2013 International Symposium on Theoretical Aspects of Software Engineering

Modeling and Verification of AUTOSAR OS and EMS Application

Yunhui Peng[1], Yanhong Huang[1], Ting Su[1], Jian Guo[1,2,*]
[1]Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, P. R. China,
[2]Key Lab of Information Network Security, Ministry of Public Security, Shanghai, P. R. China
email: {yhpeng, tsu}@ecnu.cn
{yhhuang, jguo}@sei.ecnu.edu.cn

***Abstract*—AUTOSAR, derived from OSEK/VDX, is the most popular industrial standard in the automotive electric development. It is challenging to manually verify or validate the correctness and safety of AUTOSAR Operating System (OS) as well as mission-critical or real-time applications built on it. In this paper, we adopt timed CSP to describe and reason about the *Schedule Table*, a new task scheduling mechanism in AUTOSAR. We also employ timed CSP to model AUTOSAR OS and a real-time application, i.e., the Engine Management System (EMS), based on the *Schedule Table* mechanism, and verify some safety properties. In addition, we simulate and verify our models in Process Analysis Toolkit (PAT). The result indicates that both AUTOSAR OS and EMS application conform to the specifications and requirements.**

I. Introduction

AUTOSAR stands for AUTomotive Open System ARchitecture [2], and has become the industrial standard in the automotive electric software development. AUTOSAR is developed from OSEK/VDK [3], another open-ended architecture for automotive system. Recently, more and more automotive companies tend to take AUTOSAR as their primary standard and concentrate their interests on the improvement and research about it rather than OSEK/VDK. The core parts of OSEK/VDK including task, resource and scheduling management [4] have been integrated into AUTOSAR, which has also incorporated other new mechanisms such as schedule table, stack monitor, memory protection, etc, to improve its flexibility, safety and portability [1].

Moreover, many vendors offer their own automotive applications based on this standard. These applications include engine management system [5], automated transmission system [7], steer-by-wire system [6], etc. Many of them are real-time and safety-critical applications, which involve complicate execution logic and strict time constraint. From the perspective of benefit, AUTOSAR facilitates Original Equipment Manufacturers (OEM), suppliers, tool developers and new market entrants. In essence, AUTOSAR supports a higher reuse of software components and gives convenience to develop more mature automotive applications by the integration of components. It brings higher flexibility but without compromising high software quality.

In spite of the benefits of AUTOSAR mentioned above, we still have to validate the correctness and safety of the kernel of this operating system and some critical automotive applications built on it. However, manual checking and analysis have limitations and may miss intricate bugs. So many people have resorted to formal methods to conduct the verification work and analyzed advantages and limitations of this promising technology [8]. This kind of machine-checked or automated model checking methodology facilitate verification tasks to a great extent.

Waszniowski et al. employed model checking theory based on timed automata to build the non-preemptive model of tasks and resources of a real-time OSEK/VDX Operating System (OS), and they also treated the application related control system as separated automata to verify safety properties and bounded liveness properties of the whole system [9]. Huang et al. set out from the code of one real OSEK/VDX OS and use Communication Sequential Process (CSP) [11] to model the tasks, resources and scheduling. They also simulated system service interfaces and finally reasoned about task scheduling scheme and resource management [14]. Bertrand et al. compared the timing protection mechanism of AUTOSAR with that of other real time operating systems, and pointed out some pitfalls in this mechanisms of AUTOSAR [15].

In this paper, we focus on adopting timed CSP [10] calculus to model and verify AUTOSAR OS and a complicate real-time application, Engine Management System (EMS), which is based on AUTOSAR OS. We firstly build the formal model of the OS and focus on the scheduling policies to model AUTOSAR OS. We use a formal model to represent task management, synchronization (resources management and event mechanism), and schedule table scheduling. On the other hand, we build the formal model of EMS control part and the execution part. Despite of inheriting those API of AUTOSAR OS, we use processes in PAT to model the pipeline of four strokes. Next, we investigate the attributes of tasks, i.e., types, priorities, states, as well as the schedule policy in AUTOSAR OS standard and the application requirements. Those properties are about tasks, resources, and schedule tables, i.e., mutual exclusion between schedule tables, excluded priority inversion, etc. The EMS application also has its requirements, i.e., fixed started order of cylinders, mutual exclusion between cylinders.

The remainder of this paper is organized as follows. Section II gives a general impression of AUTOSAR OS, EMS application and timed CSP. In section III, we explain the overview of our approach. Section IV and section V show the formal models of AUTOSAR OS and EMS. Section VI abstracts some important properties from AUTOSAR OS requirements and EMS application requirements. The formal models and verification are implemented in PAT [16, 17] in section VII. Section VIII concludes the paper and discusses some future improvements of our work.

* Corresponding Author

978-0-7695-5053-4/13 $26.00 © 2013 IEEE
DOI 10.1109/TASE.2013.13

II. BACKGROUND

In this section, we give some general information about AUTOSAR Operating System (OS) and its schedule table mechanism. Next, we introduce the Engine Management System (EMS) based on AUTOSAR OS follow by some background on timed CSP.

A. AUTOSAR Operating System

AUTOSAR [1] has been gradually accepted as an automotive system architecture standard in automotive industry and applied in different kinds of Electronic Control Units (ECU) of modern vehicles.

The AUTOSAR OS inherits many core mechanisms of OSEK/VDX standard. For instance, it provides task management, two means of synchronization on tasks, i.e., resource management and event control. For task management, it provides several interfaces or services to manipulate tasks and achieve the expected purposes in some applications. Before the execution of the OS, the user has to configure the operating system such as setting priorities of tasks and running parameters of the target hardware. In brief, it gives a basis for application to run independently and enable simultaneous execution of several processes on one processor.

In the AUTOSAR OS, applications are composed of tasks, which are divided into two types, namely *basic* task and *extended* task. A basic task has three states: *ready*, *running* and *suspended*, while a extended task has an extra *waiting* state. A task transits from *ready* to *running* when it gets the opportunity to execute. It may enter into *suspended* when it terminates and get back to *ready* after activation. In addition, an extended task always remains *waiting* until some related events are trigged.

The OSEK/VDX OS employs alarm mechanism to achieve tasks activation and event setting by means of using OS counters and a series of predefined alarms. But in the AUTOSAR OS, it makes use of schedule table to implement synchronization. A schedule table encapsulates a set of statically defined *expiry point*s. Each expiry point is composed of the following two components:

- **Action**: one or more actions that must occur when it is processed. An action is the activation of a task or the setting of an event.
- **Offset**: an offset in ticks from the start of the schedule table.

Generally, a schedule table has its own fixed duration. Its expiry points will be sequentially processed by an OS counter. Actually, schedule table is a time-based scheduling strategy. If a schedule table repeats after its final expiry point is processed, it is a *repeating* schedule table. Otherwise, it is a *one-shot* schedule table if it stops at the end after one execution. In our paper, the EMS is an application based on repeating schedule table.

Example. Fig.1 shows an example of a schedule table. The schedule table has three expiry points with the duration of 39 ticks. The initial expiry point locates at 8 ticks from the beginning as indicated by its defined offset. Expiry point 1 contains an activation of *TaskA* and a event setting of *TaskB*. The other two expiry points, i.e., expiry point 2 and expiry point 3, locate at 18 ticks and 31 ticks respectively. Note that not every expiry point defines *Task Activation* or *Event Setting* but it must define an *offset*.

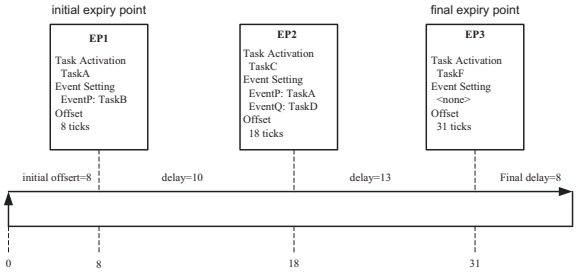

Fig.1 An Example of Schedule Table

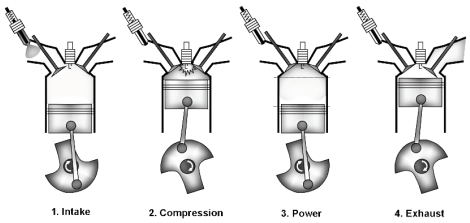

Fig.2 The Four-stroke of A Single Cylinder

B. Engine Management System

EMS is an automotive engine control application. This application periodically collects environmental parameters affected by the driver and controls the combustion processes in the cylinders of an automobile to achieve the desired driving speed. The environmental parameters include the rotation speed of the engine, the temperature and pressure of intake air-flow, the temperature of coolant. With these parameters, the application figures out the amount of fuel to be injected and the desired driving torque. Automobiles are usually equipped with four-cylinder or eight-cylinder engines, and these cylinders are air-flow controlled and intake manifold fuel injected. In this paper, we illustrate this application with the four-cylinders engine. The combustion process including four strokes, i.e., *intake*, *compression*, *power*, and *exhaust*. Fig.2 shows the combustion process in a single cylinder following these four strokes sequentially.

During the *intake* stroke, fuel is injected into the intake manifold by an injector where it is mixed with air from the inlet. The air/fuel mixture in the combustion chamber is compressed until the piston reaches the top dead center during the *compression* stroke. In *power*, the air/fuel mixture is burned and the chemical energy of the air/fuel mixture is transformed into heat and mechanical energy. At last, the outlet valve opens and the exhaust gases stream out of the combustion chamber. These four steps periodically repeat during the combustion process.

C. Timed CSP

Timed CSP is an extension of CSP by adding some time operators, which is proposed by Reed and Roscoe [12], and later modified by Davies and Schneider [13]. The syntax of timed CSP which we will use to model the AUTOSAR OS and its application EMS is given as follows.

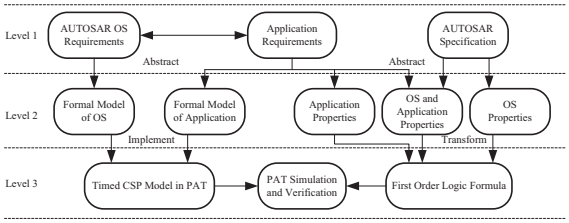

Fig.3 The Overview of Our Research

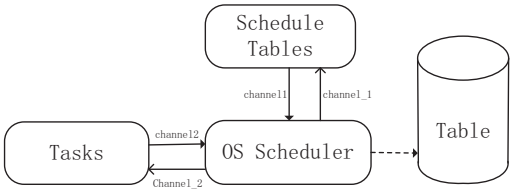

Fig.4 The Architecture of AUTOSAR OS

$$P, Q \ ::= \ Stop \mid Skip \mid Wait\ t;\ P \mid a \rightarrow P \mid P; Q \mid P \,\square\, Q$$
$$c!a \rightarrow P \mid c?x \rightarrow P \mid P|||Q \mid P\|Q \mid \mu X \bullet F(X)$$

Besides those operator in CSP, there are new operators, in our paper we use the new operator $Wait\ t;\ P$, it is a delay form of $Skip$ which also does nothing but executes P after t time units.

III. Overview

In this section, we present the overview of our approach to formalizing AUTOSAR OS and a relative application EMS. Our goal is to model the behaviors of period tasks dispatched by "schedule table" introduced in AUTOSAR OS, and verify whether the formal model satisfies the properties abstracted from AUTOSAR OS specification and EMS requirements. In our approach, a key element is to describe the real time in AUTOSAR OS by timed CSP. Fig.3 shows three phases of our approach.

Level 1. As mentioned before, we fucus on the schedule table mechanism, which is a new mechanism introduced in AUTOSAR OS. Our purpose is to use a formal model to represent task management, synchronization (resources management and event mechanism), schedule table of a AUTOSAR OS implementation provided by the company *iSoft* (iSoft Infrastructure Software CO.). Based on this model, we also represent an application EMS applied in AUTOSAR OS. We verify whether those implementations satisfy the AUTOSAR OS specification and the EMS requirements.

Level 2. In this level, the left part shows the formal models abstracted from the implementations. Firstly, we use a formal model to describe AUTOSAR OS implementation. We investigate the attributes of tasks, i.e., types, priorities, states, as well as the schedule policy. We also investigate the allocation of resources and the event control. As for the schedule table, we investigate its novelty: activate a task or set an event periodically. Secondly, we use a formal model to describe EMS application. We fucus on both the control part and the execution part of this application. Meanwhile, the right part shows the properties got from the AUTOSAR OS specification and the EMS requirements. AUTOSAR OS specification is an industry standard used for motor vehicles. It not only presents a description which relates to a specific implementation, but also lists the requirements the implementation should satisfy. We abstract some properties about tasks, resources, and schedule tables, i.e., excluded priority inversion, deadlock freedom, mutual exclusion between schedule tables, and etc. The EMS application also has its requirements, i.e., fixed started order of cylinders.

Level 3. In this level, we adopt timed CSP to implement the formal model in level 2. Timed CSP provides a timed stability model, which assumes that all events recorded by processes within the system relate to conceptual global clock. The assumption is in accordance with the characteristics of AUTOSAR OS and EMS applications. Moreover, the timed CSP model can be translated in PAT. It has a timed CSP-style process algebra combines high-level modeling operators and low-level constructs, which can help describing the details of the model. At the same time, all the properties will be rewritten by first-order logic which is also supported in PAT.

IV. Modeling AUTOSAR Operating System

Schedule table mechanism plays important role of activating tasks and setting related events at a series of predefined time points (measured by *offset*). In the context of real-time and concurrent applications, this mechanism provides the precise task scheduling and synchronization. It also agrees with the *static* property of the AUTOSAR OS, i.e., all tasks are pre-configured with proper time constraints before the system starts. From the perspective of this view, the schedule table's configuration has a direct impact on the tasks execution. So it is important to ensure the correctness of the schedule table. In fact, one application may contain one or more schedule tables. Actually, it's impossible to conduct manual checking and human inspection to decide wether an application runs as expected in various environments. In [14], Huang et al. have adopted CSP to formalize task management, resource allocation and event control of the OSEK/VDX OS from the code level, but they did not take time properties into consideration during the system execution. As we mentioned before, AUTOSAR is based on OSEK/VDX has added some new mechanisms, i.e., the "schedule table" discussed in this paper. So, in order to deal with time properties of real automotive applications, we adopt timed CSP to model the AUTOSAR schedule tables, tasks and OS scheduling to verify whether they satisfy the specification and requirements in this paper.

Fig.4 shows the whole architecture of AUTOSAR OS. It shows the relationship among tasks, schedule tables, OS scheduler and data table. Tasks and schedule tables can send different requests to the OS scheduler via corresponding channels. When the OS scheduler receives the requests, it disposes them immediately and sends the result to the requested task or schedule table. The OS scheduler also can access and modify the table which records the information of the tasks, resources and schedule tables. The details are listed in Table 1.

The AUTOSAR OS is modeled as the parallel composition of tasks, schedule tables and OS scheduler as below:

$$System =_{df} (|||_{i\in 0\ldots(n-1)} Task_i) \ \|(|||_{j\in 0\ldots(m-1)} SchTab_j) \ \| \ OSschedule$$

We use the processes names: $Task_i$, $SchTab_j$ and

Table 1. Records of Tasks, Resources and Schedule Tables

task	task identifier, task initial priority, task running priority, tasks number, task activated times, task ready queue, task occupied resources number, max priority task identifier, ready tasks number
resource	resource identifier, resource priority, resource state, occupied resources number
schedule table	schedule table identifier, schedule tables number, schedule table state, schedule table ready queue, minimal offset schedule table identifier, ready schedule tables number, schedule table initial offset

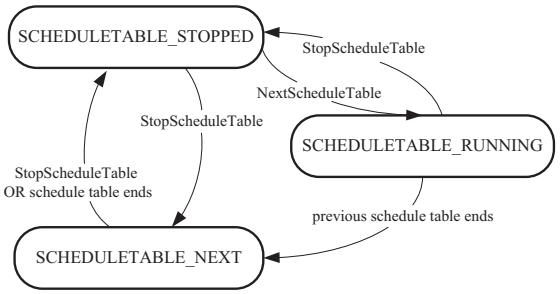

Fig.5 The State Transition of Schedule Table

$OSschedule$ to represent tasks, schedule tables and OS scheduler. The detailed communication between them will be explained in the subsections. We divide the modeling of the AUTOSAR OS into three subsections: task management and synchronisation, schedule table and the OS schedule. The OS schedule involves the accessing and modification of the tables.

A. Task

In AUTOSAR OS, one task can not only provide the same services just as in OSEK/VDX OS, but also can start a schedule table (*StartScheduleTable*), stop a schedule table (*StopScheduleTable*) or start any schedule table before one schedule table stopped (*NextScheduleTable*). There are two modes to start the schedule table, in this paper, we suppose that the schedule table is started in the absolute mode, and accordingly, it takes the implicit synchronization way. All the new operations about schedule table cause the system reschedule the schedule table, which we use the process $STScheduler$ which is presented in subsection C, to deal with the schedule table's rescheduling. The task uses channel SS to tell the OS schedule that it requests to reschedule the schedule table.

Overall, one schedule table has three states: *running*, *next* and *stopped*. Fig.5 shows the state transition of a schedule table. Service $StartScheduleTable$ can invoke a stopped schedule table into the next state and make no effect when the started task is not in the stopped state. Service $StopScheduleTable$ can force the running schedule table into stopped schedule table. $NextScheduleTable$ causes the termination of the running schedule table, and starts one schedule table. The three services change the states of one or more schedule table in the table.

We define the process $Task_i$ as follows where i belongs to set $0...(n-1)$ and i stands for the task identifier while the first m tasks are extended tasks:

$$Task_i =_{df} (\Box_{tid\in 0...(n-1)} AT!i.tid \rightarrow (response?i.OK \rightarrow Skip \Box response?i.Pause \rightarrow response?i.OK \rightarrow Skip); Task_i)$$
...
$$\Box(\Box_{stid\in 0...(s-1)} StartST!i.stid \rightarrow (response?i.OK \rightarrow Skip \Box response?i.Pause \rightarrow response?i.OK \rightarrow Skip); Task_i)$$
$$\Box(\Box_{stid_0\in 0...(s-1)\wedge stid_1\in 0...(sn-1)} NextST!stid_0.stid_1 \rightarrow (response?i.OK \rightarrow Skip \Box response?Pause \rightarrow response?i.OK \rightarrow Skip); Task_i)$$
$$\Box(\Box_{stid\in 0...(s-1)} StopST!i.stid \rightarrow (response?i.OK \rightarrow Skip \Box response?i.Pause \rightarrow response?i.OK \rightarrow Skip); Task_i)$$
$$\Box(\Box_{stid\in 0...(s-1)} SS!i.stid \rightarrow (response?i.OK \rightarrow Skip \Box response?i.Pause \rightarrow response?i.OK \rightarrow Skip); Task_i);$$

In the process $Task_i$, the ignored parts are the nine services which have already been modeled in OSEK/VDX OS: TerminateTask, ChainTask, Scheduler, SetEvent, WaitEvent, ClearEvent, GetResource, ReleaseResource. The functionalities of them and the scheduling mechanism of the nine services are just the same as those in OSEK/VDX OS, the detailed definitions of them can be found in [14]. On the whole, when the task sends messages in different channels, it indicates that the task calls the corresponding services. Now we consider the new service $StartScheduleTable$ in AUTOSAR OS. The task sends its own identifier i and the started schedule table identifier $stid$ to $OSschedule$ in $StartST$ channel through distinct channels. The task can start any schedule table. So $stid$ can be chosen from 0 to $s-1$, where s is the total schedule table number. If the task receives the OK signal, the running task carries on its next service. If the reply is $Pause$, the task should save its content and wait for another message OK to continue executing again.

B. Schedule Table

In the following, we firstly give more details about AUTOSAR schedule table. Then we propose our method on how to model the schedule table with timed CSP. A schedule table is composed of several expiry points, which include two components, i.e., offset and action (task activations and event settings). The offset represents the time interval between two expiry points expressed by *tick*s. The schedule table is actually driven by an OS counter. In the sequel, we use ticks to measure the actual system execution time for convenience. Then, the constraints on time intervals can be precisely expressed by the primitives of timed CSP. For instance, the fact that the schedule table starts after *t* ticks is formalized as *Wait[t]*. During the system execution, the schedule table sequentially iterate on its expiry points to conduct their own actions. System services like *SActivateTask* and *SSetEvent* will be called to conduct corresponding behaviors. These system calls involve the synchronization between tasks.

Take a single repeating schedule table for example, it can be formalized as timed CSP processes listed as follows. Here, we omit the internal execution time of system calls and suppose all the events take no time. These events can be easily realized by defining all the events to be urgent events in timed CSP.

$$SchTab_j =_{df} EP_1; EP_2; ...EP_k; Wait[finaldelay]; SchTab_j;$$
$$EP_k =_{df} Wait[delay]; (\Box_{tid\in 0...(n-1)} SAT!j.tid \rightarrow Skip) \Box (\Box_{tid\in 0...(n-1)} SSE!tid.eid \rightarrow Skip) \rightarrow$$

$(Sresponse?j.OK_S \rightarrow Skip \Box Sresponse?j.Pause_S \rightarrow$
$Sresponse?j.OK_S \rightarrow Skip);$

In the process $SchTab_j$, j stands for the schedule table identifier, during every execution of each expiry point, it waits for the corresponding delay, whose value is the difference of offset of the two adjacent expiry points. Then it can activate one task or more, or set an event to any extended task. The following expiry points follow the same procedure. A schedule table may have a final delay which we can see just as Fig.1 shows, it may also delay for a while before the next repetition. As illustrated above, the schedule table introduces a new mechanism to active a task or set an event, which is different from [14].

We use another different channel SAT to denote activating a task in schedule table instead of AT in task. If we use the same channel AT, the OS Scheduler can not distinguish whether it is the task or the schedule table call for activating a task just by the identifier, because the identifier do not contain any additional information. In addition, we even use different replies (OK_S and $Pause_S$) to distinguish the response from OS scheduler for the same request in two different ways. It's the same reason that we use SSE in schedule table while use SE in task to set a event to any extended task.

C. OS Schedule

The process $OSschedule$ takes charge of managing tasks, schedule tables and resources which has four duties: handling the requests from task, accessing the table and modifying the records according to different demands, rescheduling and scheduling the schedule tables. SO $OSschedule$ consists of four paralleling processes, i.e., OS, $MdfTable$, $CallScheduler$ and $STScheduler$ which deal with the above duties respectively. The process is shown as below:

$$OSschedule =_{df} OS \parallel MdfTable \parallel CallScheduler \parallel STScheduler$$

The process OS communicates with task or schedule table directly, which deals with kinds of requests from task or schedule table. It contains 14 sub-processes which have the same names as corresponding services, ActivateTask, TerminateTask, and etc. 9 services are the same as those in OSEK/VDX OS, 5 services are added, which are StartScheduleTable, StopScheduleTable, NextScheduleTable, SActivateTask, and SSetEvent. We omit the 9 services in OSEK/VDX and list all the new services, OS is defined as follows:

$$OS =_{df} ActivateTask \parallel ... \parallel StartScheduleTable \parallel SActivateTask \parallel StopScheduleTable \parallel NextScheduleTable \parallel SSetEvent ;$$

We take the $StartScheduleTable$ service for example, it is defined as follows:

$StartScheduleTable =_{df}$
$StartST?i.stid \rightarrow mdfTable \rightarrow ScallScheduler \rightarrow$
$(response!i.OK \rightarrow Skip \Box response!i.Pause \rightarrow Skip);$

This process deals with the request that task start a schedule table. The OS receives the calling task identifier i and the started schedule table identifier $stid$ from task in the $StartST$ channel. The action $mdfTable$ modifies the records based on the two identifiers. The action $ScallScheduler$ chooses which schedule table is the next running schedule table and modifies the related schedule tables states. If the calling task is still the next running task, the process sends OK via the $response$ channel, or sends $Pause$ to task to suspend it.

Considering another process $SActivateTask$, which is defined as:

$SActivateTask =_{df} SAT?j.tid \rightarrow mdfTable \rightarrow callScheduler$
$\rightarrow (Sresponse!j.OK_S \rightarrow Skip \Box Sresponse!j.Pause_S \rightarrow Skip);$

This process is similar to the process $ActivateTask$ and deals with the request that a schedule table activate any task. For both schedule table and task can activate a task or set an event, we use different request and response channels and replies to distinguish them: we use another different channel SAT to denote activating a task in schedule table instead of AT in task to distinguish whether it is the schedule table or the task call for activating a task, because the identifier do not contain any additional information. In addition, we even use different replies (OK_S and $Pause_S$) to distinguish the response from OS scheduler for the same request in two different ways. It's the same reason that we use SSE in schedule table while use SE in task to set a event to any extended task and channel $Sresponse$ in OS response to schedule table while using $response$ in response to task in both the services $SActivateTask$ and $SSetEvent$.

Expressing the actions dealing with the records in the table is trivial so that we use actions $mdfTable$, $callScheduler$ and $ScallScheduler$ to replace the functionality of the three processes in timed CSP. The schedule table's scheduling policies is also very important to the whole system. The AUTOSAR operating system introduces the time sequence of the initial offset to decide the running schedule table, it forbids any two schedule tables running synchronously at any time. The system always schedule the schedule table whose initial offset is the least, to avoid deadlocks. The modeling of the schedule table's scheduling policies $STScheduler$ is similar to those of $CallScheduler$, which can be found in [14].

V. Modeling EMS Application

In this section, we use the EMS application to illustrate the utility of our method. EMS is one of the most important applications in automotive domain, it requires strict time constraints to ensure its proper running. The EMS includes two parts, i.e, the control part and the execution part, which will be introduced in the following. We adopt the *schedule table* and *task* mechanism in AUTOSAR to describe the control part. For the execution part, we rely on timed CSP to simulate the parallel features of four different strokes in four cylinders.

A. Architecture of EMS

Fig.6 shows the basic architecture of EMS, including the control part and the execution one. The application involves four basic steps, the first three steps belong to the control part and the last step belongs to the execution one. These four steps are:

Step 1. The application initializes the automotive Electronic Control Unit (ECU) and pins of related peripheral hardware. This first job is one-shot, which is expressed by $TaskS_0$ in the following part.

Step 2. It samples environmental parameters on the pressure and temperature of intake air-flow and coolant temperature

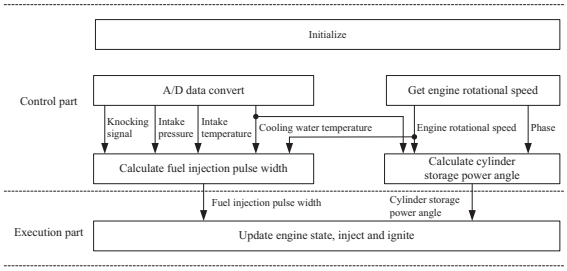

Fig.6 The Procedure of EMS

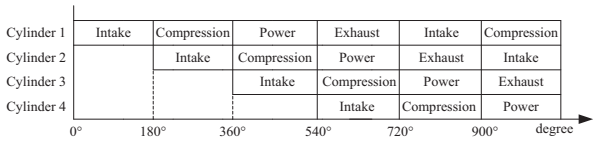

Fig.7 Pipeline of Combustion in the Four-cylinder Engine

according to A/D sampling signal, all the actions are presented in $TaskS_2$. Meanwhile, this step gets engine rotational speed according to enhanced Time Processor Unit (eTPU) signal, which is expressed in $TaskS_1$ in the following part. The former job executes periodically with 2 unit time duration and the latter job with 1 unit time.

Step 3. It calculates fuel injection pulse width to decide the amount of fuel injection according to environmental parameters. It also yields cylinder storage power angle to decide when to ignite four cylinders. All the actions are done in $TaskS_3$, which will be explained in next subsection. According to the requirements of EMS, this job takes place periodically with 10 unit time duration.

Step 4. At last, the application updates the engine state and triggers the combustion process of four cylinders. The four strokes in four cylinders are conducted in a specified pattern. This pattern, kind of "pipeline" execution, will be discussed in detail in the process $Cylinder$ that will be presented in the following, once started, the four cylinder may not terminate.

Step 4 is the execution part in EMS, which describes the Internal Combustion Engine (ICE) work. Since the ICE may be single-cylinder or multi-cylinder, in this paper, we consider four-cylinder engine in the implementation, the whole combustion processes are arranged as Fig.7. As it shows, four strokes, i.e., *intake*, *compression*, *power* and *exhaust* take place in the cylinder sequentially. And each combustion process in the cylinders makes contribution to overall driving torque. In such multi-cylinder engines, the single combustion process need to be coordinated according to a specific pattern. This kind of coordination aims to reduce vibrations during the combustion processes as much as possible. From Fig.7, we can see that the combustion process of each cylinder has a relative offset $180°$. The combustion processes are independent of each other, meaning that no two combustion processes in any two cylinders happen at the same time. This property ensures a smooth running state of the automobile.

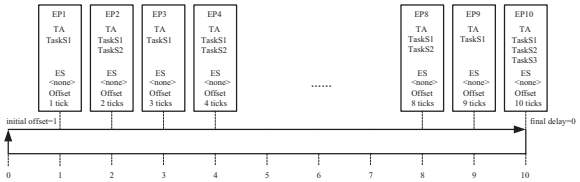

Fig.8 The Schedule Table of EMS Control Part

B. Modeling EMS

We propose an approach to model EMS based on the AUTOSAR OS which will be explained in detail in two parts: modeling the control part and the execution part.

Modeling the Control Part

We abstract four tasks from four main jobs in the control part described before. $TaskS_0$ is a one-shot task to do initialization work. $TaskS_1$ is responsible for sampling A/D data, $TaskS_2$ gets the rotational speed of the engine, $TaskS_3$ updates the engine state and triggers the combustion process in four cylinders. Considering that $TaskS_1$, $TaskS_2$ and $TaskS_3$ are periodical tasks, we adopt the schedule table mechanism to schedule these three tasks. Hence, we arrange the control part of the EMS to be four tasks and one schedule table. Fig.8 shows this repeating schedule table with 10 duration whose final delay is 0. We can conclude that when $TaskS_3$ takes place, $TaskS_1$ must have happened 10 times and $TaskS_2$ 5 times.

We use $TaskS_i$ and $Task_i$ to express the task whose identifier is i, the former one indicates all the jobs in the actual part of EMS respectively, the latter one is the process we presented in section 4, which means the jobs can use the services provided by AUTOSAR OS. Here, we present each jobs $TaskS_i$, their definitions are listed below:

$$
\begin{aligned}
TaskS_0 &=_{df} response?0.OK \rightarrow initial \rightarrow Task_0;\\
TaskS_1 &=_{df} response?1.OK \rightarrow rpm!mid \rightarrow phase!mid \rightarrow\\
&\quad Task_1;\\
TaskS_2 &=_{df} response?2.OK \rightarrow knock!mid \rightarrow\\
&\quad intake\_pressure!mid \rightarrow coolant\_temp!mid \rightarrow\\
&\quad Task_2;\\
TaskS_3 &=_{df} response?3.OK \rightarrow rpm?mid \rightarrow phase?mid \rightarrow\\
&\quad knock?mid \rightarrow intake\_pressure?mid \rightarrow\\
&\quad coolant\_temp?mid \rightarrow width\_value!mid \rightarrow\\
&\quad TDC\_ING!mid \rightarrow Task_3;
\end{aligned}
$$

$TaskS_0$ firstly receives the reply from system, and then do the initialization, here we use an execution $initial$ to represent the whole initialization action, at last it can do any services of the fourteen ones by the external selection just as described in the OS model. $TaskS_1$, $TaskS_2$ and $TaskS_3$ also receive the system's reply at first. $TaskS_1$ can send the engine rotational speed to $TaskS_3$, we use the signal mid and channel rpm and channel $knock$ to express the communication of the two tasks. The message between $TaskS_2$ and $Task_3$ are knocking sinal, intake pressure and cooling water temperature, we use the same signal mid and three different channels: $knock$, $intake\_pressure$, $coolant\_temp$ to describe their communications. $TaskS_3$ not only communicates with $TaskS_1$ and $TaskS_2$, but also triggers the combustion process of four cylinders, which is the process $Cylinder$ presented in the

following. We use two different channel $width\_value$ and TDC_ING to express the communication between $TaskS_3$ and $Cylinder$. All the things that $Task_0$, $Task_1$, $Task_2$ and $Task_3$ can do is the same services as those in OS.

We adopt $ScheduleTables_{EMS}$ to schedule $TaskS_1$, $TaskS_2$ and $TaskS_3$, since all of the three tasks are periodic, the $ScheduleTables_{EMS}$ is repeating. Its definition is shown as below:

$$
\begin{aligned}
&ScheduleTables_{EMS} =_{df} Sresponse?0.OK_S \rightarrow ScheduleTable;\\
&ScheduleTable =_{df} EP_0; EP_1; EP_0; EP_1; EP_0; EP_1;\\
&\qquad EP_0; EP_1; EP_0; EP_2; ScheduleTable;\\
&EP_0 =_{df} Wait[1]; SAT!0.1 \rightarrow (Sresponse?1.OK \rightarrow Skip \Box\\
&\qquad Sresponse?1.Pause \rightarrow Sresponse?1.OK \rightarrow Skip);\\
&EP_1 =_{df} Wait[1]; SAT!0.1 \rightarrow (Sresponse?1.OK \rightarrow Skip \Box\\
&\qquad Sresponse?1.Pause \rightarrow Sresponse?1.OK \rightarrow Skip);\\
&\qquad SAT!0.2 \rightarrow (Sresponse?2.OK \rightarrow Skip \Box\\
&\qquad Sresponse?2.Pause \rightarrow Sresponse?2.OK \rightarrow Skip);\\
&EP_2 =_{df} Wait[1]; SAT!0.1 \rightarrow (Sresponse?1.OK \rightarrow Skip \Box\\
&\qquad Sresponse?1.Pause \rightarrow Sresponse?1.OK \rightarrow Skip);\\
&\qquad SAT!0.2 \rightarrow (Sresponse?2.OK \rightarrow Skip \Box\\
&\qquad Sresponse?2.Pause \rightarrow Sresponse?2.OK \rightarrow Skip);\\
&\qquad SAT!0.3 \rightarrow (Sresponse?3.OK \rightarrow Skip \Box\\
&\qquad Sresponse?3.Pause \rightarrow Sresponse?3.OK \rightarrow Skip);
\end{aligned}
$$

$ScheduleTables_{EMS}$ firstly receives the system's reply, then as we metioned before, the delay of any two adjacent expiry points in the schedule table for the application is one unit time. The schedule table contains three kinds of expiry points: one is responsible for request to the system to activating $TaskS_1$, one is responsible for request to the system to activating both $TaskS_1$ and $TaskS_2$, and the last one is responsible for request to the system to activating $TaskS_1$, $TaskS_2$ and $TaskS_3$.

Modeling the Execution Part

As for the combustion process of four cylinders, we rely on timed-CSP to simulate the parallel features of strokes in four cylinders. Four strokes under execution of four cylinders are different from each other at the same time. This parallel process is actually conducted by hardware under the supervision of software. The model of the cylinder pipeline is:

$$
\begin{aligned}
&Cylinder =_{df} width\_value?mid \rightarrow TDC\_ING?mid \rightarrow pip_0;\\
&pip_0 =_{df}\\
&\quad Wait[1]; atomic\{cyl_0\_intake \rightarrow Skip\};\\
&\quad Wait[1]; atomic\{cyl_0\_compression \rightarrow Skip \| cyl_1\_intake \rightarrow\\
&\quad Skip\};\\
&\quad Wait[1]; atomic\{cyl_0\_power \rightarrow Skip \| cyl_1\_compression \rightarrow\\
&\quad Skip \| cyl_2\_intake \rightarrow Skip\}; pip_1;\\
&pip_1 =_{df}\\
&\quad Wait[1]; atomic\{cyl_0\_exhaust \rightarrow Skip \| cyl_1\_power \rightarrow\\
&\quad Skip \| cyl_2\_compression \rightarrow Skip \| cyl_3\_intake \rightarrow Skip\};\\
&\quad Wait[1]; atomic\{cyl_0\_intake \rightarrow Skip \| cyl_1\_exhaust \rightarrow\\
&\quad Skip \| cyl_2\_power \rightarrow Skip \| cyl_3\_compression \rightarrow Skip\};\\
&\quad Wait[1]; atomic\{cyl_0\_compression \rightarrow Skip \| cyl_1\_intake \rightarrow\\
&\quad Skip \| cyl_2\_exhaust \rightarrow Skip \| cyl_3\_power \rightarrow Skip\};\\
&\quad Wait[1]; atomic\{cyl_0\_power \rightarrow Skip \| cyl_1\_compression \rightarrow\\
&\quad Skip \| cyl_2\_intake \rightarrow Skip \| cyl_3\_exhaust \rightarrow Skip\}; pip_1;
\end{aligned}
$$

When the $Cylinder$ process receives signals from $TaskS_3$ through channel $width\_value$ and TDC_ING, it starts the pipeline, we divide the pipeline into two parts: pip_0 is responsible for the unsteady level of the pipeline, and pip_1 for the steady level of the pipeline, which is the level that for one step, the four cylinders do mutually exclusive actions, but for every cylinder, it do its own actions in a sequential order. At every moment, in order to make the four actions of the four cylinder at the same time. We force every step to be an atomic action.

The EMS application is modeled as the parallel composition of four tasks, one schedule table, one process $Cylinder$ and OS scheduler, its definition is listed as below:

$$
\begin{aligned}
&Application =_{df} (TaskS_0 ||| TaskS_1 ||| TaskS_2 ||| TaskS_3)\\
&\qquad \| ScheduleTableS_{EMS} \| OSschedule \| Cylinder
\end{aligned}
$$

The process $Application$ represents the EMS's actions, each $TaskS_i$ and $ScheduleTableS_{EMS}$ can send requests to the AUTOSAR OS, the process $OSschedule$ receives the request and deals with them.

VI. Properties

For the purpose of verifying whether the models obey the specifications and requirements of AUTOSAR OS and EMS application or not, we propose eight safety properties based on the specifications and requirements by first-order logic formulas. These properties involve task management, resource assignment and schedule table scheduling in AUTOSAR OS and cylinder properties in the EMS application. Since we had explained four properties: **mutual exclusion (ME)**, **priority-sensitive scheme (PS)**, **excluded priority inversion (EPI)** and **deadlock freedom (DF)** of task scheduling and resource assignment in [14], AUTOSAR OS and EMS application essentially conforms to those properties, here we focus on other four important properties listed below. The notation $STIDS =_{df} 0...(s-1)$ contains all the schedule table identifiers of s schedule tables both in AUTOSAR OS and EMS application, while $CylIDS =_{df} 0...3$ contains the identifiers of four cylinders in the EMS application.

Property 1: Mutual Exclusion of Schedule Tables (MEST)

$$
\begin{aligned}
&\forall i, j \in STIDS \bullet\\
&(SchTabState_i == running \wedge j \neq i\\
&\qquad \Rightarrow SchTabState_j \neq running)
\end{aligned}
$$

At any time at most one schedule table is running during the system execution. It means that when $SchTab_i$ is running, others cannot run.

The following three properties are all about the cylinder, the global two-dimensional array $CylState[4][4]$ records the 4 cylinder's four strokes whose default value is $unused_C$, variable $status$ indicates the pipeline of the cylinders is stable or not, while $data$ means the states of cylinders are readable or not since during the atomic action in the pipeline, the modifications of the states may not complete, and we cannot access the states. When $data$ is unlocked, we can access the data.

Property 2: Fixed Started Order of Cylinders (FSOC)

$$
\begin{aligned}
&\forall i, j \in CylIDS, k \in \{0,1,2,3\} \bullet\\
&(status == unstable \wedge i < j \wedge data == unlock\\
&\qquad \wedge CylState_{ik} == intake\\
&\qquad \Rightarrow CylState_{jk} == unused_C)
\end{aligned}
$$

The cylinders should start in proper order, that means in the unstable state if the former cylinder starts, its following

cylinders cannot start. We use the identifier value to indicate the order of the cylinder, the less the value is, the earlier the cylinder starts. The value of $CylState_{ik}$ and $CylState_{jk}$ correspond to the same step k.

Property 3: Mutual Exclusion between Cylinders (MEC)

$$\forall i, j \in CylIDS, k \in \{0,1,2,3\} \bullet$$
$$(status == stable \wedge data == unlock \wedge i \neq j \Rightarrow CylState_{ik} \neq CylState_{jk})$$

According to the EMS application requirements, the running states of the four cylinders cannot be the same once they all started. So if the pipeline of the the four cylinders is stable, any two cylinder at the same step k cannot be the same.

Property 4: Fixed Order of Four Strokes (FOFS)

$$\forall i \in CylIDS, j \in \{0,1,2,3\} \bullet$$
$$(status == stable \wedge data == unlock \wedge CylinderState_{ij} == intake \Rightarrow CylState_{i((j+1)\%4)} == compression \wedge CylState_{i((j+2)\%4)} == power \wedge CylState_{i((j+3)\%4)} == exhaust)$$

When the pipeline is stable, for a single cylinder, the process always follows the fixed sequence: intake, compression, power and exhaust. So when the state of $cylinder\ i$ in the step j is $intake$, then the states of the next three steps are $compression$, $power$ and $exhaust$ undoubtedly. Because of limited memory, we save the states of the 4 cylinders as circular queue, if the column index increases to 3, then the value of its following index is updated to 0.

VII. Implementation

PAT is a CSP-based tool designed to apply model checking techniques for simulating the behaviors and verifying the properties of concurrent systems. We check the correctness of AUTOSAR OS model and EMS application model through simulation and verification in PAT. We have verified eight properties in all. The eight properties have been translated into assertions in PAT. Table 2 shows the verification results of AUTOSAR OS model and EMS application model. There

Table 2. Verification results

properties type	properties name	verification results
OS properties	MEST	satisfied
	ME	satisfied
	PS	satisfied
	EPI	satisfied
	DF	satisfied
EMS properties	FSOC	satisfied
	MEC	satisfied
	FOFS	satisfied

are 989 lines of our models and assertions, since we express some properties by contradiction, the maximum memory consumption even goes to 6.46 GB during verifying. From the verification results in PAT, we can conclude that AUTOSAR OS and EMS application conform to the above important properties.

VIII. Conclusion and Future Work

This paper has employed timed CSP in modeling and verification of the AUTOSAR Operating System and Engine Management System application. The OS and application are modeled as timed CSP processes, all the properties are expressed in first-order logic formula. The formal models and properties are implemented in PAT. In the future, we will analyze other parts of AUTOSAR OS and related applications.

Acknowledgment

This work was partly supported by East China Normal University Project of Funding Graduate Study Abroad, Domestic Visiting and Attend International Conference, the Danish National Research Foundation and the National Natural Science Foundation of China (Grant No. 61061130541). And, it is also supported by NBRPC(Grant No. 2011CB302904), National High Technology Research and Development Program of China (Grant Nos. 2011AA010101 and 2012AA011205), NSFC(Grant Nos. 61021004 and 91118008), Shanghai Science and Technology Commitment Project (No. 12511504205), Open Project of Key Lab of Information Network Security, Ministry of Public Security(No. C12604) and Shanghai Knowledge Service Platform Project (No. ZF1213).

References

[1] AUTOSAR, *Operating System Specification 4.0.0*, The AUTOSAR group, 2009.

[2] An introduction to AUTOSAR, http://www.autosar.org/.

[3] An introduction to OSEK/VDX, http://www.osek-vdx.org/.

[4] OSEK/VDX, *Operating System Specification 2.2.3*, 17th ed. The OSEK group, compiled by: Jochem Spohr MBtech, February 2005.

[5] Uwe Kiencke and Lars Nielsen. Automotive Control Systems for Engine, Driveline and Vehicle. Springer, 2005.

[6] Kay Klobedanz, Christoph Kuznik, Andreas Thuy and Wolfgang Müller, Timing modeling and analysis for AUTOSAR-based software development - a case study, DATE, 2010, 642-645.

[7] Libor Waszniowski and Zdenek Hanzalek, Analysis of OSEK/VDX based Automotive Applications, 2006.

[8] Gerwin Klein, Ralf Huuck and Bastian Schlich, Operating System Verification, J. Autom. Reasoning, 42, 2-4, 2009.

[9] Libor Waszniowski and Zdenek Hanzálek, Formal verification of multitasking applications based on timed automata model, Real-Time Systems, 38, 1, 2008, 39-65.

[10] Steve Schneider, An Operational Semantics for Timed CSP, Inf. Comput., 116, 2, 1995, 193-213.

[11] C. A. R. Hoare, Communicating Sequential Processes. Prentice Hall International Series in Computer Science, 1985.

[12] George M. Reed and A. W. Roscoe, A Timed Model for Communicating Sequential Processes, Theor. Comput. Sci., 58, 1988, 249-261.

[13] Jim Davies and Steve Schneider, A Brief History of Timed CSP, Theor. Comput. Sci., 138, 2, 1995, 243-271.

[14] Yanhong Huang, Yongxin Zhao, Longfei Zhu, Qin Li, Huibiao Zhu, Jianqi Shi, Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP, TASE, 2011, 142-149.

[15] Dominique Bertrand, Sébastien Faucou and Yvon Trinquet, An Analysis of the AUTOSAR OS Timing Protection Mechanism, ETFA, 2009, 1-8.

[16] Jun Sun, Yang Liu and Jin Song Dong, Model Checking CSP Revisited: Introducing a Process Analysis Toolkit, ISoLA, 2008, 307-322.

[17] *Process Analysis Toolkit PAT home page*, National University of Singapore, http://www.comp.nus.edu.sg/pat.