed automata. We write $(l,u) \models \mathsf{f}$ to denote that the state (l,u) satisfies the state–formula f and $\sigma \models \varphi$ to denote that the trace σ satisfies the trace–formula φ. The interpretation is defined on the structure of f and φ, given in Table 1. Naturally, if all the traces of a timed automaton satisfy a trace–formula, we say that the automaton satisfies the formula.

Definition 6. *Assume a network of automata $\overline{A}$ and a trace–formula φ. We write $\overline{A} \models \varphi$ if and only if $\sigma \models \varphi$ for all $\sigma \in Tr(\overline{A})$.* □

3.2 Verifying Bounded Response Time Properties by Reachability Analysis

The current version of UPPAAL can only model–check invariant properties by reachability analysis. The question is how to use a tool like UPPAAL to check for bounded response time properties i.e. how to transform the model–checking problem $A \models \mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$ to a reachability problem. The traditional

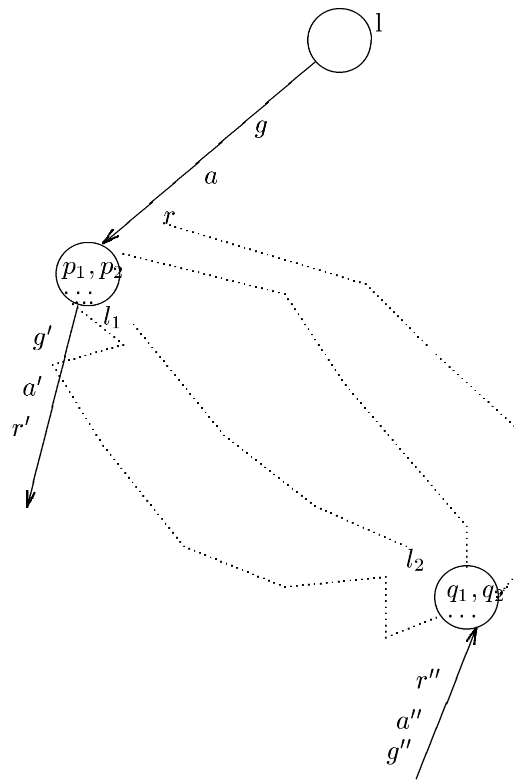

Fig. 1. Illustration of a timed automaton A.

solution is to translate the formula to a testing automaton t (see e.g. [JLS96]) and then check whether the parallel system $A||t$ can reach a designated state of t.

We take a different approach. We modify (or rather decorate) the automaton A according to the state-formulas f_1 and f_2, and the time bound T and then construct a state–formula f such that

$$\mathcal{M}(A) \models \mathit{Inv}(\mathsf{f}) \;\; \textit{iff} \;\; A \models \mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$$

where $\mathcal{M}(A)$ is the modified version of A.

We study an example. First assume that each node of an automaton is assigned implicitly a proposition $\mathsf{at}(l)$ meaning that the current control node is l. Consider an automaton A illustrated in Figure 1 and a formula $\mathsf{at}(l_1) \leadsto_{\leq 3} \mathsf{at}(l_2)$ (i.e. it should always reach l_2 from l_1 within 3 time units). To check whether A satisfies the formula, we introduce an extra clock $\mathsf{c} \in \mathcal{C}$ and a boolean variable [6] v_1 into the automaton A, that should be initiated with ff. Assume that the node l_1 has no local loops, i.e. containing no edges leaving and entering l_1. We modify the automaton A as follows:

1. Duplicate all edges entering node l_1.
2. Add $\neg\mathsf{v}_1$ as a guard to the original edges entering l_1.
3. Add $\mathsf{v}_1 := \mathsf{tt}$ and $\mathsf{c} := 0$ as reset–operations to the original edges entering l_1.
4. Add v_1 as a guard to the auxiliary copies of the edges entering l_1.
5. Add $\mathsf{v}_1 := \mathsf{ff}$ as a reset–operation to all the edges entering l_2.

The modified (decorated) automaton $\mathcal{M}(A)$ is illustrated in Figure 2. Now we claim that

$$\mathcal{M}(A) \models \mathit{Inv}(\mathsf{v}_1 \Rightarrow \mathsf{c} \leq 3) \;\; \textit{iff} \;\; A \models \mathsf{at}(l_1) \leadsto_{\leq 3} \mathsf{at}(l_2)$$

[6] Note that a boolean variable may be represented by an integer variable in Uppaal.

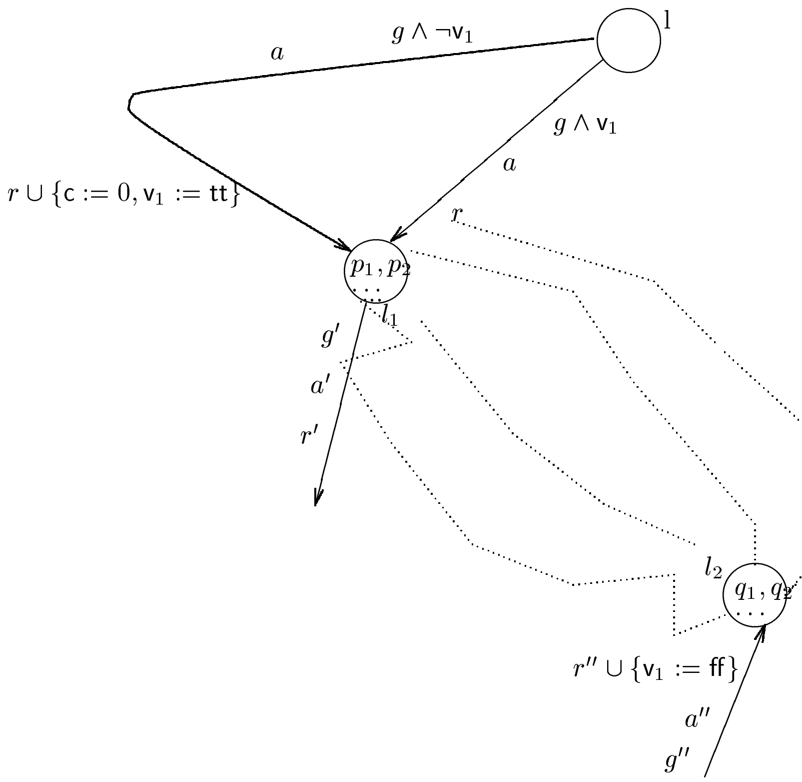

Fig. 2. Illustration of a modified timed automaton $\mathcal{M}(A)$ of A.

The invariant property $\mathsf{v}_1 \Rightarrow \mathsf{c} \leq 3$ states that either $\neg\mathsf{v}_1$ or if v_1 then $\mathsf{c} \leq 3$. There is only one situation that violates the invariant: v_1 and $\mathsf{c} > 3$. Due to the progress property of time (or non–zenoness), the value of c should always increase. It will sooner or later pass 3. But if l_2 is reached before c reaches 3, v_1 will become ff. Therefore, the only way to keep the invariant property true is that l_2 is reached within 3 time units whenever l_1 is reached.

The above method may be generalized to efficiently model–check response time formulas for networks of automata. Let $\mathcal{A}(\mathsf{f})$ denote the set of atomic propositions occuring in a state–formula f. Assume a network $\overline{A}$ and a response time formula $\mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$ For simplicity, we consider the case when only atomic propositions occur in f_1 and f_2. Note that this is not a restriction, the result can be easily extended to the generl case. We introduce to $\overline{A}$:

1. an auxiliary clock $\mathsf{c} \in \mathcal{C}$ and an boolean variable v_1 (to denote the truth value of f_1) and
2. an auxiliary boolean variable v_p for all $p \in \mathcal{A}(\mathsf{f}_1) \cup \mathcal{A}(\mathsf{f}_2)$.

Assume that all the booleans of $\mathcal{A}(\mathsf{f}_1), \mathcal{A}(\mathsf{f}_2)$ and v_1 are initiated to ff.

Let $\mathcal{E}(\mathsf{f})$ denote the boolean expression by replacing all $p \in \mathcal{A}(\mathsf{f})$ with their corresponding boolean variable v_p. As usual, $\mathcal{E}(\mathsf{f})[\mathsf{tt}/\mathsf{v}_p]$ denotes a substitution that replaces v_p with tt in $\mathcal{E}(\mathsf{f})$. This can be extended in the usual way to set of substitutions. For instance, the truth value of f at a given state s may be calculated by $\mathcal{E}(\mathsf{f})[\mathsf{tt}/\mathsf{v}_p|p \in V(s)][\mathsf{ff}/\mathsf{v}_p|p \notin V(s)]$.

Now we are ready to construct a decorated version $\mathcal{M}(\overline{A})$ for the network $\overline{A}$. We modify all the components A_i of $\overline{A}$ as follows:

1. For all edges of A_i, entering a node l_1 such that $V(l_1) \cap \mathcal{A}(\mathsf{f}_1) \neq \emptyset$:
   - Make two copies of each such edge.
   - To the original edge, add v_1 as a guard.

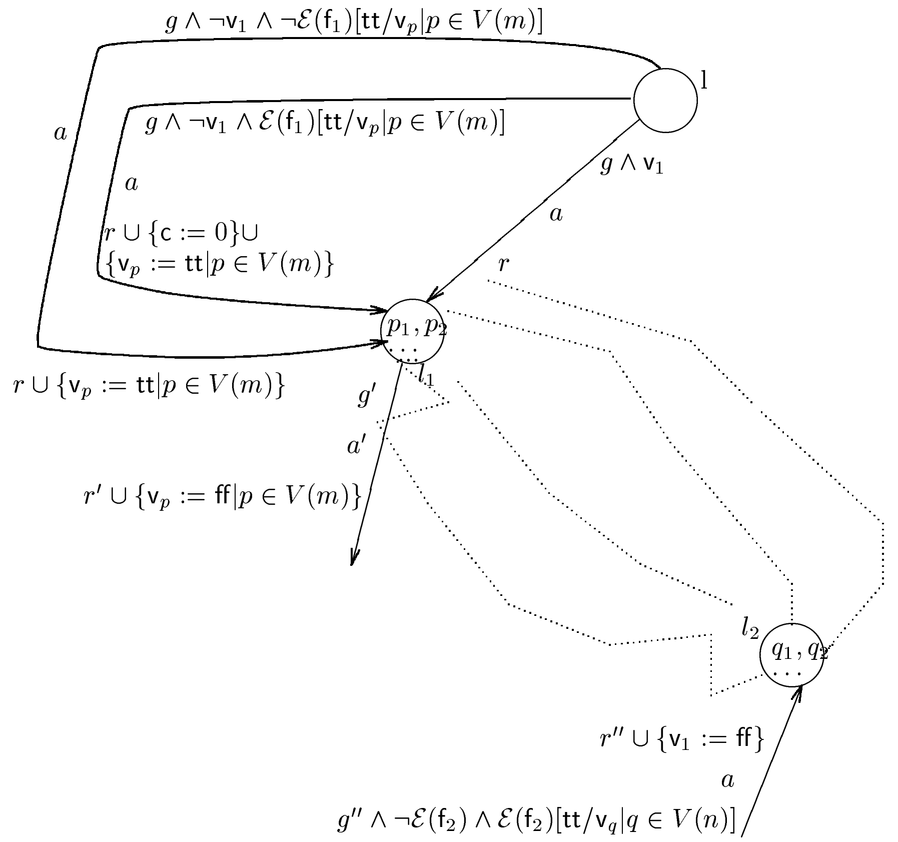

Fig. 3. Illustration of the decorated version $\mathcal{M}(A_i)$ of A_i.

  - To the first copy, add $\neg \mathcal{E}(\mathsf{f}_1) \wedge \mathcal{E}(\mathsf{f}_1)[\mathsf{tt}/\mathsf{v}_p | p \in V(l_1)]$ as a guard and $\mathsf{c} := 0, \mathsf{v}_1 := \mathsf{tt}$ and $\mathsf{v}_p := \mathsf{tt}$ for all $p \in V(l_1)$ as reset–operations.
  - To the second copy, add $\neg \mathsf{v}_1 \wedge \neg \mathcal{E}(\mathsf{f}_1)[\mathsf{tt}/\mathsf{v}_p | p \in V(l_1)]$ as a guard and $\mathsf{v}_p := \mathsf{tt}$ for all $p \in V(l_1)$ as reset–operations.
2. For all edges of A_i leaving a node l_1 such that $V(l_1) \cap \mathcal{A}(\mathsf{f}_1) \neq \emptyset$: add $\mathsf{v}_p := \mathsf{ff}$ for all $p \in V(l_1)$ as reset–operations.
3. For all edges of A_i entering a node l_2 such that $V(l_2) \cap \mathcal{A}(\mathsf{f}_2) \neq \emptyset$: add $\neg \mathcal{E}(\mathsf{f}_2) \wedge \mathcal{E}(\mathsf{f}_2)[\mathsf{tt}/\mathsf{v}_q | q \in V(l_2)]$ as a guard and $\mathsf{v}_1 := \mathsf{ff}$ as a reset–operation.
4. Finally, remove $\mathsf{v}_p := \mathsf{tt}$ and $\mathsf{v}_p := \mathsf{ff}$ whenever they occur at the same edge [7].

Thus, we have a decorated version $\mathcal{M}(A_i)$ for each A_i of $\overline{A}$. Assume that a component automaton A_i is as illustrated in Figure 1; we show its decorated version $\mathcal{M}(A_i)$ in Figure 3. We shall take $\mathcal{M}(A_1) || \dots || \mathcal{M}(A_n)$ to be the decorated version of $\overline{A}$, i.e. $\mathcal{M}(\overline{A}) \equiv \mathcal{M}(A_1) || \dots || \mathcal{M}(A_n)$.

Note that we could have constructed the product automaton of $\overline{A}$ first. Then the construction of $\mathcal{M}(\overline{A})$ from the product automaton would be much simpler. But the size of $\mathcal{M}(\overline{A})$ will be much larger; it will be exponential in the size of the component automata. Our construction here is purely syntactical based on the syntactical structure of each component automaton. The size of $\mathcal{M}(\overline{A})$ is in fact linear in the size of the component automata. It is particularly appropriate for a tool like UPPAAL, that is based on on-the-fly generation of the state–space of a network. For each component automaton A, the size of $\mathcal{M}(A)$ can be calculated precisely as follows: In addition to one auxiliary clock c and

[7] This means that a proposition p is assigned to both the source and the target nodes of the eadge; v_p must have been assigned tt on all the edges entering the source node.

$|P(\mathsf{f}_1) \cup P(\mathsf{f}_2)|$ boolean variables in $\mathcal{M}(A)$, the number of edges of $\mathcal{M}(A)$ is $3 \times |E_A|$ where $|E_A|$ is the number of edges of A (note that no extra nodes introduced in $\mathcal{M}(A)$).

Note also that in the above construction, we have the restriction that f_1 and f_2 contain no constraints, but only atomic propositions. The construction can be easily generalized to allow constraints by considering each constraint as a proposition and decorating each location (that is, the incomming edges) where the constraint could become true when the location is reached. In fact, this is what we did above on the boolean expressions (constraints) $\mathcal{E}(\mathsf{f}_1)$ and $\mathcal{E}(\mathsf{f}_2)$. Finally, we have the main theoretical result of this paper.

Theorem 7. *$\mathcal{M}(\overline{A}) \models Inv(\mathsf{v}_1 \Rightarrow \mathsf{c} \leq T)$ iff $\overline{A} \models \mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$ for a network of timed automata $\overline{A}$ and a bounded response time formula $\mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2$.* □

4 The Gear Controller

In this section we informally describe the functionality and the requirements of the gear controller proposed by Mecel AB, as well as the abstract behavior of the environment where the controller is supposed to operate.

4.1 Functionality

The gear controller changes gears by requesting services provided by the components in its environment: the gear-box, the clutch, and the engine. The interaction with these components is over the vehicles communication network. Similarly, the gear controller provides services to its users through its interface. A description of the gear controller and its interface is as follows.

Interface: The interface receives service requests and keeps information about the current status of the gear controller, which is either changing gear or idling. The user of this service is either the driver using the gear stick or a dedicated component implementing a gear change algorithm. The interface is assumed to respond when the service is completed.

Gear Controller: The only user of the gear controller is its interface. The controller performs a gear change in five steps beginning when a gear change request is received from the interface. The first step is to accomplish a zero torque transmission, making it possible to release the currently set gear. Secondly the gear is released. The controller then achieves synchronous speed over the transmission and sets the new gear. Once the gear is set the engine torque is increased so that the same wheel torque level as before the gear change is achieved.
Under difficult driving conditions the engine may not be able to accomplish zero torque or synchronous speed over the transmission. It is then possible to change gear using the clutch. By opening the clutch, and consequently the transmission, the connection between the engine and the wheels is broken. The gear-box is at this state able to release and set the new gear, as zero torque and synchronous speed is no longer required. When the clutch closes it safely bridges the speed and torque differences between the engine and the wheels. We refer to these exceptional cases as *recoverable errors*.

4.2 Environment

The environment of the gear controller consists of the following three components:

Gear-Box: It is an electrically controlled gear-box with control electronics. It provides services to *set* a gear in 100 to 300 ms and to *release* a gear in 100 to 200 ms. If a setting or releasing-operation of a gear takes more than 300 ms or 200 ms respectively, the gear-box will indicate this and stop in a specific error state.

Clutch: It is an electrically controlled clutch that has the same sort of basic services as the gear-box. The clutch can *open* or *close* within 100 to 150 ms. If a opening or closing is not accomplish within the time bounds, the clutch will indicate this and reach a specific error state.

Engine: It offers three modes of control: normal torque, zero torque, and synchronous speed. The normal mode of operation is *normal torque* mode where the engine gives the requested engine torque. In *zero torque* mode the engine will try to find a zero torque difference over the transmission. Similarly, in *synchronous speed* mode the engine searches zero speed difference between the engine and the wheels[8].

The engine have an angular acceleration without load that can reach 10000 rpm/s and is controlled twice per lap though it is a four cylinder engine. The decrease of the angular velocity is achieved by removing the fuel, ignition and air. As a result, the engine retards through its friction. The decrease of the angular velocity is generally 4 to 5 times less than the angular acceleration and has a linear dependability on the friction. This asymmetric difference between acceleration and retardation of the angular velocity is difficult to control especially when the engine is running without load, because the difference then is maximal. Due to these circumstances the maximum time bound searching for zero torque is limited to 400 ms within which a safe state is entered. Furthermore, the maximum time bound for synchronous speed control is limited to 500 ms. If 500 ms elapse the engine enters an error state.

We will refer the error states in the gear-box, the clutch and the engine as *unrecoverable errors* since it is impossible for the gear controller alone to recover from these errors.

4.3 Requirements

In this section we list the informal requirements and desired functionality on the gear controller, provided by Mecel AB. The requirements are to ensure the correctness of the gear controller. A few operations, such as gear shifts and error detections, are crucial to the correctness. They must be guaranteed within certain time bounds. In addition, there are also requirements on the controller to ensure desired qualities of the vehicle, such as: good comfort, low fuel consumption, and low emission.

1. **Performance.** These requirements limit the maximum time to perform a gear shift when no unrecoverable errors occur.
   (a) A gear shift should be completed within 1.5 seconds.
   (b) A gear shift, under normal operation conditions, should be performed within 1 second.
2. **Predictability**. The predictability requirements are to ensure strict synchronization and control between components.
   (a) There should not be dead-locks or live-locks in the system.
   (b) When the engine is regulating torque, the clutch should be closed.
   (c) The gear has to be set in the gear-box when the engine is regulating torque.
3. **Functionality.** The following requirements are to ensure the desired functionality of the gear controller.
   (a) It has one reverse gear, five forward gears and one neutral gear.
   (b) It is able to use all gears.
   (c) It uses the engine to enhance zero torque and synchronous speed over the transmission.
   (d) It uses the gear-box to set and release gears.
   (e) It is allowed to use the clutch in difficult conditions.
   (f) It does not request zero torque when shifting from neutral gear.
   (g) The gear controller does not request synchronous speed when shifting to neutral gear.
4. **Error Detection.** The gear controller detects and indicates error only when:
   (a) the clutch is not opened in time,
   (b) the clutch is not closed in time,

[8] Synchronous speed mode is used only when the clutch is open or no gear is set.

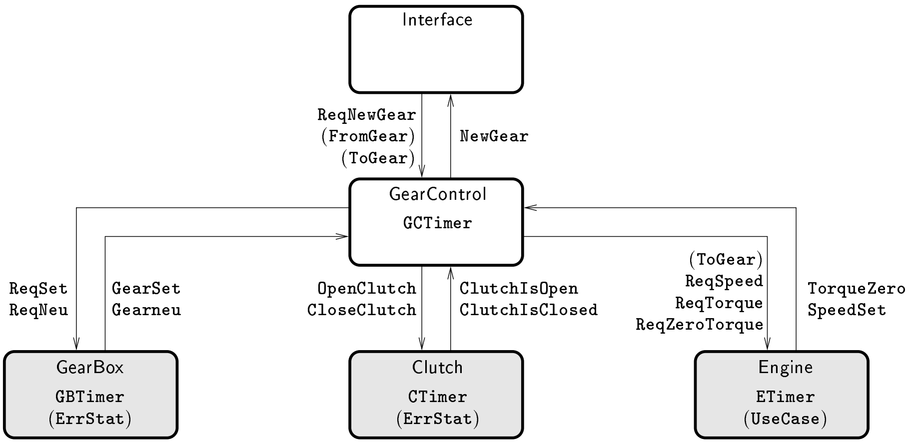

Fig. 4. A Flow-Graph of the Gear-Box System.

(c) the gear-box is not able to set a gear in time,
(d) the gear-box is not able to release a gear in time.

5 Formal Description of the System

To design and analyze the gear controller we model the controller and its environment in the UPPAAL model. The modeling phase has been separated in two steps. First a model of the environment is created, as its behavior is specified in advance as assumptions (see Section 4.2). Secondly, the controller itself and its interface are designed to be functionally correct in the given environment. Figure 4 shows a flow-graph of the resulting model where nodes represent automata and edges represent synchronization channels or shared variables (enclosed within parenthesis). The gear controller and its interface are modeled by the automata GearControl (GC) and Interface (I). The environment is modeled by the three automata: Clutch (C), Engine (E), and GearBox (GB).

The system uses six variables. Four are timers (i.e. real-valued clocks) that measure 1/1000 of seconds (ms): `GCTimer`, `GBTimer`, `CTimer` and `ETimer`. The two other variables, named `FromGear` and `ToGear`, are used at gear change requests[9]. In the following we describe the five automata of the system.

5.1 Environment

The three automata of the environment model the basic functionality and time behavior of the components in the environment. The components have two channels associated with each service: one for requests and one to respond when service have been performed.

Gear-Box: In automaton GearBox, shown in Figure 6, inputs on channel `ReqSet` request a gear set and the corresponding response on `GearSet` is output if the gear is successfully set. Similarly, the channel `ReqNeu` requests the neutral gear and the response `GearNeu` signals if the gear is successfully released. If the gear-box fails to set or release a gear the locations named ErrorSet and ErrorNeu are entered respectively.

Clutch: The automaton Clutch is shown in Figure 7. Inputs on channels `OpenClutch` and `CloseClutch` instruct the clutch to open and close respectively. The corresponding response channels are `ClutchIsOpen` and `ClutchIsClosed`. If the clutch fails to open or close it enters the location named ErrorOpen and ErrorClose respectively.

[9] The domains of `FromGear` and `ToGear` are bounded to $\{0, ..., 6\}$, where 1 to 5 represent gear 1 to gear 5, 0 represents gear N, and 6 is the reverse gear.

Engine: The automaton Engine, shown in Figure 8 accepts incoming requests for synchronous speed, a specified torque level or zero torque on the channels `ReqSpeed`, `ReqTorque` and `ReqZeroTorque` respectively. The actual torque level or speed being requested is not modeled since it does not affect the design of the gear controller[10]. The engine responds to the requests on the channels `TorqueZero` and `SpeedSet` when the services have been completed. Requests for specific torque levels (i.e. signal `ReqTorque`) are not answered, instead torque is assumed to increase immediately after the request.

The engine may fail to deliver zero torque or synchronous speed in time as described in Section 4.2. It will then enter a location named ClutchOpen without responding to the request. A more dangerous scenario will occur if the engine regulates on synchronous speed in too long time. To avoid damage a timeout interrupts the engine after 500 ms of regulation and a location named ErrorSpeed is entered.

5.2 Functionality

In this section we describe the model of the designed gear controller and its interface. Given the formal model of the environment, the gear controller have been designed both to satisfy the correctness requirements given in Section 4.3, and the functionality requirements in Section 4.1.

Gear Controller: The GearControl automaton is shown in Figure 5. Each main loop implements a gear change by interacting with the components of the environment.

The designed controller measures response times (using the timer `GCTimer`) from the components to detect errors (as failures are not signaled). The reaction of the controller depends on how serious the occurred error is. It either recovers the system from the error, or terminates in a pre-specified location that points out the (unrecoverable) error: COpenError, CCloseError, GNeuError or GSetError. Recoverable errors are detected in the locations CheckTorque and CheckSyncSpeed.

Interface: The automaton Interface, shown in Figure 9, requests gears R, N, 1, ..., 5 from the gear controller. Requests and responses are sent through channel `ReqNewGear` and channel `NewGear` respectively. When a request is sent, the shared variables `FromGear` and `ToGear` are assigned values corresponding to the current and the requested new gear respectively.

6 Formal Validation and Verification

In this section we formalize the informal requirements given in Section 4.3 and prove their correctness using the symbolic model-checker of Uppaal.

6.1 System Decoration

To enable formalization (and verification) of requirements, we decorate the system description with two integer variables, `ErrStat` and `UseCase`. The variable `ErrStat` is assigned values at unrecoverable errors: 1 if Clutch fails to close, 2 if Clutch fails to open, 3 if GearBox fails to set a gear, and 4 if GearBox fails to release a gear. The variable `UseCase` is assigned values whenever a recoverable error occurs in Engine: 1 if it fail to deliver zero torque, and 2 if it is not able to find synchronous speed. The system model is also decorated to enable verification of bounded response time properties, as described in Section 3.2.

[10] Hence, the time bound for finding zero torque (i.e. 400 ms) should hold when decreasing from an arbitrary torque level.

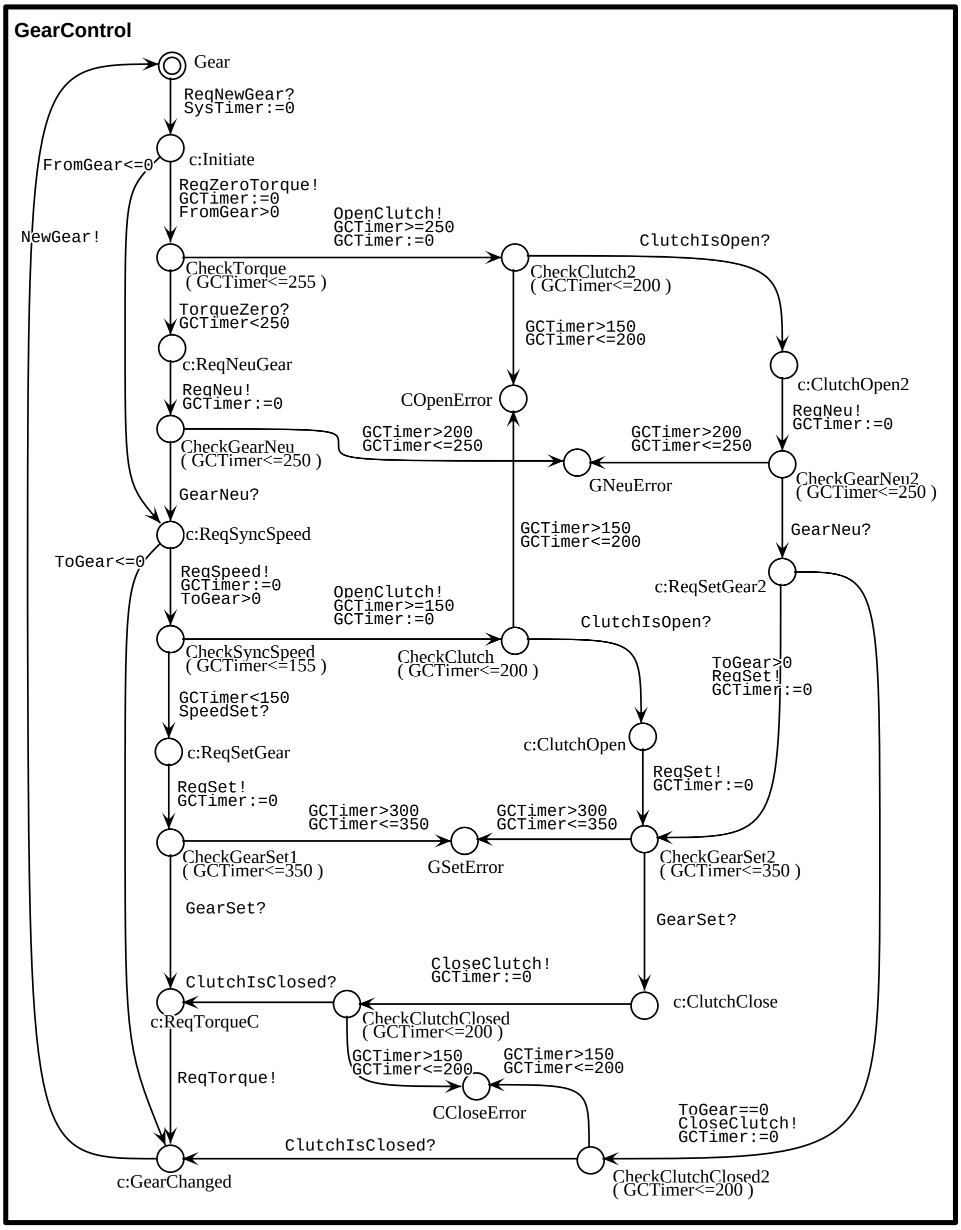

Fig. 5. The Gear Box Controller Automaton.

6.2 Requirement Specification

Before formalizing the requirement specification of the gear controller we define negation and conjunction for the bounded response time operator and the invariant operator defined in Section 3.2,

$$\overline{A} \models \varphi_1 \wedge \varphi_2 \textit{ iff } \overline{A} \models \varphi_1 \text{ and } \overline{A} \models \varphi_2$$
$$\overline{A} \models \neg\varphi \textit{ iff } \overline{A} \not\models \varphi$$

$$\mathsf{GearControl@Initiate} \leadsto_{\leq 1500} ((\mathtt{ErrStat} = 0) \Rightarrow \mathsf{GearControl@GearChanged}) \tag{1}$$

$$\mathsf{GearControl@Initiate} \leadsto_{\leq 1000} ((\mathtt{ErrStat} = 0 \wedge \mathtt{UseCase} = 0) \Rightarrow \mathsf{GearControl@GearChanged}) \tag{2}$$

$$\mathsf{Clutch@ErrorClose} \leadsto_{\leq 200} \mathsf{GearControl@CCloseError} \tag{3}$$

$$\mathsf{Clutch@ErrorOpen} \leadsto_{\leq 200} \mathsf{GearControl@COpenError} \tag{4}$$

$$\mathsf{GearBox@ErrorIdle} \leadsto_{\leq 350} \mathsf{GearControl@GSetError} \tag{5}$$

$$\mathsf{GearBox@ErrorNeu} \leadsto_{\leq 200} \mathsf{GearControl@GNeuError} \tag{6}$$

$$Inv (\mathsf{GearControl@CCloseError} \Rightarrow \mathsf{Clutch@ErrorClose}) \tag{7}$$

$$Inv (\mathsf{GearControl@COpenError} \Rightarrow \mathsf{Clutch@ErrorOpen}) \tag{8}$$

$$Inv (\mathsf{GearControl@GSetError} \Rightarrow \mathsf{GearBox@ErrorIdle}) \tag{9}$$

$$Inv (\mathsf{GearControl@GNeuError} \Rightarrow \mathsf{GearBox@ErrorNeu}) \tag{10}$$

$$Inv (\mathsf{Engine@ErrorSpeed} \Rightarrow \mathtt{ErrStat} \neq 0) \tag{11}$$

$$Inv (\mathsf{Engine@Torque} \Rightarrow \mathsf{Clutch@Closed}) \tag{12}$$

$$\bigwedge_{i \in \{R,N,1,\ldots,5\}} Poss (\mathsf{Gear@Gear}_i) \tag{13}$$

$$\bigwedge_{i \in \{R,1,\ldots,5\}} Inv ((\mathsf{GearControl@Gear} \wedge \mathsf{Gear@Gear}_i) \Rightarrow \mathsf{Engine@Torque}) \tag{14}$$

Table 2. Requirement Specification

We also extend the (implicit) proposition $\mathsf{at}(l)$ to $\mathsf{at}(A, l)$, meaning that the control location of automaton A is currently l. Finally, we introduce $Poss(\mathsf{f})$ to denote $\neg Inv(\neg \mathsf{f})$, $\mathsf{f}_1 \not\leadsto_{\leq T} \mathsf{f}_2$ to denote $\neg(\mathsf{f}_1 \leadsto_{\leq T} \mathsf{f}_2)$, and $A@l$ to denote $\mathsf{at}(A, l)$. We are now ready to formalize the requirements.

The first performance requirement 1a, i.e. that a gear change must be completed within 1.5 seconds given that no unrecoverable errors occur, is specified in property 1. It requires the location GearChanged in automaton GearControl to be reached within 1.5 seconds after location Initiate has been entered. Only scenarios without unrecoverable errors are considered as the value of the variable `ErrStat` is specified to be zero[11]. To consider scenarios with normal operation we restrict also the value of variable `UseCase` to zero (i.e. no recoverable errors occurs). Property 2 requires gear changes to be completed within one second given that the system is operating normally.

The properties 3 to 6 require the system to terminate in known error-locations that point out the specific error when errors occur in the clutch or the gear (requirements 4a to 4d). Up to 350 ms is allowed to elapse between the occurrence of an error and that the error is indicated in the gear controller. The properties 7 to 10 restrict the controller design to indicate an error *only* when the corresponding error has arised in the components. Observe that no specific location in the gear controller is dedicated to indicate the unrecoverable error that may occur when the engines speed-regulation is interrupted (i.e. when location Engine@ErrorSpeed is reached). Property 11 requires that no such location is needed since this error is always a consequence of a preceding unrecoverable error in the clutch or in the gear.

Property 13 holds if the system is able to use all gears (requirement 3a). Furthermore, for full functionality and predictability, the system is required to be dead-lock and live-lock free (requirement 2a). In this report, dead-lock and live-lock properties are not specified due to lack of space. However, property 1 (and 2) guarantee progress within bounded time if no unrecoverable error causes the system

[11] Recall that the variable `ErrStat` is assigned a positive value (i.e. greater than zero) whenever an unrecoverable error occurs.

$$\mathsf{GearControl@Initiate} \leadsto_{<900}$$
$$\big((\ \mathtt{ErrStat} = 0\ \wedge\ \mathtt{UseCase} = 0\ )\ \Rightarrow\ \mathsf{GearControl@GearChanged}\ \big) \tag{15}$$
$$\mathsf{GearControl@Initiate} \not\leadsto_{\leq 899}$$
$$\big((\ \mathtt{ErrStat} = 0\ \wedge\ \mathtt{UseCase} = 0\ )\ \Rightarrow\ \mathsf{GearControl@GearChanged}\ \big) \tag{16}$$

Table 3. Time Bounds

to terminate. The properties 12 and 14 specifies the informal predictability requirements 2b and 2c.

A number of functionality requirements specify how the gear controller should interact with the environment (e.g. 3a and 3c to 3g). These requirements have been used to give the gear controller the desired design. They have later been validated using the simulator in UPPAAL and have not been formally specified and verified.

6.3 Time Bound Derivation

Property 1 requires that a gear change should be performed within one second. Even though this is an interesting property in itself one may ask for the *lowest* time bound for which a gear change is *guaranteed*. We show that this time bound is 900 ms for error-free scenarios by proving that the change is guaranteed at 900 ms (property 15), and that the change is possibly *not* completed at 899 ms (property 16). Similarly, for scenarios when the engine fails to deliver zero torque we derive the bound 1055 ms, and if synchronous speed is not delivered in the engine the time bound is 1205 ms.

We have shown the shortest time for which a gear change is *possible* in the three scenarios to be: 150 ms, 550 ms, and 450 ms. However, gear changes involving neutral gear may be faster as the gear does not have to be released (when changing from gear neutral) or set (when changing to gear neutral). Finally we consider the same three scenarios but without involving neutral gear by constraining the values of the variables `FromGear` and `ToGear`. The derived time bounds are: 400 ms, 700 ms and 750.

6.4 Verification Results

We have verified totally 46 properties of the system[12] using UPPAAL installed on a 75 MHz Pentium PC equipped with 24 MB of primary memory. The verification of all the properties consumed 2.99 second.

7 Conclusion

In this paper, we have reported an industrial case study in applying formal techniques for the design and analysis of control systems for vehicles. The main output of the case-study is a formally described gear controller and a set of formal requirements. The designed controller has been validated and verified using the tool UPPAAL to satisfy the safety and functionality requirements on the controller, provided by Mecel AB. It may be considered as one piece of evidence that the validation and verification tools of today are mature enough to be applied in industrial projects.

We have given a detailed description of the formal model of the gear controller and its surrounding environment, and its correctness formalized in 46 logical formulas according to the informal requirements delivered by industry. The verification was performed in a few seconds on a Pentium PC[13] running UPPAAL version 2.12.2. Another contribution of this paper is a solution to a problem we got

[12] A complete list of the verified properties can be found in the full version of this paper.
[13] 2.99 seconds on a Pentium 75MHz equipped with 24 MB of primary memory.

in this case study, namely how to use a tool like UPPAAL, which only provides reachability analysis to verify bounded response time properties. We have presented a logic and a method to characterize and model–check such properties by reachability analysis in combination with simple syntactical manipulation on the system description.

This work concerns only one component, namely gear controller of a control system for vehicles. Future work, naturally include modelling and verification of the whole control system. The project is still in progress. We hope to report more in the near future on the project.

References

[AD94] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.

[BGK+96] Johan Bengtsson, David Griffioen, Kåre Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of 8th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.

[BLL+96] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Proc. of the 2nd Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 431–434. Springer–Verlag, Mars 1996.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Proc. of Workshop on Verification and Control of Hybrid Systems III*, Lecture Notes in Computer Science, pages 208–219. Springer–Verlag, October 1995.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: The Next Generation. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 56–65, December 1995.

[JLS96] H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL. In *Proc. of 2nd International Workshop on the SPIN Verification System*, pages 1–20, August 1996.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. To appear in *International Journal on Software Tools for Technology Transfer*, 1997.

[SMF97] Thomas Stauner, Olaf Müller, and Max Fuchs. Using hytech to verify an automotive control system. In *Proc. Hybrid and Real-Time Systems, Grenoble, March 26-28, 1997*. Technische Universität München, Lecture Notes in Computer Science, Springer, 1997.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the 7th International Conference on Formal Description Techniques*, 1994.

Appendix: The System Description

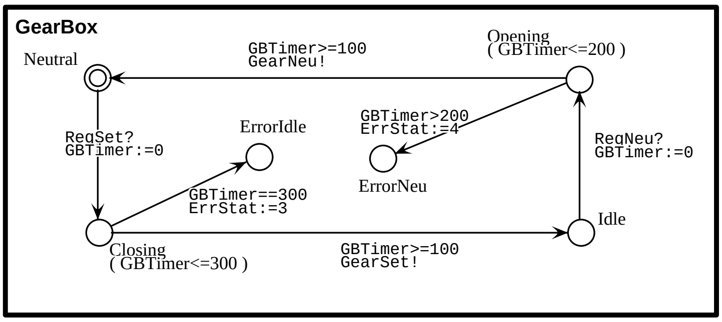

Fig. 6. The Gear-Box Automaton.

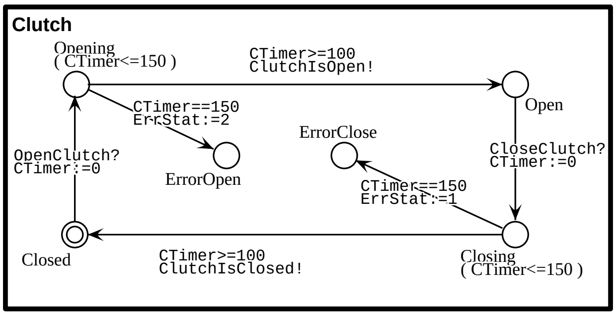

Fig. 7. The Clutch Automaton.

This article was processed using the LaTeX macro package with LLNCS style

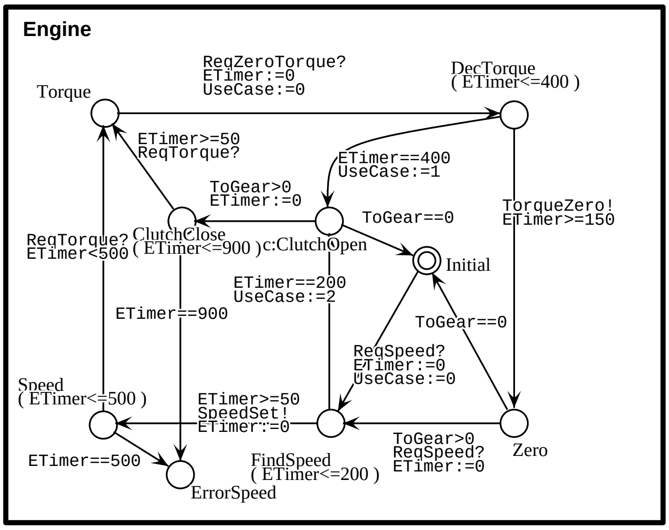

Fig. 8. The Engine Automaton.

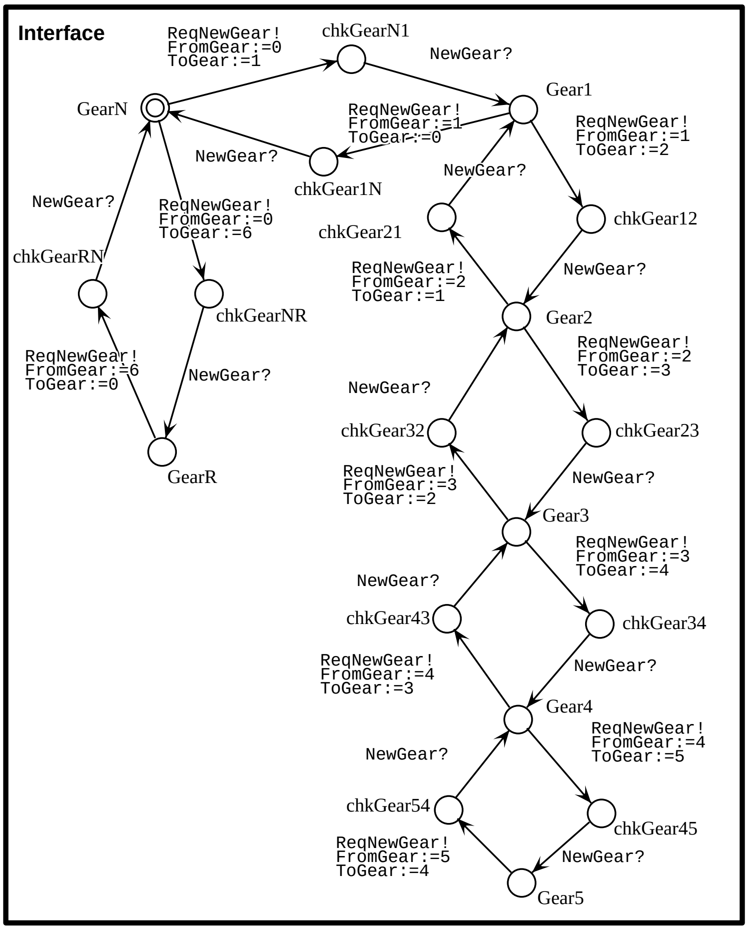

Fig. 9. The Interface Automaton.