Modeling Automotive Software Requirements with Aspectual Models

Xiaojian Liu, Xuqin Yan, Yang Li, Xiaobo Che and ChengYong Mao
Shandong Provincial Key Laboratory of Automotive Electronic Techniques
Institute of Automation, Shandong Academy of Science, Jinan, China, 250014

***Abstract*—Modeling software requirements is one of the grand challenges of ECU (Electronic control Unit) development. One of the main issues existing in this domain is how to organize the large amount of requirement information in a concise and manageable means to facilitate the further requirements verification and system design activities. To this end, we propose a requirement modeling framework, based on the philosophy of *separation of concerns* and the *formal modeling* techniques. The main advantages of our approach are of two-folds: (1) Analyzers only need to concentrate on one aspect each time, thus the complexity of the requirements modeling is decreased, and accordingly the models' understandability is enhanced; (2) The adoption of formal techniques allows us to simulate and verify the properties of the requirements in the early stage of development, therefore the quality of requirements can be improved.**

***Keywords*-Model-driven development; Requirements modeling; Automotive software; Architecture description language; Timed automata**

I. Introduction

Software takes an increasingly essential role in the construction of modern automotive electronic systems. About 90% innovations of ECU directly or indirectly come from software technologies[1]. Modern automotive software development is facing various grand challenges, one of which is of the requirements modeling.

The difficulties of requirements modeling for automotive software come from three aspects: (1) *data complexity*. A typical high-end automotive system usually includes more than 2000 signals, which exchange among sensors, ECUs and actuators. We must find a way to effectively manage so large amount of signals; (2) *behavioral complexity*. The majority of automotive systems are hard or soft real-time systems, thus a requirement specification should contain all the necessary timing requirements to facilitate further analysis; (3) *environment complexity*. The physical environment of an ECU may be physical environment, mechanical components to be controlled, senors, actuators, other ECUs and even human operators. The description of the environment serve as an indispensable part in requirements modeling.

To tackle this problem, we follow the *separation of concerns* philosophy and the *formal modeling* techniques. The requirements for a complicated automotive software can be divided as three aspects: *structure*, *behavior* and *communication*. The structure aspect mainly concerns about the input and output interfaces of a functionality, and the decomposition of a functionality into sub-functionalities as well; The behavior aspect describes how a functionality reacts to the input stimulus by producing the output; and the communication aspect specifies the signals exchanged between the target system and its environment. To model these three aspects, we choose three formal techniques to describe them respectively: *EAST-ADL2*[2], an architecture description language dedicated to automotive electronic systems, *timed automata*[3] and *signal matrix*, a 2-dimension table recording the signals and their properties. To reason about the requirements model, we combine these three aspectual models together, and transform them into UPPAAL specifications, which can be simulated and verified by UPPAAL tool[4].

The main contributions of this paper are of: (1) by decomposing the whole requirements into separated aspects, we can decrease the complexity of the requirements modeling, and accordingly enhance the models' comprehensibility; (2) The adoption of formal techniques and tools allows us to simulate and verify the properties of the requirements in the early stage of development, therefore the quality of requirements can be improved.

The remainder of the paper is organized as follows. Section 2 introduces related works; Section 3 introduces the three modeling notations for the three requirement aspects; Section 4 illustrates how to transform the aspectual models into an UPPAAL specification, and how to simulate and verify the requirements by using UPPAAL tool; and finally Section 5 concludes the paper.

II. Related Works

We mainly compare our work with the works of *architecture description languages* and *component-based* development for automotive applications.

Besides EAST-ADL2, SAE AADL(Architecture Analysis and Design Language)[5] are also widely used in automotive systems. In contrast to AADL, EAST-ADL2 particularly focuses on the automotive domain. EAST-ADL2 includes modeling entities to describe features, requirements, variability, software and hardware components. However, the main weakness of EAST-ADL2 and AADL is that they does not specify the behavior modeling entities, because of the very different requirements in the various automotive application domains. In this paper, we extend EAST-ADL2 with timed automata formalism, which allows us to model, simulate and verify the system behavior in the early phase of system development.

Similar to our idea, SaveCCM[6], a component-based design methodology for automotive system development, also separates a software component into structure and behavior aspects. The structure model of SaveCCM is much like EAST-ADL2, but its behavior is modeled with task automata formalism[7], an extension of timed automata with tasks. Therefore the behavior model of SaveCCM can be used for the schedulability analysis. Obviously, the main purpose of SaveCCM is for system design activities, however, the aim of this paper is mainly for requirements modeling.

III. Modeling Notations

A. EAST-ADL2

EAST-ADL2 is an architecture description language specific to automotive electronic systems. It describes the whole vehicle system from several abstraction layers: vehicle layer, analysis level, design level, implementation level and operational level. As this paper only discusses the requirement modeling, we choose a subset of EAST-ADL2, i.e., the function modeling package of EAST-ADL2, as the modeling language to specify the structure of a functionality.

With EAST-ADL2 language, a functionality is modeled as an element of `ADLFunctionType`, which contains a set of ports (the elements of `ADLFlowPort` or `ADLClientServerPort`), a set of function prototypes(the elements of `ADLFunctionPrototype`), and a set of connectors(the elements of `ADLConnectorPrototype`). A function prototype appears as a part of `ADLFunctionType` and is itself typed by an `ADLFunctionType`. Connectors can be classified as two categories: *delegate* and *assembly*, the former connect the ports of a function type and the ports of its contained function prototypes, and the latter connect the ports between function prototypes. Every flow port is associated with a data type, which specifies the properties of the data exchanged via the port.

Example. The BCM(Body Control Module) is a ECU which controls the vehicle body components, such as doors, tailgate, hood, windows and lights etc. The BCM usually contains a number of functionalities, such as Perimeter Alarm and Door Locking/Unlocking etc. Each function has a group of flow ports and client-server ports. The Figure 1 illustrates the structure model of the BCM. Here we only demonstrate two functionalities of the BCM: `LOCKCONTR` and `ALARM`. The connectors between the ports of BCM and the ports of `LOCKCONTR` or `ALARM` are delegate connectors, and the connectors between the ports of `LOCKCONTR` and `ALARM` are assembly connectors which combine both function prototypes together.

The structure model of the BCM actually specifies the set of common phenomena (including events and variables) shared with the BCM and its environment. We can divide the common phenomena into 4 kinds of sets: *input_events*, *output_events*, *input_variables* and *output_variables*. For the structure model of the BCM, these sets are:

- *input_events*={OPEN_Doors, LOCK_Remoter, UNLOCK_Remoter}
- *output_events*={LOCK_DoorEnable, UNLOCK_DoorEnable, OPEN_DoorEnable, TurnlightEnable, HORNEnable}
- *input_variables*={DoorStatus, VehicleSpeed, AirbagStatus}
- *output_variables*={DoorStatus}.

Note that, EAST-ADL2 language specifies little about the behavior modeling of a functionality, because of the very different requirements in the various automotive application domains. In this paper, we extend EAST-ADL2 with timed automata formalism as the model to specify the behavior of a functionality. Timed automata formalism allow us to model the majority of timing behavior patterns existing in automotive applications, and furthermore, its tool support, UPPAAL, will facilitate us to simulate and verify the requirements in the early stage of the development.

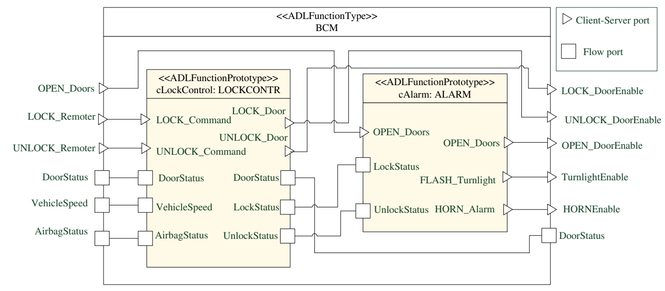

Figure 1. The structural model of the BCM

B. Timed Automata

Timed automata formalism is an extension of traditional untimed automata, in which time clocks and timed state invariants are introduced to describe the timing behavior of a system. A timed automaton interacts with its environment through synchronized actions and global variables. To simulate and verify the timed automata model of a target system, we must model both the behavior of the target system and that of its environment, they are composed parallel as a network of timed automata.

Let C be a set of clocks, $B(C)$ is the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, >, \geq\}$. A timed automata is a tuple (L, l_0, A, E, I), where L is a set of locations, $l_0 \in L$ is the initial location, C is the set of clocks, A is a set of actions, co-actions and the internal τ-action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I : L \to B(C)$ assigns invariants to locations.

UPPAAL modeling language, a version of timed automata, extends the original timed automata with a number of features, one of which is that the expressions in UPPAAL can range over clocks and bounded integer variables (or arrays of these types). This feature improves the expressiveness of timed automata by allowing us to model the complicated guard conditions and assignments. In the next section, we will give examples of timed automata for the functionality `LOCKCONTR` and its environment `Driver`, and discuss how to simulate and verify the behavior model using UPPAAL tool.

C. Signal Matrix

On the requirement level, the communication aspect of an automotive software is mainly presented as a set of signals and their properties. An input signal may comes from networks(such as CAN/LIN bus), sensors, or some other physical pulse lines; likewise, an output signal may be sent to networks, actuators or hardware/software drivers. Each signal has a number of properties, including *name*, *period*, *priority*(if the signal is transformed through CAN bus), *size*(number of bits), *unit*, *sender*, *receiver*, *resolution*, *offset*, *range of values*, *default value* and *value descriptions*. Here, we use a *Signal Matrix*, a two-dimension table, to represent all the relevant signals and their properties.

Example. The BCM application requires all of 91 signals, in which about 20 signals come from CAN bus, and the rest of them come from sensors or go to actuators. The following Table I demonstrates a part of the signal matrix. In this table, signals MasterVehicleSpeed and ActualEngineTorque are transmitted through CAN bus, thus they need the priorities to avoid the conflictions in the signal transformation. The remainder signals either come from sensors or go to actuators, therefore they does not need any priorities. The columns *resolution* and *offset* are used to transform signal values to its actual values. For instance, for the signal ActualEngineTorque, its actual values can be derived from its signal values via the relation: $actual\_values = signal\_values \times 0.5 - 100$.

IV. Simulation and Verification

A. Transform the Aspectual Models to UPPAAL

The purpose of the formal modeling of requirements is to validate the models and to reason about the properties of the requirements to ensure their correctness in the early development stage. In this section, we will represent how to simulate the models and verify some very basic properties of the requirements with the help of UPPAAL tool. For the following discussion, we use *StrModel* and *BehModel* to denote the structure model and behavior model for a functionality.

To simulate and verify the requirement model of a functionality, we adopt the following steps: (1)Transform *strModel* and *BehModel* to an UPPAAL specification; (2)Model the behavior of the functionality's environment in terms of timed automata as well; (3) Combine the UPPAAL specifications of both the functionality and its environment together, and perform the simulation and the verification by using UPPAAL tool.

A complete UPPAAL specification includes four parts: *global declarations*, *templates* for timed automata, *process assignments* and a *system definition*. The global declarations declare clocks, data variables, channels, and constants, which are shared with all of the timed automata in a system; the templates are of the text descriptions of timed automata equipped with lists of formal parameters and with local declarations of clocks, data variables, channels, and constants; The process assignments instantiate the templates by substituting actual parameters for the formal ones. An instantiated template is called a *process*; A system definition consists of a list of process.

We propose a number of transformation rules to transform the models *strModel* and *BehModel* to UPPAAL specifications:

R1: *input_events* and *output_events* of *strModel* are transformed as global channels;
R2: *input_variables* and *output_variables* of *strModel* are transformed as global data variables;
R3: Transform *BehModel* and its local clocks, data variables as a template with local clocks and variables;
R4: Specify the environment of a functionality as the templates as well;
R5: The variables and the events which are expected to be instantiated when creating a process, are transformed as the formal parameters of the templates;
R6: Transform the composition of the functionality and its environment as a system definition.

Example. We present a simplified version of the functionality LOCKCONTR to show how to build the UPPAAL specification. LOCKCONTR is an important function in the BCM, which controls the locking and unlocking of the vehicle doors under various circumstances. The environment of LOCKCONTR includes drivers, turnlight actuators, horn actuators, and other ECUs such as EMS(Engine Management System) and TCU(Terminal Control Unit). As an example, here we only consider the drivers as the unique environment of LOCKCONTR. The structure and behavior models of LOCKCONTR are demonstrated in Figure 2 and 3. The functionality LOCKCONTR interacts with the drivers through the ports, whose explanations are illustrated in the Table II.

Following the above transformation rules, we translate the structure model into the global declarations and the templates:

Table I
THE SIGNAL MATRIX FOR THE BCM

name	*period*	*priority*	*size*	*unit*	*sender*	*receiver*	*resolution*	*offset*	...
LockControlofDoor	100ms	-	2	bit	lock/unlock sensor	BCM	N/A	0	...
OpenControlofDoor	100ms	-	2	bit	open/close sensor	BCM	N/A	0	...
MasterVehicleSpeed	20ms	0x088	8	KPH	EMS	BCM	1KPH	0	...
ActualEngineTorque	10ms	0x082	12	NM	EMS	TCU	0.5NM	-100	...
DoorLock	150ms	-	1	bit	BCM	lock actuator	N/A	0	...
DoorAjarStatus	150ms	-	1	bit	BCM	TCU	N/A	0	...
DoorLockStatus	150ms	-	1	bit	BCM	TCU	N/A	0	...

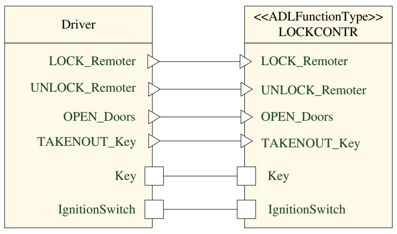

Figure 2. The structure model of LOCKCONTR

Table II
EXPLANATIONS OF THE PORTS

LOCK_Remoter	LOCK button on the remoter is pressed
UNLOCK_Remoter	UNLOCK button on the remoter is pressed
OPEN_Doors	Vehicle doors are opened
TEKNOUT_Key	Key is taken out of ignition switch
Key	The variable representing the state of 'ignition key' Key=IN, 'key' being in ignition switch Key=OUT 'key' being out of ignition switch

IgnitionSwitch	The variable representing the state of 'ignition switch'
	IgnitionSwitch=ON, 'switch' being in ON position
	IgnitionSwitch=OFF, 'switch' being in OFF position

Global declaration
```
const int[0,1] IN=1, OUT=0;
const int[0,1] OFF=0, ON=1;
int[0,1] Key:=OUT,IgnitionSwitch:=OFF;
chan UNLOCK_Remoter,LOCK_Remoter;
chan OPEN_Doors;
chan TAKENOUT_Key;
```
Template LOCKCONTR(`int[0,254] VehicleSpeed`)
```
clock t;
int[0,1] KeyWarning:=0;
```
System definition
```
LOCKCONTR30=LOCKCONTR(30);
system LOCKCONTR30, Driver;
```

With the help of UPPAAL tool, we can simulate the above UPPAAL specification in a step-by-step manner. The simulation illustrates all possible runs of the combined system LOCKCONTR || Driver, which will help us to find the potential errors in the requirement models.

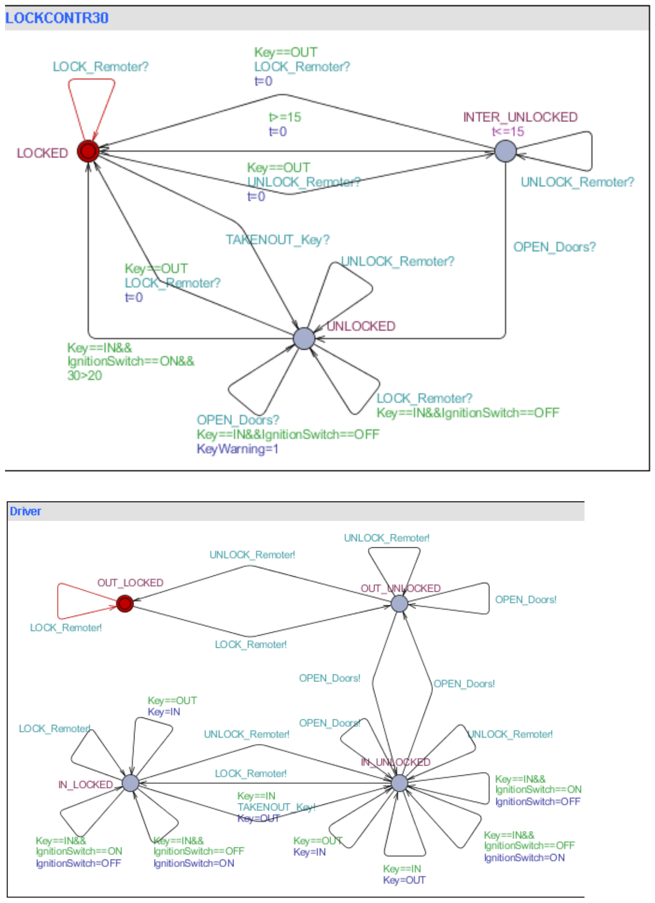

Figure 3. Timed automata for LOCKCONTR and Driver

B. Verification

Based on the formal UPPAAL model, we can verify some properties of the requirements model. Generally, a property may be either a very general purpose one which must hold for any valid requirement model, such as *deadlock freedom*, or a domain-specific one which should be identified according to the knowledge of an application domain.

The problem of property checking is really a problem of model checking. The model is a system definition just like the UPPAAL specification illustrated in the previous section; and the properties to be checked are usually represented by a subset of TCTL formula[8], which consist of path formula and state formula: A$\Box\phi$, E$\Diamond\phi$, A$\Diamond\phi$, E$\Box\phi$ and $\phi \rightsquigarrow \psi$. Due to space reason, here we do not explain their semantics furthermore, the interesting readers please refer [4] for details.

We present a number of very basic properties which should hold for all the requirement models of automotive applications. Let *TA* be the timed automata for the behavior of a functionality, and *Env* be the environment of *TA*, $(TA \| Env)$ be the parallel composition of *TA* and *Env*.

1) Requirement models should be *deadlock freedom*: A$\Box$ `not deadlock` must hold for $TA\|Env$.
2) Every safe state is *reachable*: E$\Diamond$ $TA.l_{safe}$ and E$\Diamond$ $Env.l_{safe}$ must hold, where $TA.l_{safe}$ and $Env.l_{safe}$ are of the labels representing safe states for TA and Env.
3) Every intermediate state will eventually move to one of the stable states: $TA.l_{inter} \rightsquigarrow TA.l_{stable}$ must hold, where $TA.l_{inter}$ and $TA.l_{stable}$ denote the intermediate and stable states of TA.
4) Some invalid state compositions should not be reached: E$\Diamond$($TA.l_1$ and $Env.l_2$) does not hold if $(TA.l_1, Env.l_2)$ is an invalid state composition.

Example. We use the requirement model illustrated in Figure 3 to show the verification of the basic properties:

- `LOCKCONTR30` and `Driver` should be deadlock freedom: `A[] not deadlock`.
- The state `INTER_UNLOCKED` of `LOCKTROL30` is an intermediate state with the time invariant `t<=15`. It will eventually move to the stable states:
  `LOCKTROL30.INTER_UNLOCKED`
  `--> (LOCKCONTR30.LOCKED or`
  `LOCKCONTR30.LOCKED).`
  This property can be used to check the omission of timeout transitions from an intermediate state.
- The state `LOCKED` of `LOCKCONTR30` should synchronize only with the states `OUT_LOCKED` and `IN_LOCKED` of `Driver`, thus
  `E<>(LOCKCONTR30.LOCKED and`
  `Driver.OUT_UNLOCKED)` and
  `E<>(LOCKCONTR30.LOCKED and`
  `Driver.IN_UNLOCKED)` should not hold.

The results of the verification will help us to ensure the correctness of the requirements model. However, the properties listed above are only the very basic ones, thus themselves only are not sufficient for ensuring the total correctness of the requirements model. To get more confidence of its correctness, we need to identify and verify more properties.

V. Conclusion

In this paper, we introduce how to model the requirements of automotive software in terms of formal aspectual models. The main advantages of this approach are of: (1) decreasing the descriptive complexity of the requirement modeling; and (2) allowing us to simulate and verify the models with the help of tools. In the near future, we want to establish an integral requirements model which combines structure, behavior and communication models together. Another work which will be carried out is to verify the completeness and consistency of requirements model with formal methods.

References

[1] M. Broy, I. Kruger, A. Pretschner and C. Salzmann. Engineering Automotive Software. Proceedings of THE IEEE. 95(2): 356-373, Febrary 2007.

[2] EAST-ADL2: http://www.atesst.org/

[3] R. Alur and D. L. Dill. A theory of timed automata. Theor. Comput. Sci., 126(2):183-235, 1994.

[4] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal implementation secrets. In Proc. of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, 2002.

[5] Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis and Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Society of Automotive Engineers (2006)

[6] SAVE Project, http://www.mrtc.mdh.se/SAVE/

[7] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - a tool for modelling and implementation of embedded systems. In TACAS 2002, volume 2280 of Lecture Notes in Computer Science, pages 460-464. Springer-Verlag, April 2002.

[8] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science, pages 414C425. IEEE Computer Society Press, 1990.