Formal Design and Analysis of a Gear Controller

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

Mecel AB, Göteborg, Sweden. Email magnus.lindahl@mecel.se.
Department of Computer Systems, Uppsala University, Sweden. Email: {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. We give a detailed description of the formal model of the gear controller and its surrounding environment, and its correctness formalised 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. The advantage of our solution is that we need no additional implementation work to extend the existing model-checker, but simple manual syntactical manipulation on the system description.

1 Introduction

Over the past few years, a number of modeling and verification tools for real-time systems [5, 6, 7] have been developed based on the theory of timed automata [3]. They have been successfully applied in various case-studies [4, 9, 11]. 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 (or a dedicated component implementing a gear change algorithm) 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 under given assumptions on the environment where the gear controller is supposed to operate (i.e. the clutch, the engine, the gearbox, etc.). The requirements on the controller and assumptions on the environment have been described by Mecel AB in an informal document, and then formalised 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 characterise 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[1] 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, 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 47 logical formulas derived from the informal requirements specified by Mecel AB. The verification was performed in a few seconds on a Pentium PC [2] running Uppaal.

Related to the approach of checking bounded response time properties presented in this paper is the work on *reachability testing* due to Aceto et.al. [9, 1, 2]. In their approach, a logical formula of a safety and bounded-liveness logic is transformed into a testing automaton which is composed in parallel with the system description to check the validity of the formula by reachability analysis[3]. This may seems more attractive as no syntactical manipulations of the system description seem to be required. However, in practice it is often the case that the system description indeed must be manipulated to make all the system actions appearing in the logical formulae visible to the testing automaton. The reachability testing approach also requires some actions of the system description to be urgent (i.e. to be taken as early as possible) [1, 2]. This is not the case with our technique. In fact, in the presented case study we verify several bounded response time properties of the gear controller involving non-urgent behaviors.

The paper is organised as follows: In the next section we present a simple logic to characterise safety and response time properties and a method to model-check such properties. In section 5 and 6 the gear controller system and its requirements are informally and formally described. In section 7 the formal description of the system and its requirements are transformed using the technique developed in section 3 for verification by reachability analysis. Section 8 concludes the paper. Finally, as appendices, we enclose the list of requirements in the q-format and the formal descriptions for the whole system in the atg-format of Uppaal.

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 [12, 10].

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

Definition 1. *A* timed transition system *over $\mathcal{A}$ 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:

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

where $\alpha_i \in \Sigma$.

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

We use $\delta(s \xrightarrow{\alpha} s')$ to denote the duration of the transition, defined by $\delta(s \xrightarrow{\alpha} s') = 0$ if $\alpha \in \mathcal{A}$ 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 any natural number T there exists a position k such that $\Delta(\sigma, 0, k) > T$. 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:

[1] Further information on installation and documentation for Uppaal is available at http://www.uppaal.com/.

[2] 2.99 seconds on a Pentium 75MHz equipped with 24 MB of primary memory.

[3] The technique of reachability testing is related to the use of “never claims” in the verification tool Spin [8].

$x \sim n$ 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*. Similarily, we use $I(\mathcal{C})$ to stand for the set of conjunctivie guards of the form: $x < n$ or $x \leq n$, and call the elements of $I(\mathcal{C})$ *invariant conditions.*

To manipulate clock and data variables, we use reset–operations of the form: $v := e$ where v is a clock or data variable and e is an expression. A reset-operation on a clock variable should be in the form $x := 0$; a reset-operation on an integer variable is similar to an assignment statement in a high-level programming language in the form: $i := e$ where e is an arithmethic expression[4]. 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 $\mathcal{A}$, atomic propositions $\mathcal{P}$, and $\mathcal{V}$, is a tuple $\langle N, l_0, \longrightarrow, I, V \rangle$, where N is a finite set of nodes (or locations), l_0 is the initial node, and $\longrightarrow \subseteq N \times \mathcal{G}(\mathcal{V}) \times \mathcal{A} \times \mathcal{R} \times N$ corresponds to the set of edges. In the case, $\langle l, g, a, r, l' \rangle \in \longrightarrow$ we shall write, $l \xrightarrow{g,a,r} l'$. $I : N \to I(\mathcal{C})$ is a function which for each node assigns an invariant condition, and $V : N \to 2^{\mathcal{P}}$ 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 N of A, i.e. $P(A) = \bigcup_{l \in N} V(l)$.* □

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 $u'(v) = V(e, u)$ whenever $v := e \in r$ and $u'(v') = u(v')$ otherwise, where $V(e, u)$ denotes the value of e 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, $(A_1 || ... || A_n) \backslash \mathcal{A}$. 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:

- $\langle \bar{l}, u \rangle \longrightarrow \langle \bar{l}[l_i'/l_i], r_i(u) \rangle$ if there exist $l_i \in \bar{l}, g_i, r_i$ such that $l_i \xrightarrow{g_i, \tau, r_i} l_i'$ and $g_i(u)$
- $\langle \bar{l}, u \rangle \longrightarrow \langle \bar{l}[l_i'/l_i, l_j'/l_j], (r_i \cup r_j)(u) \rangle$ if there exist $l_i, l_j \in \bar{l}, g_i, g_j, \alpha, r_i$ and r_j

  such that $i \neq j, l_i \xrightarrow{g_i, \alpha!, r_i} l_i'$, $l_j \xrightarrow{g_j, \alpha?, r_j} l_j'$, $g_i(u)$, $g_j(u)$, 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 $A_1 \ldots A_n$ 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.

[4] For BNF definition of arithmethic expressions, we refer to the UPPAALhome page.

[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 component automaton.

$$
\begin{array}{rl}
(l,u) \models g & \textit{iff}\ g(u) \\
(l,u) \models p & \textit{iff}\ p \in V(l) \\
(l,u) \models \neg f & \textit{iff}\ (l,u) \not\models f \\
(l,u) \models f_1 \wedge f_2 & \textit{iff}\ (l,u) \models f_1 \text{ and } (l,u) \models f_2 \\
\sigma \models \mathsf{INV}(f) & \textit{iff}\ \forall i : \sigma[i] \models f \\
\sigma \models f_1 \leadsto_{\leq T} f_2 & \textit{iff}\ \forall i : (\sigma[i] \models f_1 \Rightarrow \exists k \geq i : (\sigma[k] \models f_2 \text{ and } \Delta(\sigma, i, k) \leq T))
\end{array}
$$

Table 1. Definition of Satisfiability.

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

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. It later 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. [4, 9]).

3.1 Syntax and Semantics

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

Definition 4 (State-Formulas). Assume that $\mathcal{C}$ is a set of clocks and P is a finite set of propositions. Let $\mathcal{F}_s$ denote the set of state-formulas over $\mathcal{C}$ and P ranged over by f, f_1, f_2 etc. defined as follows:

$$f \ ::= \ g \mid p \mid \neg f \mid f_1 \wedge f_2$$

where $p \in P$ is an atomic proposition and g is an atomic clock constraint in the form $x \sim n$ or $x - y \sim n$ for $x, y \in \mathcal{C}$, $\sim \in \{<, \leq, =, \geq, >\}$ and n being a natural number. □

As usual, we use $f_1 \vee f_2$ to stand for $\neg(\neg f_1 \wedge \neg f_2)$, and tt and ff for $\neg f \vee f$ and $\neg f \wedge f$ respectively. Further, we use $f_1 \Rightarrow f_2$ to denote $\neg f_1 \vee f_2$.

Definition 5 (Trace-Formulas). The set $\mathcal{F}_t$ ranged over by φ, ψ of trace-formulas over $\mathcal{F}_s$ is defined as follows:

$$\varphi \ ::= \ \mathsf{INV}(f) \mid f_1 \leadsto_{\leq T} f_2$$

where T is a natural number. If f_1 and f_2 are boolean combinations of atomic propositions, we call $f_1 \leadsto_{\leq T} f_2$ a bounded response time formula. □

$\mathsf{INV}(f)$ states that f is an invariant property. A system satisfies $\mathsf{INV}(f)$ 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. $f_1 \leadsto_{\leq T} f_2$ is similar to the strong Until-operator in LTL, but with an explicit 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 continuously before f_2 becomes true as required by the traditional Until-operator.

We shall call a formula of the form $f_1 \leadsto_{\leq T} f_2$ a *bounded response time formula*. Intuitively, f_1 may be considered as a *request* and f_2 as a *response*; thus $f_1 \leadsto_{\leq T} f_2$ 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 $(l, u) \models f$ 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 $\overline{A} \models \varphi$ iff $\sigma \models \varphi$ for all $\sigma \in Tr(\overline{A})$. □

4 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 $A \models f_1 \leadsto_{\leq T} f_2$ to a reachability problem. A standard solution is to translate the formula to a testing automaton t (see e.g. [9]) and then check whether the parallel system $A||t$ can reach a designated state of t.

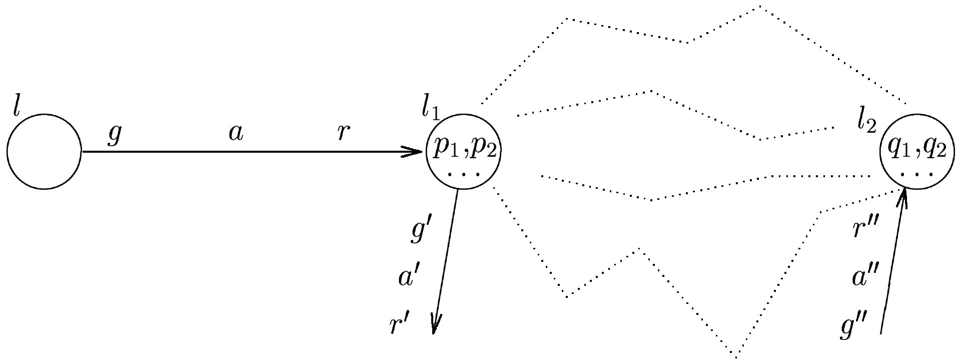

Fig. 1. Illustration of a timed automaton A.

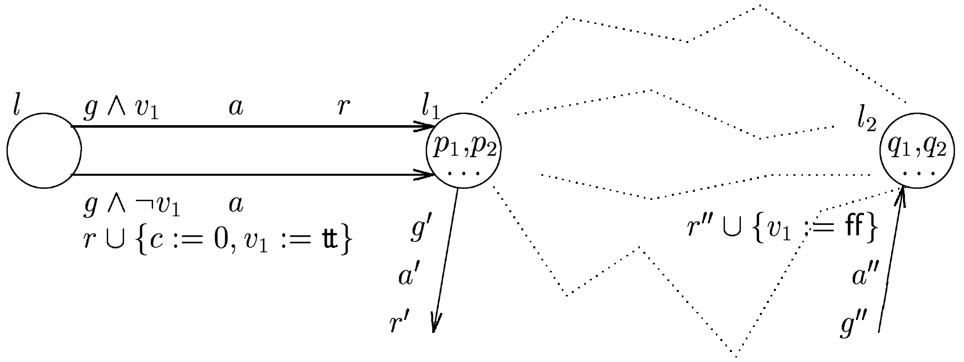

Fig. 2. Illustration of a modified timed automaton $\mathcal{M}(A)$ of A.

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

$$\mathcal{M}(A) \models \mathsf{INV}(f) \text{ iff } A \models f_1 \leadsto_{\leq T} f_2$$

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

We study an example. As usual, 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 $c \in \mathcal{C}$ 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 $\neg v_1$ as a guard to the original edges entering l_1.
3. Add $v_1 := \mathsf{tt}$ and $c := 0$ 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 $v_1 := \mathsf{ff}$ 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

$$\mathcal{M}(A) \models \mathsf{INV}(v_1 \Rightarrow c \leq 3) \text{ iff } A \models \mathsf{at}(l_1) \leadsto_{\leq 3} \mathsf{at}(l_2)$$

The invariant property $v_1 \Rightarrow c \leq 3$ states that either $\neg v_1$ or if v_1 then $c \leq 3$. There is only one situation that violates the invariant: v_1 and $c > 3$. 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 $\mathcal{P}(f)$ denote the set of atomic propositions occurring in a state-formula f. Assume a network $\overline{A}$ and a response time formula $f_1 \leadsto_{\leq T} f_2$. 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 general case which also allows clock constraints in f_1 and f_2. We introduce to $\overline{A}$ the following auxiliary variables:

1. an auxiliary clock $c \in \mathcal{C}$ and an boolean variable v_1 (to denote the truth value of f_1), and
2. an auxiliary boolean variable v_p for all $p \in \mathcal{P}(f_1) \cup \mathcal{P}(f_2)$.

Assume that all the booleans of $\mathcal{P}(f_1), \mathcal{P}(f_2)$ and v_1 are initiated to ff.

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

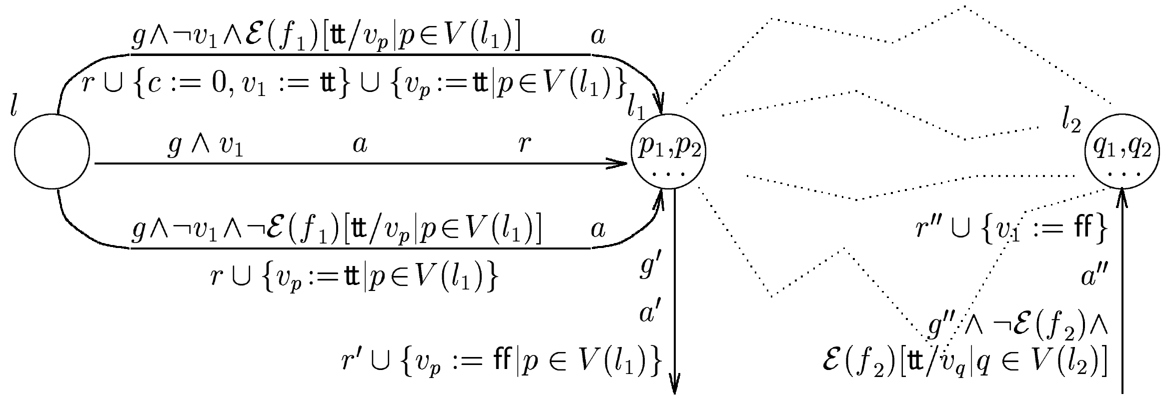
$g\wedge\neg v_1\wedge\mathcal{E}(f_1)[\mathsf{tt}/v_p|p\in V(l_1)]$
a
$r\cup\{c:=0, v_1:=\mathsf{tt}\}\cup\{v_p:=\mathsf{tt}|p\in V(l_1)\}$
l
l_1
$g\wedge v_1$
a
r
p_1,p_2
$\dots$
l_2
q_1,q_2
$\dots$
$g\wedge\neg v_1\wedge\neg\mathcal{E}(f_1)[\mathsf{tt}/v_p|p\in V(l_1)]$
a
$r\cup\{v_p:=\mathsf{tt}|p\in V(l_1)\}$
g'
a'
$r'\cup\{v_p:=\mathsf{ff}|p\in V(l_1)\}$
$r''\cup\{v_1:=\mathsf{ff}\}$
a''
$g''\wedge\neg\mathcal{E}(f_2)\wedge$
$\mathcal{E}(f_2)[\mathsf{tt}/v_q|q\in V(l_2)]$

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

Let $\mathcal{E}(f)$ denote the boolean expression by replacing all $p \in \mathcal{P}(f)$ with their corresponding boolean variable v_p. As usual, $\mathcal{E}(f)[\mathsf{tt}/v_p]$ denotes a substitution that replaces v_p with tt in $\mathcal{E}(f)$. 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 $\mathcal{E}(f)[\mathsf{tt}/v_p|p \in V(s)][\mathsf{ff}/v_p|p \notin V(s)]$.

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 $V(l_1) \cap \mathcal{P}(f_1) \neq \emptyset$:
   (a) Make two copies of each such edge.
   (b) To the original edge, add v_1 as a guard.
   (c) To the first copy, add $\neg v_1 \wedge \mathcal{E}(f_1)[\mathsf{tt}/v_p|p \in V(l_1)]$ as a guard and $c := 0, v_1 := \mathsf{tt}$ and $v_p := \mathsf{tt}$ for all $p \in V(l_1)$ as reset-operations.
   (d) To the second copy, add $\neg v_1 \wedge \neg\mathcal{E}(f_1)[\mathsf{tt}/v_p|p \in V(l_1)]$ as a guard and $v_p := \mathsf{tt}$ for all $p \in V(l_1)$ as reset-operations.
2. For all edges of A_i leaving a node l_1 such that $V(l_1) \cap \mathcal{P}(f_1) \neq \emptyset$: add $v_p := \mathsf{ff}$ for all $p \in V(l_1)$ as reset-operations.
3. For all edges of A_i entering a node l_2 such that $V(l_2) \cap \mathcal{P}(f_2) \neq \emptyset$: add $\neg\mathcal{E}(f_2) \wedge \mathcal{E}(f_2)[\mathsf{tt}/v_q|q \in V(l_2)]$ as a guard and $v_1 := \mathsf{ff}$ as a reset-operation.
4. Finally, remove $v_p := \mathsf{tt}$ and $v_p := \mathsf{ff}$ 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; its decorated version $\mathcal{M}(A_i)$ is shown in Figure 3. We take $\mathcal{M}(A_1) \,||\dots||\mathcal{M}(A_n)$ to be the decorated version of $\overline{A}$, i.e. $\mathcal{M}(\overline{A}) \equiv \mathcal{M}(A_1) \,||\dots||\, \mathcal{M}(A_n)$. For a bounded response time formula $f_1 \leadsto_{\leq T} f_2$, we now have the following fact:

$$\mathcal{M}(\overline{A}) \models \mathsf{INV}(v_1 \Rightarrow c \leq T) \;\text{ iff }\; \overline{A} \models f_1 \leadsto_{\leq T} f_2$$

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 $|P(f_1) \cup P(f_2)|$ boolean variables in $\mathcal{M}(A)$, the number of edges of $\mathcal{M}(A)$ is $3 \times |\longrightarrow_A|$ where $|\longrightarrow_A|$ is the number of edges of A (note that no extra nodes are 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 incoming 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) $\mathcal{E}(f_1)$ and $\mathcal{E}(f_2)$.

5 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.

5.1 *Functionality*

The gear controller changes gears by requesting services provided by the components in its environment. The interaction with these components is over the vehicles communication network. 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

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

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 (i.e. disengaging the clutch), and consequently the transmission, the connection between the engine and the wheels is broken. The gearbox 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 (i.e. engages) it safely bridges the speed and torque differences between the engine and the wheels. We refer to these exceptional cases as *recoverable errors.*

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

Gearbox: It is an electrically controlled gearbox 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 gearbox 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 gearbox. 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: The engine offers three modes of operation: normal torque, zero torque, and synchronous speed. The normal mode is *normal torque* 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 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 to the error states in the environment as *unrecoverable errors* since it is impossible for the gear controller alone to recover from these errors.

5.2 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 changes and error detections, are crucial to the correctness and 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 change when no unrecoverable errors occur.
   (a) A gear change should be completed within 1.5 seconds unless an unrecoverable error occurs.
   (b) A gear change, 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 be no deadlocks in the system.
   (b) When the engine is regulating torque, the clutch should be closed.
   (c) When a gear is set, the engine should be regulating torque.
3. **Functionality.** The following requirements are to ensure the desired functionality of the gear controller.
   (a) It is able to use all gears.
   (b) It uses the engine to enhance zero torque and synchronous speed over the transmission.
   (c) It uses the gearbox to set and release gears.
   (d) It is allowed to use the clutch in difficult conditions.
   (e) It does not request zero torque when changing from neutral gear.
   (f) The gear controller does not request synchronous speed when changing 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,
   (c) the gearbox is not able to set a gear in time,
   (d) the gearbox is not able to release a gear in time.

6 Formal Description of the System

To design and analyze the gear controller we model the controller and its environment in the Uppaal model [10].

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

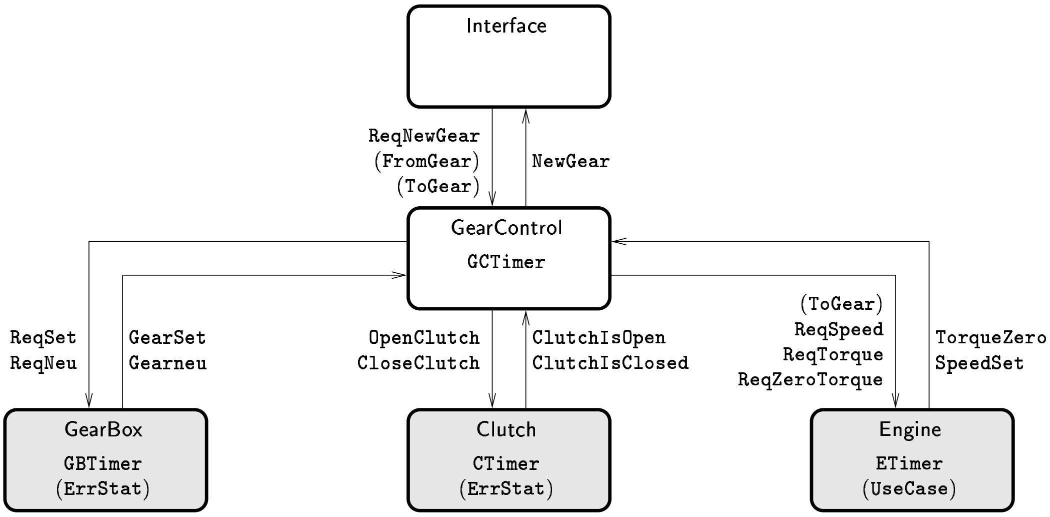

Fig. 4. A Flow-Graph of the Gearbox Model.

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 5.1). 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 eight variables. Four are timers that measure 1/1000 of seconds (ms): `GCTimer`, `GBTimer`, `CTimer` and `ETimer`. The two data variables, named `FromGear` and `ToGear`, are used as gear change requests[9] and the variables `UseCase` and `ErrStat` are assigned when errors occur in the system (see Section 7). In the following we describe the five automata of the system.

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 has been performed.

Gearbox: In automaton GearBox, shown in Figure B3, 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 gearbox fails to set or release a gear the locations named ErrorSet and ErrorNeu are entered respectively.

Clutch: The automaton Clutch is shown in Figure B1. 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 ErrorOpen and ErrorClose respectively.

Engine: The automaton Engine, shown in Figure B4, 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 requested speed is not modeled since it does not affect the design of the gear controller[10]. The engine responds 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. If the engine fails to deliver zero torque or synchronous speed in time, it enters location CluthOpen without responding to the request. Similarly, the location ErrorSpeed is entered if the engine regulates on synchronous speed in too long time.

Given the formal model of the environment, the gear controller has been designed to satisfy both the functionality requirements given in Section 5.1, and the correctness requirements in Section 5.2

Gear Controller: The GearControl automaton is shown in Figure B5. Each main loop implements a gear change by interacting with the components of the environment. The designed controller measures response times 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,

[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.

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

GNeuError or GSetError. Recoverable errors are detected in the locations CheckTorque and CheckSyncSpeed.

Interface: The automaton Interface shown in Figure B2, 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.

7 Formal Validation and Verification

In this section we formalise the informal requirements given in Section 5.2 and prove their correctness using the symbolic model-checker of Uppaal.

To enable formalisation (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 4.

Before formalising 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 4,

$$\overline{A} \models \varphi_1 \wedge \varphi_2 \text{ if and only if } \overline{A} \models \varphi_1 \text{ and } \overline{A} \models \varphi_2$$
$$\overline{A} \models \neg\varphi \text{ if and only if } \overline{A} \not\models \varphi$$

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 $\mathsf{Poss}(f)$ to denote $\neg\mathsf{INV}(\neg f)$, $f_1 \not\leadsto_{\leq T} f_2$ to denote $\neg(f_1 \leadsto_{\leq T} f_2)$, and $A.l$ to denote $\mathsf{at}(A, l)$. We are now ready to formalise the requirements.

7.1 Requirement Specification

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 (see Table 2). 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 ensures 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 12 holds if the system is able to use all gears (requirement 3a). Furthermore, for full functionality and predictability, the system is required to be deadlock-free (requirement 2a). This has been checked with an internal version of the Uppaal tool [12].

The properties 13 and 14 specify 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 to 3f). These requirements have been used to design the gear controller. They have later been validated using the simulator in Uppaal and have not been formally specified and verified.

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 the 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

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

[12] The deadlock checker will be distributed with the next release of the Uppaal tool, which will have version number 3.2.

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

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

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

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

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

$$\mathsf{GearBox.ErrorNeu} \leadsto_{\leq 300} \mathsf{GearControl.GNeuError} \tag{6}$$

$$\mathsf{INV}\ (\ \mathsf{GearControl.CCloseError} \Rightarrow \mathsf{Clutch.ErrorClose}\ ) \tag{7}$$

$$\mathsf{INV}\ (\ \mathsf{GearControl.COpenError} \Rightarrow \mathsf{Clutch.ErrorOpen}\ ) \tag{8}$$

$$\mathsf{INV}\ (\ \mathsf{GearControl.GSetError} \Rightarrow \mathsf{GearBox.ErrorIdle}\ ) \tag{9}$$

$$\mathsf{INV}\ (\ \mathsf{GearControl.GNeuError} \Rightarrow \mathsf{GearBox.ErrorNeu}\ ) \tag{10}$$

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

$$\bigwedge_{i \in \{R,N,1,\ldots,5\}} \mathsf{Poss}\ (\ \mathsf{Gear.Gear}_i) \tag{12}$$

$$\mathsf{INV}\ (\ \mathsf{Engine.Torque} \Rightarrow \mathsf{Clutch.Closed}\ ) \tag{13}$$

$$\bigwedge_{i \in \{R,1,\ldots,5\}} \mathsf{INV}((\ \mathsf{GearControl.Gear} \land \mathsf{Gear.Gear}_i) \Rightarrow \mathsf{Engine.Torque}) \tag{14}$$

Table 2. Requirement Specification

$$\mathsf{GearControl.Initiate} \leadsto_{<900} ((\ \mathtt{ErrStat} = 0 \land \mathtt{UseCase} = 0\ ) \Rightarrow \mathsf{GearControl.GearChanged}) \tag{15}$$

$$\mathsf{GearControl.Initiate} \not\leadsto_{\leq 899} ((\ \mathtt{ErrStat} = 0 \land \mathtt{UseCase} = 0\ ) \Rightarrow \mathsf{GearControl.GearChanged}) \tag{16}$$

Table 3. Time Bounds

`ToGear`. The derived time bounds are: 400 ms, 700 ms and 750.

Verification Results

We have verified totally 47 logical formulas (listed in Appendix A) of the system using UPPAAL installed on a 75 MHz Pentium PC equipped with 24 MB of primary memory. The verification of all the formulas consumed 2.99 second.

8 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 formalised in 47 logical formulas according to the informal requirements delivered by industry. The verification was performed in a few seconds on a Pentium PC running UPPAAL. Another contribution of this paper is a solution to a problem we got 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 characterise 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 includes 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.

Acknowledgment: The work has been supported by the ASTEC competence center (Advanced Software Technology) at Uppsala University. The authors want to thank Johan Bengtsson who developed a preliminary version of the UPPAAL model for the gear box system, and Hans Hansson and Mikael Strömberg for many fruitful discussions.

References

1. Luca Aceto, Augusto Bergueno, and Kim G. Larsen. Model Checking via Reachability Testing for Timed Automata. In Bernard Steffen, editor, *Proc. of the 4th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 263–280. Springer–Verlag, 1998.

2. Luca Aceto, Patricia Bouyer, Augusto Burgueo, and Kim G. Larsen. The Power of Reachability Testing for Timed Automata. In Arvind and Ramanujam, editors, *Proc. of the 18*th *Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1530 in Lecture Notes in Computer Science, pages 245–256. Springer–Verlag, 1998.
3. R. Alur and D. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.
4. Johan Bengtsson, W.O. David Griffioen, Kåre J. 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 the 8*th *Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.
5. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Proc. of the 2*nd *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, March 1996.
6. 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*, number 1066 in Lecture Notes in Computer Science, pages 208–219. Springer–Verlag, October 1995.
7. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: The Next Generation. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 56–65. IEEE Computer Society Press, December 1995.
8. Gerard Holzmann. *The Design and Validation of Computer Protocols.* Prentice Hall, 1991.
9. Henrik E. Jensen, Kim G. Larsen, and Arne Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL. In *Proc. of 2*nd *Int. Workshop on the SPIN Verification System*, pages 1–20, August 1996.
10. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
11. Thomas Stauner, Olaf Müller, and Max Fuchs. Using HyTech to Verify an Automotive Control System. In *Proc. of the 1*st *Int. Workshop on Hybrid and Real-Time Systems.* Technische Universität München, Lecture Notes in Computer Science, Springer, 1997.
12. Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.

Appendix A: The Requirement Specification

```
//////////////////////////////////////////////////////////////////
// 1996-11-20, 1997-02-20--27, and 1997-07-31 @ Uppsala University
// Paul Pettersson and Wang Yi, DoCS and Magnus Lindahl, Mecel AB.
//////////////////////////////////////////////////////////////////
//
// OVERVIEW
//
// This document is the specification file (engine.q) in UPPAALs
// q-format. It is the actual input file used to check properties
// of the gearbox controller. The system is modelled in the file
// engine.atg. To generate the ta-format of the system, run:
// 'atg2ta engine.atg engine.ta' and to verify the system, run:
// 'verifyta -sT engine.ta engine.q'.
//
// E1 to E11 are requirements on the environment of the gearbox
// controller, that should be respected by the formal model of
// the environment. R1 to R9 are requirements on the gearbox
// controller design given by Mecel AB in natural language. P1 to
// P17 formalize R1 to R9, that should be satisfied by the formal
// model of the gearbox controller.
//
// INFORMAL REQUIREMENTS ON THE ENVIRONMENT OF GEARBOX CONTROLLER
//
// E1 to E11 are requirements that the environment of the gearbox
// controller design should satisfy to guarantee the behavior of
// the whole system works properly. That is, if any of the
// requirements E1 to E11 are not satisfied by the environment
// then P1 to P17 are *not* guaranteed to hold:
//
// E1. Initially the clutch is closed.
//
// E2. To open the clutch, it takes at least 100 ms and at most
//      150 ms.
//
// E3. To close the clutch, it takes at least 100 ms and at most
//      150 ms.
//
// E4. Initially the gearbox is neutral.
//
// E5. To release the gear, it takes at least 100 ms and at most
//      200 ms.
//
// E6. To set a gear, it takes at least 100 ms and at most 300 ms.
//
// E7. The engine is always in a predefined state called "Initial"
//      when no gear is set.
//
// E8. To find zero torque in the engine, it takes at least 150 ms
//      and at most 400 ms. But at 400 ms, the engine may enter
//      an error state or find zero torque.
//
// E9. To find synchronous speed, it takes at least 50 ms and at
//      most 200 ms. But at 200 ms the engine may enter an error
//      state or find synchronous speed.
//
// E10. The engine may regulate on synchronous speed in at most
//      500 ms.
//
// E11. When in an error state, the engine will regulate on
//      synchronous speed in at least 50 ms  and at most 500 ms.
//
//
// INFORMAL REQUIREMENTS ON THE GEARBOX CONTROLLER DESIGN
//
// The Gearbox controller should satisfy the following informal
// requirements.  The properties given in parentheses are the
// formal description of the listed requirement.
//
// R1. A gear change should be performed within 1 second (P6 - P8,
//      P3).
//
// R2. When an error arises, the system will reach a predefined
//      error state marking the error (P9 - P11).
//
// R3. The system should be able to use all gears (P2 - P3).
//
// R4. There will be no deadlocked state in the system (P17).
//
// R5. When the system indicates gear neutral, the engine should
//      be in initial state (P12).
//
// R6. When the system indicates a gear, the engine should be in
//      a state performing torque regulation (P13).
//
// R7. The gearbox controller will never indicate open or closed
//      clutch when the clutch is closed or open respectively
//      (P14).
//
// R8. The gearbox controller will never indicate gear set or
//      gear neutral when the gear is not set or not idle,
//      respectively (P15).
//
```

```
// R9. When the engine is regulating on torque, the clutch is
//      closed (P16).
//
//
// FORMALIZING THE REQUIREMENTS
//
// The requirements above have been formalised using variables
// and locations of automata. The system variables listed below
// are variables used by the components of the system; the
// auxiliary variables are decorations to the system used to
// formalize the requirements only. In the system description, the
// auxiliary variables appear only in assignments (not in guards).
// This ensures that the system behavior is not changed when the
// auxiliary variables are introduced (or removed).
//
// The variables ErrStat and UseCase are used to trace errors.
// ErrStat is set when unrecoverable errors occur; UseCase is
// set when recoverable errors occur, that will be recovered by
// the gearbox controller.
//
// The systems component locations that appear in the formulae
// below can be found in the system description file
// engine.{atg|ta}.
//
// System Variables:
//
// o GCTimer  - gearbox controller timer,
// o ETimer   - engine timer,
// o GBTimer  - gearbox timer,
// o CTimer   - clutch timer,
// o FromGear - selected gear before gear change (0=N, 1=1, ...,
//              6=R),
// o ToGear   - selected gear after gear change (0=N, 1=1, ...,
//              6=R).
//
// Auxiliary Variables:
//
// o SysTimer - system timer, reset at each request for new gear
//              (in the gearbox controller),
// o ErrStat  - 0 = no errors,
//              1 = close clutch error,
//              2 = open clutch error,
//              3 = set gear failure,
//              4 = error releasing gear.
// o UseCase  - 0 = ideal scenario, no problems occurred,
//              1 = engine was not able to deliver zero torque,
//              2 = engine was not able to find synchronous speed.

///////////////////////////////////////////////////////////////////
//
// ----------------------------------------------------------------
// P1. It is possible to change gear.
// ----------------------------------------------------------------
E<> GearControl.GearChanged

// ----------------------------------------------------------------
// P2. It is possible to switch to gear nr 5 and to reverse gear
//     (i.e. R).
// ----------------------------------------------------------------
// a)
E<> Interface.Gear5
// b)
E<> Interface.GearR

// ----------------------------------------------------------------
// P3. It is possible to switch gear in 1000 ms (not very interest
//     ing).
// ----------------------------------------------------------------
E<> ( GearControl.GearChanged and ( SysTimer<=1000 ) )

// ----------------------------------------------------------------
// P4. When the gearbox is in position N, the gear is not in
//     position 1-5 or R.
// ----------------------------------------------------------------
A[] not ( GearBox.Neutral and \
          ( Interface.Gear1 or Interface.Gear2 or \
            Interface.Gear3 or Interface.Gear4 or \
            Interface.Gear5 or Interface.GearR ) )

// ----------------------------------------------------------------
// P5. The gear is never N, when the gearbox is idle (expected to
//     be neutral).
// ----------------------------------------------------------------
// a)
A[] not ( GearBox.Idle and Interface.GearN )
```

```
// b)
A[] ( Interface.GearN imply GearBox.Neutral )

// ----------------------------------------------------------------
// P6. In the case of no errors (in gear and clutch) and ideal
//     (engine) scenario,
// a)  a gear switch is guaranteed in 900 ms (including 900 ms),
// a') a gear switch is not guaranteed in strictly less than 900
//     ms,
// b)  it is impossible to switch gear in less than 150 ms,
// b') it is possible to switch gear at 150 ms,
// c)  it is impossible to switch gear in less than 400 ms if the
//     switch is not from/to gear N.
// c') it is possible to switch gear at 400 ms if the switch is
//     not from/to gear N.
// ----------------------------------------------------------------
// a)
A[] ( ( ErrStat==0 and UseCase==0 and SysTimer>=900 ) imply \
      ( GearControl.GearChanged or GearControl.Gear ) )
// a')
E<> ( ErrStat==0 and UseCase==0 and \
      SysTimer>899 and SysTimer<900 and \
      not ( GearControl.GearChanged or GearControl.Gear ) )
// b)
A[] ( ( ErrStat==0 and UseCase==0 and ( SysTimer<150 ) ) imply \
      not ( GearControl.GearChanged ) )
// (In (b) GearControl.Gear is not implied since the property is
// then satisfied by the systems initial state.)

// b')
E<> ( ErrStat==0 and UseCase==0 and GearControl.GearChanged and \
      ( SysTimer==150 ) )
// c)
A[] ( ( ErrStat==0 and UseCase==0 and FromGear>0 and \
        ToGear>0 and ( SysTimer<400 ) ) imply \
      not ( GearControl.GearChanged ) )
// c')
E<> ( ErrStat==0 and UseCase==0 and FromGear>0 and ToGear>0 and \
      GearControl.GearChanged and ( SysTimer==400 ) )

// ----------------------------------------------------------------
// P7. When no errors (in gear and clutch) occur, but engine fails
//     to deliver zero torque:
// a)  a gear switch is guaranteed after 1055 ms (not including
//     1055),
// a') it is impossible to switch gear in 1055 ms,
// b)  it is impossible to switch gear in less than 550 ms,
// b') it is possible to switch gear at 550 ms,
// c)  it is impossible to switch gear in less than 700 ms if the
//     switch is not from/to gear N.
// c') it is possible to switch gear at 700 ms if the switch is
//     not from/to gear N.
// ----------------------------------------------------------------
// a)
A[] ( ( ErrStat==0 and UseCase==1 and SysTimer>1055 ) imply \
      ( GearControl.GearChanged or GearControl.Gear ) )
// a')
E<> ( ErrStat==0 and UseCase==1 and SysTimer==1055 and \
      not ( GearControl.GearChanged or GearControl.Gear ) )
// b)
A[] ( ( ErrStat==0 and UseCase==1 and SysTimer<550 ) imply \
      not ( GearControl.GearChanged or GearControl.Gear ) )
// b')
E<> ( ErrStat==0 and UseCase==1 and GearControl.GearChanged and \
      ( SysTimer==550 ) )
// c)
A[] ( ( ErrStat==0 and UseCase==1 and FromGear>0 and \
        ToGear>0 and SysTimer<700 ) imply \
      not ( GearControl.GearChanged and GearControl.Gear ) )
// c')
E<> ( ErrStat==0 and UseCase==1 and FromGear>0 and ToGear>0 and \
      GearControl.GearChanged and \
      ( SysTimer==700 ) )

// ----------------------------------------------------------------
// P8. When no errors occur, but engine fails to find synchronous
//     speed:
// a)  a gear switch is guaranteed in 1205 ms (including 1205),
// a') a gear switch is not guaranteed at less than 1205 ms,
// b)  it is impossible to switch gear in less than 450 ms,
// b') it is possible to switch gear at 450 ms,
// c)  it is impossible to switch gear in less than 750 ms if the
//     switch is not from/to gear N.
// c') it is possible to switch gear at 750 ms if the switch is
//     not from/to gear N.
// -----------------------------------------------------------------
// a)
A[] ( ( ErrStat==0 and UseCase==2 and SysTimer>=1205 ) imply \
      ( GearControl.GearChanged or GearControl.Gear ) )
// a')
E<> ( ErrStat==0 and UseCase==2 and SysTimer>1204 and \
      SysTimer<1205 and \
      not ( GearControl.GearChanged or GearControl.Gear ) )
// b)
A[] ( ( UseCase==2 and ( SysTimer<450 ) ) imply \
      not ( GearControl.GearChanged or GearControl.Gear ) )
// b')
E<> ( UseCase==2 and GearControl.GearChanged and \
      ( SysTimer==450 ) )
// c)
A[] ( ( ErrStat==0 and UseCase==2 and FromGear>0 and \
        ToGear>0 and SysTimer<750 ) imply \
      not ( GearControl.GearChanged and GearControl.Gear ) )
// c')
E<> ( ErrStat==0 and UseCase==2 and FromGear>0 and ToGear>0 and \
      GearControl.GearChanged and \
      ( SysTimer==750 ) )

// -----------------------------------------------------------------
// P9. Clutch Errors.
// a) If the clutch is not closed properly (i.e. a timeout
//    occurs) the gearbox controller will enter the location
//    CCloseError within 200 ms.
// b) When the gearbox controller enters location CCloseError,
//    there is always a problem in the clutch with closing the
//    clutch.
// -----------------------------------------------------------------
// a)
A[] ( ( Clutch.ErrorClose and ( GCTimer>200 ) ) imply \
      GearControl.CCloseError )
// b)
A[] ( GearControl.CCloseError imply Clutch.ErrorClose )

// -----------------------------------------------------------------
// P9. Clutch Errors (cont.)
// c) If the clutch is not opened properly (i.e. a timeout occurs)
//    the gearbox controller will enter the location COpenError
//    within 200 ms.
// d) When the gearbox controller enters location COpenError,
//    there is always a problem in the clutch with opening the
//    clutch.
// -----------------------------------------------------------------
// c)
A[] ( ( Clutch.ErrorOpen and ( GCTimer>200 ) ) imply \
      GearControl.COpenError )
// d)
A[] ( ( GearControl.COpenError ) imply Clutch.ErrorOpen )

// -----------------------------------------------------------------
// P10. Gearbox Errors.
// a) If the gearbox can not set a requested gear (i.e a timeout
//    occurs) the gearbox controller will enter the location
//    GSetError within 350 ms.
// b) When the gearbox controller enters location GSetError, there
//    is always a problem in the gearbox with setting the gear.
// -----------------------------------------------------------------
// a)
A[] ( ( GearBox.ErrorIdle and ( GCTimer>350 ) ) imply \
      GearControl.GSetError )
// b)
A[] ( ( GearControl.GSetError ) imply GearBox.ErrorIdle )

// -----------------------------------------------------------------
// P10. Gearbox Errors (cont).
// c) If the gearbox can not switch to neutral gear (i.e. a
//    timeout occurs) the gearbox controller will enter the
//    location GNeuError within 300 ms.
// d) When the gearbox controller enters location GNeuError there
//    is always a problem in the gearbox with switching to neutral
//    gear.
// -----------------------------------------------------------------
// c)
A[] ( ( GearBox.ErrorNeu and ( GCTimer>300 ) ) imply \
      GearControl.GNeuError )
// d)
A[] ( ( GearControl.GNeuError ) imply GearBox.ErrorNeu )

// -----------------------------------------------------------------
// P11. If no errors occur in the engine, it is guaranteed to find
//      synchronous speed.
```

```
// -----------------------------------------------------------------
A[] not ( ErrStat==0 and Engine.ErrorSpeed )

// -----------------------------------------------------------------
// P12. When the gear is N, the engine is in initial or on its way
//      to initial (i.e. ToGear==0 and engine in zero).
// -----------------------------------------------------------------
A[] ( Interface.GearN imply \
      ( ( ToGear==0 and Engine.Zero ) or Engine.Initial ) )

// -----------------------------------------------------------------
// P13. When the gear controller has a gear set, torque regulation
//      is always indicated in the engine.
// -----------------------------------------------------------------
// a)
A[] ( ( GearControl.Gear and Interface.GearR ) imply \
      Engine.Torque )
// b)
A[] ( ( GearControl.Gear and Interface.Gear1 ) imply \
      Engine.Torque )
// c)
A[] ( ( GearControl.Gear and Interface.Gear2 ) imply \
      Engine.Torque )
// d)
A[] ( ( GearControl.Gear and Interface.Gear3 ) imply \
      Engine.Torque )
// e)
A[] ( ( GearControl.Gear and Interface.Gear4 ) imply \
      Engine.Torque )
// f)
A[] ( ( GearControl.Gear and Interface.Gear5 ) imply \
      Engine.Torque )

// -----------------------------------------------------------------
// P14. a) If clutch is open, the gearbox controller is in one of
//         the predefined locations.
//      b) If clutch is closed, the gearbox controller is in one
//         of the predefined locations.
// -----------------------------------------------------------------
// a)
A[] ( Clutch.Open imply \
      ( GearControl.ClutchOpen or GearControl.ClutchOpen2 or \
        GearControl.CheckGearSet2 or GearControl.ReqSetGear2 or \
        GearControl.GNeuError or \
        GearControl.ClutchClose or \
        GearControl.CheckClutchClosed or \
        GearControl.CheckClutchClosed2 or \
        GearControl.CCloseError or \
        GearControl.GSetError or GearControl.CheckGearNeu2 ) )
// b)
A[] ( Clutch.Closed imply \
      ( GearControl.ReqTorqueC or GearControl.GearChanged or \
        GearControl.Gear or GearControl.Initiate or \
        GearControl.CheckTorque or GearControl.ReqNeuGear or \
        GearControl.CheckGearNeu or GearControl.GNeuError or \
        GearControl.ReqSyncSpeed or \
        GearControl.CheckSyncSpeed or GearControl.ReqSetGear or \
        GearControl.CheckGearSet1 or GearControl.GSetError ) )

// -----------------------------------------------------------------
// P15. a) If gear is set, the gearbox controller is in
//         one of the predefined locations.
//      b) If gear is neutral, the gearbox controller is in
//         one of the predefined locations.
// -----------------------------------------------------------------
// a)
A[] ( GearBox.Idle imply \
      ( GearControl.ClutchClose or \
        GearControl.CheckClutchClosed or \
        GearControl.CCloseError or \
        GearControl.ReqTorqueC or GearControl.GearChanged or \
        GearControl.Gear or GearControl.Initiate or \
        GearControl.CheckTorque or GearControl.ReqNeuGear or \
        GearControl.CheckClutch2 or GearControl.COpenError or \
        GearControl.ClutchOpen2 ) )
// b)
A[] ( GearBox.Neutral imply \
      ( GearControl.ReqSetGear or \
        GearControl.CheckClutchClosed2 or \
        GearControl.CCloseError or GearControl.ReqTorqueC or \
        GearControl.GearChanged or GearControl.Gear or \
        GearControl.Initiate or GearControl.ReqSyncSpeed or \
        GearControl.CheckSyncSpeed or GearControl.ReqSetGear or \
        GearControl.CheckClutch or GearControl.COpenError or \
        GearControl.ClutchOpen or GearControl.ReqSetGear2 ) )
```

```
// ---------------------------------------------------------------
// P16. If engine regulates on torque, then the clutch is closed.
// ---------------------------------------------------------------
A[] ( Engine.Torque imply Clutch.Closed )

// ---------------------------------------------------------------
// P17. As all states will satisfy "1 > 0", model-checking this
//      formula will generate the whole state-space of the system,
//      and the answer will be that the property is satisfied.
//      UPPAAL is designed to report all the deadlocked states
//      during state-space exploration. So if no deadlock is
//      reported before the final answer is given, the system is
//      deadlock-free.
// ---------------------------------------------------------------
A[] ( 1 > 0 )

/////////////////////////// - end - //////////////////////////////
```

Appendix B: The System Description

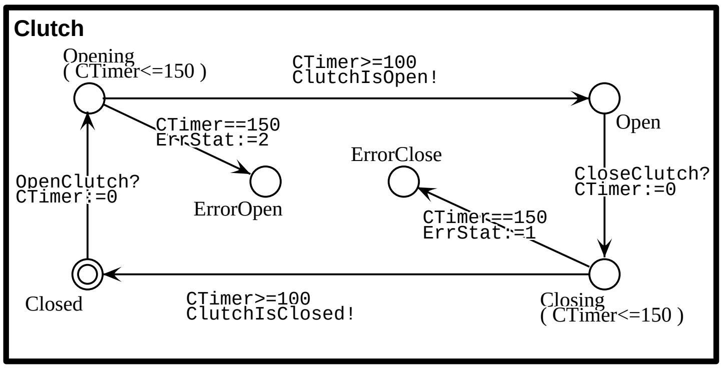

Fig. B1. The Clutch Automaton.

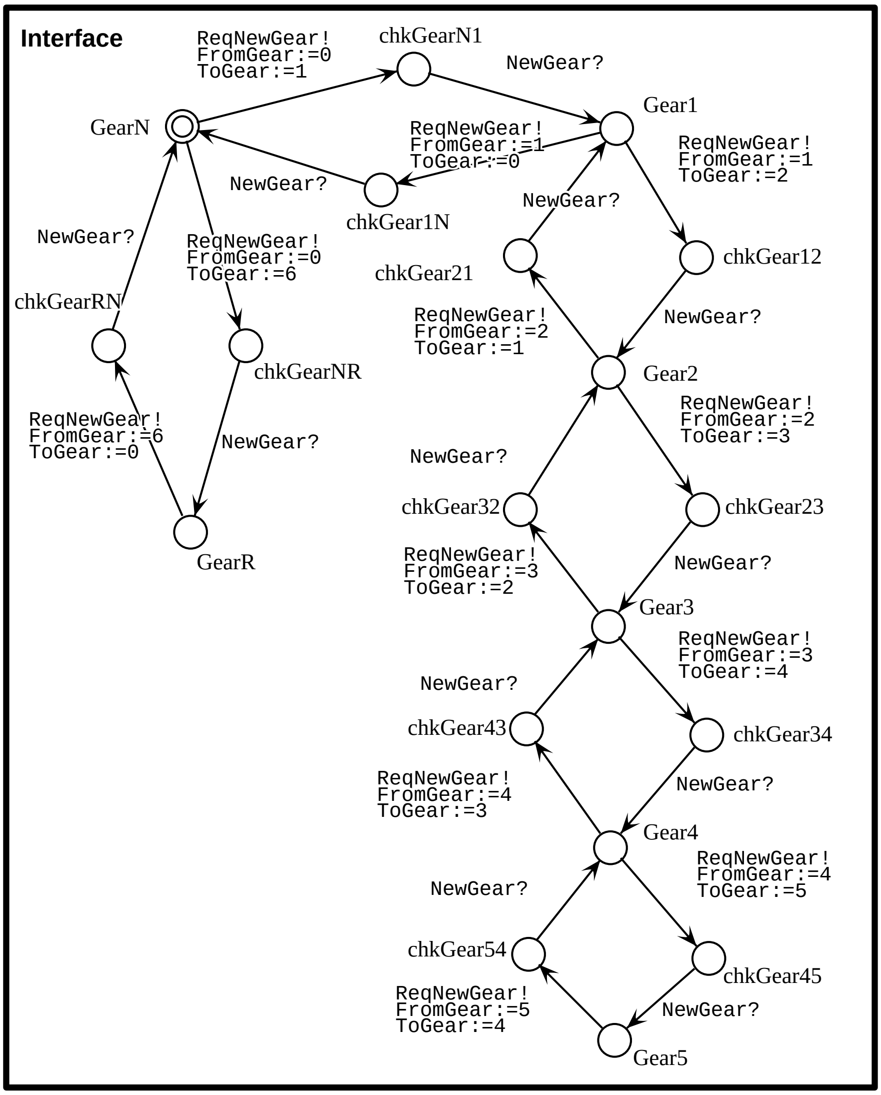

Fig. B2. The Interface Automaton.

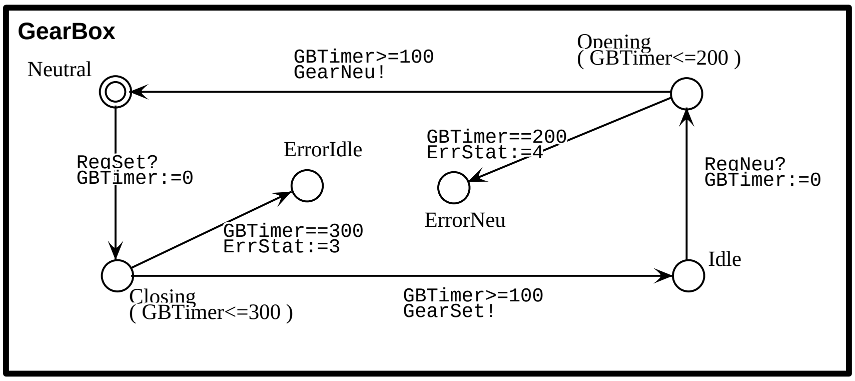

Fig. B3. The Gear-Box Automaton.

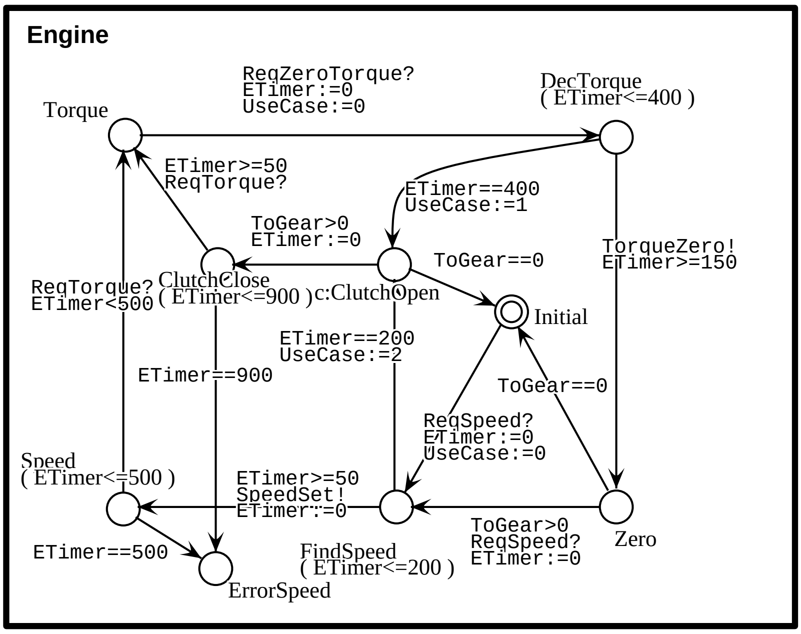

Fig. B4. The Engine Automaton.

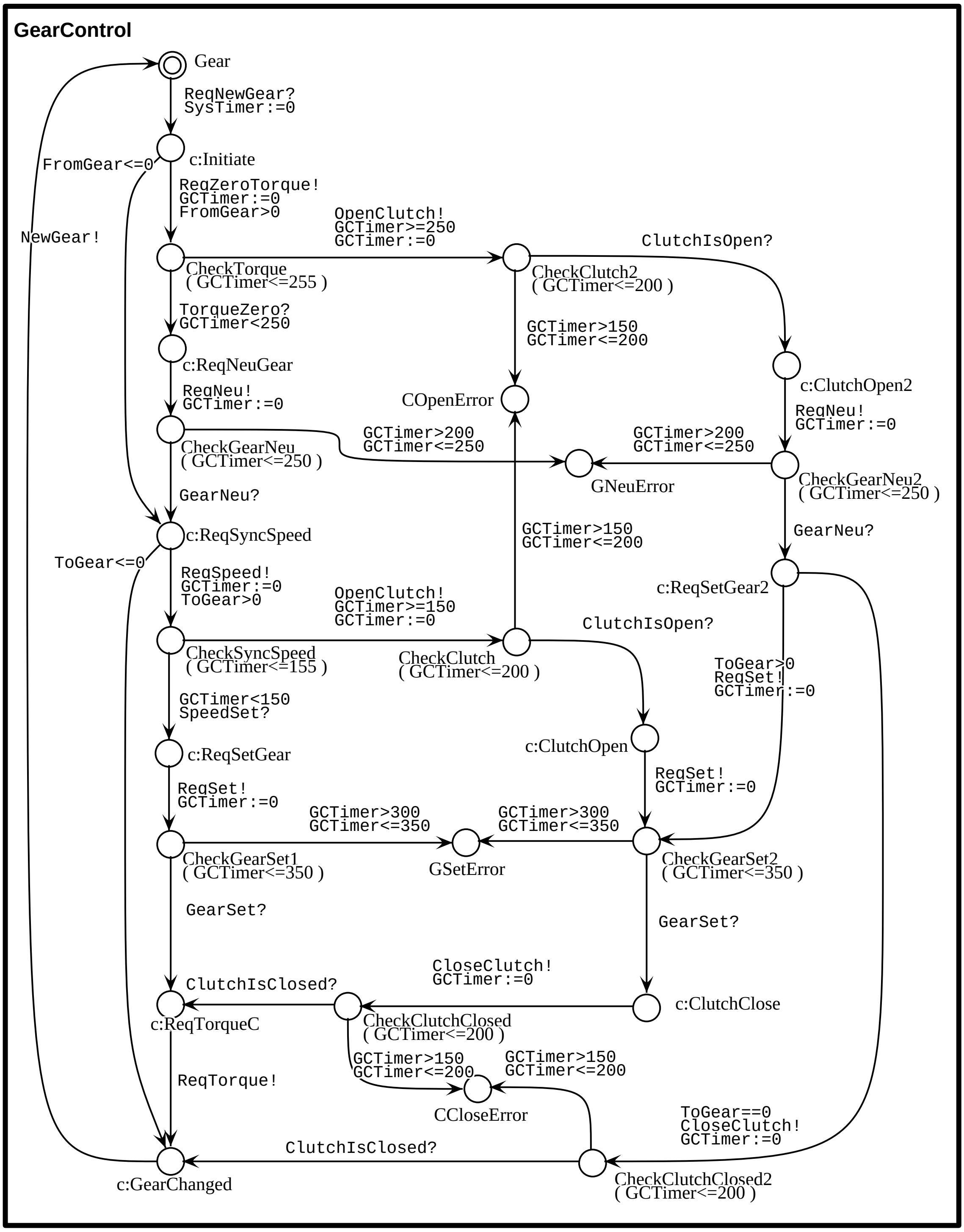

Fig. B5. The Gear Box Controller Automaton.