Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 274 (2011) 33–50
www.elsevier.com/locate/entcs

Construct Aspectual Models from Requirement Documents for Model-driven Development of Automotive Software

Xiaojian Liu[†], Zhilin Zhu[§]

[†] *Shandong Provincial Key Laboratory of Automotive Electronic Techniques*
Institute of Automation, Shandong Academy of Science, Jinan, China
[§] *Shandong Institute of Business and Technology, China*

Abstract

In the model-driven development of complex software-intensive systems, it is the first class issue how to capture the software models in the very beginning of development. In this paper, we focus on the domain of automotive software, explore how to capture software models from requirement documents. To this end, we mainly investigated these three closely related problems: what requirement information should be elicited from requirement documents; how to organize these information as aspectual models; and how to integrate the aspectual models together to form a complete specification of requirements. This work will help software analyzers to decompose complicated requirement documents into separated concerns, and organize essential requirement information as rigorous models, which will both facilitate simulation and verification of requirements and set up the starting point for the model-driven development.

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

1 Introduction

Model-driven development methodology is a promising approach to developing complex software-intensive systems. It requires that in a development process, each phase is based on the *construction of models*, models in later phases are constructed from those in earlier phases by *model transformations*, and code is an executable model generated from models in the design phase [2]. However, the application of model-driven approach to practical development is not an easy task — one of the difficult problems is how to obtain models at the very beginning phase of the development, i.e., at the requirement analysis phase. At this stage, the software to be developed is represented as a collection of requirement documents which are usually written with natural languages. The informal descriptions, ambiguous statements and unstructured organization usually prevent us from constructing rigorous models.

doi:10.1016/j.entcs.2011.07.005

In this paper, we focus on the domain of automotive software, explore how to construct models from requirement documents. We mainly investigates these closely related problems: what essential requirement information should be elicited from documents; how to organize and represent these information as models; how to integrate individual models together to form a complete requirement specification; and how to support the construction and analysis of models with tools. Although we focus on the context of automotive domain, we believe that the basic ideas and solutions to these problems are also applicable to general complex software-intensive systems.

In the development of automotive systems, developers usually start development from three kinds of requirement documents: *function description document*, which describes the functionalities of the ECU (Electronic Control Unit) to be developed; *application protocol document*, which specifies the signals communicating between the target ECU and its environment (including sensors, actuators and other ECUs); and some *constraints*, including hardware constraints, nonfunctional requirements etc. Separation of requirements into these individual documents will facilitate the understanding and organization of the requirements. However, from software developers' points of view, this separation does not yet help them to efficiently capture the software and manage the development. For example, the designers usually want to find: what's the boundary of the software interacting with its environment; what subfunctions are included in a complicated function; how a functionality reacts to the inputs by producing the outputs; and how an ECU exchanges data and control with other ECUs via communication media. These information are usually scattered over one or more places of a document, in some cases even over several different documents. As a result, when the scale of the software increases, organizing and linking the relevant requirement information together becomes a difficult task.

To tackle this problem, we need to elicit the essential requirement information from documents, and reorganize them as a group of models from which developers can easily understand and design the software. Four kinds of information should be extracted from requirement documents: *Structure information*, *Behavior information*, *Communication information* and *Platform and nonfunctional constraints*. With these information in hand, designers are able to begin their design activities: structure information will help them to design data structures; behavior information will guide them to design control algorithms; communication information will support them to design messages and communication protocols; and platform and performance constraints will provide information for further software allocations and performance testings. In order to represent and organize these information, we should firstly choose suitable notation, which should satisfy the following basic criteria: (1) Each notation only represents a kind of information; (2) The notations should be domain-specific, which can be easily understood and used by engineers; and (3) The representations should be analyzable for correctness assurance. Following these criteria, we choose notations *EAST-ADL2*, *Timed automata* and *Signal matrix* to model the structure, behavior and communication information respectively.

To capture a comprehensive requirement specification, we need to integrate the constructed models together. To this end, we use *4-variable requirement model* [9], a relational theory for requirement specifications, as the common base, and map all the elements of the aspectual models to the terms of 4-variable model. This integration approach provides us a way to understanding how each model contributes to a complete requirement specification.

This work is only in its initial stage. Our final aim is to develop a domain-specific language and an environment in order to support model-driven development for automotive software. At present, we have developed a tool to support the ideas proposed above, and also applied it to the practical development to investigate its applicability. So far, the feedbacks from engineers indicate that the proposed ideas help them to clearly shape the requirements for automotive software, and the tool significantly help them to improve the process of development.

The remainder of the paper is organized as follows. Section 2 introduces what requirement information should be elicited from requirement documents; Section 3 introduces the modeling notations chosen to represent the requirement information; Section 4 describes how to integrate the individual models based on 4-variable requirement model; Section 5 illustrate the tool support for our ideas; Section 6 summarizes related works; and finally Section 7 concludes this paper.

2 Requirement Information

In this section, we use BCM (Vehicle Body Control System) as a running example to illustrate what requirement information should be elicited from documents for automotive software.

BCM is an ECU which controls vehicle body components, its main functionalities include locking/unlocking doors, lifting up/down windows, turning on/off a number of lights, and controlling wipers and washers etc. Here, as an example, we select one of the functionalities, *locking/unlocking vehicle doors* (LOCKCTR for short), to give a concise explanation.

In what follows, we firstly represent some scenarios of LOCKCTR, and then analyze the domain characteristics of LOCKCTR, which will help us to understand what essential requirement information is required to be elicited from documents.

Req1: Drivers can lock and unlock vehicle doors by using a remoter;
Req2: Inside the vehicle, drivers can use central lock system to lock and unlock the doors;
Req3: When the vehicle speed is over 30m/h, doors are automatically locked;
Req4: When a crash happens, doors will be automatically unlocked for rescue;
Req5: Driving motors of locking/unlocking doors must keep at least 200ms.

Structure information includes inputs and outputs, and the decomposition of a functionality. Generally, an input or output of a control system is a function from time domain to value domain. For an input, we may want to observe its

state at some time instant, or want to observe the change of states, i.e., *event*, or both of them. For example, for the input of vehicle *speed*, we want to observe both its values and their changes in different circumstances. Thus, at the requirement level, to obtain abstract and concise descriptions for inputs and outputs, we need to categorize them into two classes: *flows* and *events*. A flow input (output) can be considered as a variable only whose values are to be observed; and an event input (output) represents an event which is raised when values of the input (output) are changed. In some cases, identifying and describing an event is difficult, in particular when temporal properties of inputs (outputs) are involved.

For a complicated functionality, its requirements are usually decomposed into several submodules. For example, the functionality of BCM may include sub-functionalities of locking/unlocking control, lights control and windows control etc. If a functionality is decomposed, we particularly concern about the *delegation* and *assembly* relationships between the inputs (outputs) of the top functionality and those of the sub-functionalities.

Behavior information. For the BCM application, its behavior demonstrates discrete, reactive and timing characteristics. Thus, the behavior information should includes the relation between the inputs and outputs, the collaboration among several functionalities, and some timing constraints such as timeout, deadlines etc.

Communication information. It mainly includes *communication topology*, *signals* and *protocols* which communicate between an ECU and its environments. In automotive systems, bus technologies (CAN, LIN, Flexray and MOST) are widely used to connect an ECU with its environment, and signals transmit over buses to transfer data and control commands. There exists a close relation between the communication information and the behavior information, because the functionalities are usually triggered by the received signals, and their outputs are also transmitted via signals.

Platform and nonfunctional constraints: such as mechanical and electronic constraints, predefined hardware architecture, timing and resource constraints. These information are essential for hardware design and further software allocations and performance testings.

3 Representation of Requirement Information

To represent the requirement information, we choose different modeling notations to represent them: *EAST-ADL2* for structure information, *timed automata* for behavior structure and *communication architecture* for communication information. Among these formalisms, we will put emphasis on EAST-ADL2 and communication architecture. Timed automata formalism will be introduced briefly, the interesting readers please refer to work [4] for its detail definition.

3.1 EAST-ADL2

As this paper only discusses the requirements modeling, we choose a subset of EAST-ADL2, i.e., the function modeling package of EAST-ADL2, as the modeling

language to specify structures of functionalities. Figure 1 illustrates the metamodel of this subset.

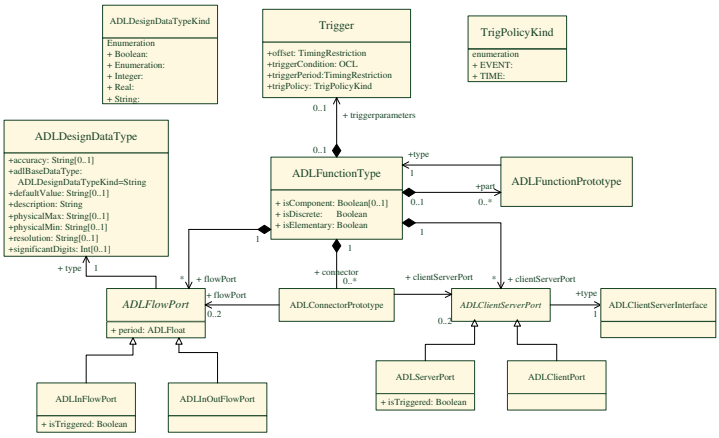

Fig. 1. Metamodel of EAST-ADL2 for the function modeling

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

`ADLInFlowPort` and `ADLServerPort` can be used to model *state* inputs and *event* inputs respectively, and the internal decomposition structure of a functionality can be modeled with function prototypes and the connectors which link function type and all its prototypes together.

Example. Figure 2 illustrates the structure model of BCM. Here, two function prototypes *lockCtr* and *lightCtr* are depicted in the function type *BCM*. The inputs of BCM are delegated to either one of the prototypes or both of them.

The structure model actually specifies the set of common phenomena shared with the functionality and its environment. To facilitate the following discussions, we divide the ports of a functionality as four sets: *inflow_ports*, *outflow_ports*, *server_ports* and *client_ports*. For the LOCKCTR functionality, its ports are of:

server_ports={RKELock, RKEUnlock, DCLLock, DCLUnlock, VehicleCrash, SpeedUp}

inflow_ports={VehicleSpeed, IgnStatus, KeyStatus, TouchStatus}

outflow_ports={LockEnable, UnlockEnable}.

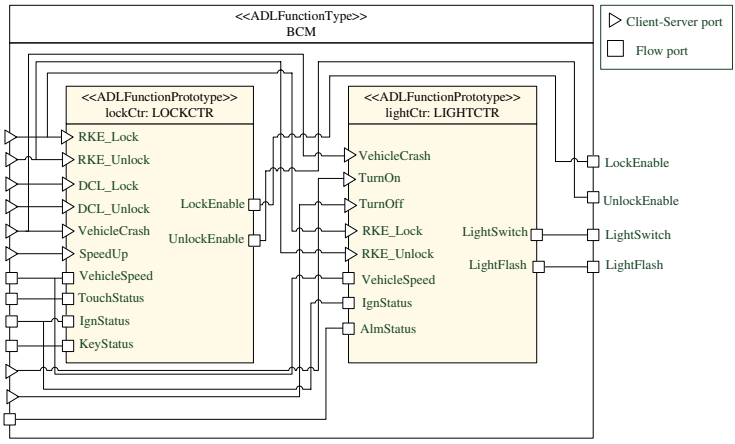

Fig. 2. The structural model of the BCM

3.2 Timed Automata

Timed automata formalism is an extension of traditional untimed automata, by introducing time clocks and timed invariants to describe timing behaviors of systems. A timed automaton interacts with its environment through channels and global variables. To simulate and verify a target timed automaton, we must additionally model its environments in terms of timed automata, and compose them parallel to form a network of timed automata.

For the simulation and verification purposes, in this paper, we choose UPPAAL version [17] of timed automata as the formalism. UPPAAL version extends pure timed automata with a number of features. One of the significant features is that the expressions are allowed to use bounded integer variables (or arrays of these types) as well as clocks, this extension will enhance the expressiveness of timed automata, allowing us to model complicated guard conditions, assignments and invariants. In what follows, if no specific explanation, the term *timed automata* is referred to the UPPAAL extension version.

Let C be a set of clocks, V a set of bounded integer variables. $\Phi(C,V)$ and $R(C,V)$ denote set of conditions and set of reset operations over C and V respectively. A timed automata is a tuple (L,B,C,V,E,I,l_0), where L is the set of locations; $l_0 \in L$ is the initial location; B is the set of channels; $E \subseteq L \times B_{?!} \times \Phi(C,V)$ $\times R(C,V)^* \times L$ is the set of edges, where $B_{?!} = \{a?|a \in B\} \cup \{a!|a \in B\} \cup \{\tau\}$ is the set of co-actions and internal action. An element $(l,\alpha,\varphi,r,l') \in E$ describes an edge from location l to l' with action α, guard φ and a list r of reset operations; and $I : L \to \Phi(C,\emptyset)$ assigns timing invariants to locations.

3.3 Communication Architecture

Communication architecture mainly depicts hardware components and their connections. Hardware components include ECUs, sensors, actuators and power suppliers; Connections usually comprise I/O connections, power connections and bus connections which connect an ECU with other ECUs or sensors (actuators). The only way

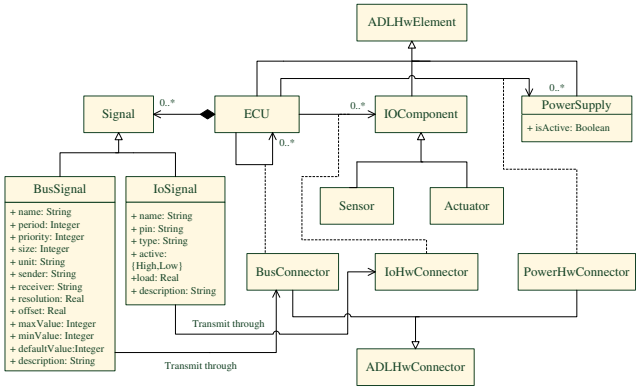

Fig. 3. The metamodel of communication architecture

that an ECU interacts with its environment is through signals, which are transmitted via physical connections. The metamodel of communication architecture is illustrated in Figure 3.

Generally, signals are divided as two categories: *BusSignal*s and *IoSignal*s. A bus signal is a string of bits, which transmits on bus through frames; An I/O signal transmits through an I/O physical wiring, its values are determined by temporal patterns of high/low electric levels.

Example 3.1 We usually use *signal matrix*, a two-dimension table, to record all the relevant signals of an ECU. To efficiently record the signals and facilitate their sharing in the tool support, we adopt XML file as the intermediate format to store the signals. The schema of the XML file should follow the signal's properties. The following XML file gives an example of signal matrix, where `VehicleSpeed_Signal` is a bus signal transmitted through CAN bus, and `DCLUnlock_Signal` is an I/O signal received via PIN D24.

```
<SignalMatrix>
<signal kind="bus signal", name="VehicleSpeed_Signal", period="10ms", priority="0x31",
size="8", unit="KPH", sender="EMS", receiver="BCM", resolution="1KPH/bit", offset="0", ...
/>
<signal kind="I/O signal", name="DCLUnlock_Signal", PIN="D24", type="Input", active="Low",
load="1-10mA", description="Central unlock switch">
.....
</SignalMatrix>
```

Bus signals always transmit periodically, they need *priorities* to avoid conflictions in transmission. The properties *resolution* and *offset* are used to transform a signal value to its actual value via the formula:

$$actual\_value = signal\_value \times resolution + offset.$$

PIN number of an I/O signal stands for the ID of the hardware interface from which the signal is received or sent. The values of a signal are represented by high/low electric levels on the physical line. For example, the "Low" electric level stands for the value "1" for the signal `DCLUnlock_Signal`, and "High" for the value "0".

We divide signals into *input_signals* and *output_signals*. For the LOCKCTR application, the input and output signals are of:

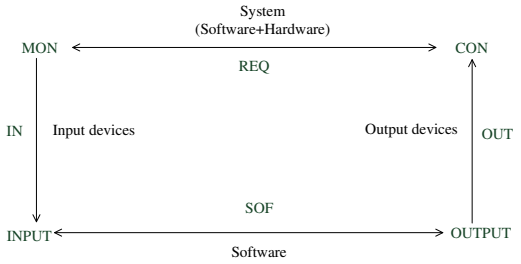

Fig. 4. 4-variable requirement model

input_signals={VehicleSpeed_Signal, InteriorTemp_Signal, BatteryVoltage_Signal, DCLUnlock_Signal, DCLLock_Signal, RKEUnlock_Signal, RKELock_Signal, KeyStatus_Signal, IgnStatus_Signal, TouchStatus_Signal, VehicleCrash_Signal}, and *output_signals*={LockEnable_Signal, UnlockEnable_Signal }.

4 Integrating Aspectual Models

In this section, we discuss how to combine these three aspectual models (structure model, behavior model and communication architecture) to form an integral system model. To this end, we must find a common framework on which the aspectual models can be explained consistently. In this paper, we choose the *4-variable requirements model*, a relational theory for requirement specifications, as the common base, and map the elements of the aspectual models to the terms of this model.

4.1 4-variable model

An overview of the 4-variable model is shown in Figure 4. According to this model, a sound requirement specification for an embedded software should be specified in terms of four groups of variables and five kinds of relations. The variables in this model are time-dependent:

- *Monitored variables* MON, the environmental quantities that influence a system's (including both hardware part and software part) behavior;
- *Controlled variables* CON, the environmental quantities that a system controls;
- *Input variables* INPUT, the boundary of the *software* that the input devices(such as sensors or hardware/software drivers) write to the software; and
- *Output variables* OUTPUT, the boundary of the *software* that the output devices(such as actuators or hardware/software drivers) read from the software.

The five mathematic relations between these variables are of:

- NAT defines nature laws or physical constraints imposed on the variables of MON and CON. For example, a reasonable range on vehicle speed is: 0~254KHP;
- REQ, a relation between MON and CON, which defines the response of the system to the values of the monitored variables by producing the values of the controlled

Table 1
Mapping the aspectual models to 4-variable model

Elements of 4-variable model	**Elements of the aspectual models**
MON	*inflow_ports*∪*server_ports*
CON	*outflow_ports*∪*client_ports*
INPUT	*input_signals*
OUTPUT	*output_signals*
NAT	Data types associated with ports
IN	Data refinement mapping: *server_ports*∪ *inflow_ports*→ *input_signals*.
OUT	Data refinement mapping: *client_ports*∪ *outflow_ports*→ *output_signals*.
REQ	The relation depicted by timed automaton model.
SOF	SOF can be derived from REQ, IN and OUT. Let TA be the timed automaton model for a functionality. SOF is a relation which can be depicted by a new timed automaton which is derived by transforming every edge (l,α,φ,r,l') of TA to a new edge (l,α',φ',r',l'), such that α'=IN(α), φ' and r' are expressions about signals, and $\varphi \wedge \text{IN} \Rightarrow \varphi'$, $r \wedge \text{OUT} \Rightarrow r'$.

variables;

- IN, a relation between MON and INPUT. It is an abstraction of the input device;
- OUT, a relation between CON and OUTPUT, which is an abstraction of the output device; and
- SOF is a relation between INPUT and OUTPUT, specifying software behavior.

4.2 Integrating Aspectual Models

The 4-variable model in fact establishes a criterion for determining whether a requirement document is complete and consistent. Therefore, in order to integrate the aspectual models to form a complete and consistent requirement model, we must compare the aspectual models with 4-variable model, that is, establish a corresponding relation between the elements of aspectual models and the terms of 4-variable models. If each elements of 4-variable model can find its counterpart in the aspectual models, then we say that the aspectual models can be integrated.

This corresponding relation is showed in Table 1. In what follows, we will explain the meaning of the mapping.

At the requirement stage, we usually identify the system (including both software and hardware) boundary by recognizing the input and output variables. Generally, these variables are of time-dependent functions, which either represent the varying of the values of data with time, or represent the triggering events. To record and manage these variables efficiently, we usually use `ADLFlowPort` and `ADLClientServerPort` (see Figure 1) to describe them. Therefore, the union of the set *inflow_ports* of the input flow ports and the set *server_ports* of the received events, in fact constitutes the set of monitored variables **MON**; and the union of the set *outflow_ports* of the output flow ports and the set *client_ports* of the sending events constitutes the set of controlled variables **CON**. The data types associated with the flow ports actually constraint the nature of data exchanged via the ports. Therefore, in some sense, they can be considered as the constraints **NAT** enforced on **MON** and **CON**.

The signals, in fact, are of the variables only handled by the software of an ECU. An ECU may sense (or act on) its environment via sensors (or actuators), physical wirings or bus networks. No matter what kinds of devices or medias are used by an ECU to communicate with its environment, from the software viewpoints, signals are the only way that an ECU sense and actuate on its environment, and all the devices and medias can be abstracted as the transformers which transform the *physical variables of the environment* (described with the ports) to the *logical variables of software* (described with the signals). Therefore, the input signals *input_signals* actually constitute the variables **INPUT**, and the output signals *output_signals* actually constitute the variables **OUTPUT**; and **IN** and **OUT** are of the data refinement relations (e.g., transformers) between the ports and the signals.

Generally, the data refinement relations **IN** and **OUT** are of simple mathematical formulas, but in some cases, they may become complicated, especially when timing is involved. We use the following two examples to illustrate the relation **IN**.

Example 4.1 The interior temperature *InteriorTemp* of the vehicle capsule is sensed by a temperature sensor, and transmitted through a signal *InteriorTemp_Signal* of CAN bus to the vehicle body control unit. The *resolution* and *offset* of *InteriorTemp_Signal* are of 0.03125°C/bit and -237. Then the relation between the port *InteriorTemp* and the signal *InteriorTemp_Signal* is

$$InteriorTemp = InteriorTemp\_Signal \times 0.03125 - 237$$

Example 4.2 The vehicle crash event is detected by identifying such a temporal pattern of the signal *Crash_Signal* on a physical wiring: *Crash_Signal* at the "Low" level 200ms followed by at the "High" level 40ms. This temporal pattern is illustrated by a timing diagram (see Figure 5) of the signal values. If we use a port $Crash \in \{0,1\}$ to denote the crash event, then the relation between the port *Crash* and the signal *Crash_Signal* can be described by the following duration calculus [8] formula:

Crash=1 iff

$(\lceil Crash\_Signal = Low \rceil \wedge \ell = 200)$;

$(\lceil Crash\_Signal = High \rceil \wedge \ell = 40)$; **true** holds.

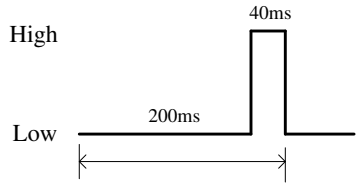

Fig. 5. Timing diagram of the signal *Crash_Signal*

The requirement relation **REQ** is actually described by the behavior specification, i.e., timed automata. There is a close relation between the structural and the behavioral specifications: the *server_ports* and *client_ports* of the structural specification constitute the channels of the timed automata; the *inflow_ports* and *outflow_ports* of the structure specification constitute the global variables or formal parameters of timed automata. Therefore the transition relation derived from the timed automata actually reflects the relation between input ports and output ports.

SOF is a relation between the input signals and the output signals, it can be derived from **REQ**, **IN** and **OUT**. The example (see Figure 6) demonstrates how to derive **SOF** from **REQ**. "Unlocked" and "Locking" are two locations of the timed automaton for **REQ**, the edge between them is the transition. The labels on the edge stand for action, guard condition and assignment respectively. **SOF** can be derived from **REQ** by transforming every edge of **REQ** to the edge of **SOF** through the refinement relations **IN** and **OUT**:

IN/OUT
SpeedUp happens iff @T(VehicleSpeed_Signal≥30) holds
IgnStatus = { ON *iff* IgnStatus_Signal=Low ; OFF *iff* IgnStatus_Signal=High
TouchStatus = { OPENED *iff* TouchStatus_Signal=Low ; CLOSED *iff* TouchStatus_Signal=High
LockEnable=LockEnable_Signal

where @T(VehicleSpeed_Signal≥30) denotes a conditioned event which can be rewritten as

$$\text{VehicleSpeed\_Signal'} \geq 30 \wedge \text{VehicleSpeed\_Signal} < 30,$$

the primed version of "VehicleSpeed_Signal" denotes its current value, and the unprimed version denotes its previous one.

5 Tool Support

This section will report our experience on the development of the tool which fulfills our basic specification methodology introduced previously. The tool has these

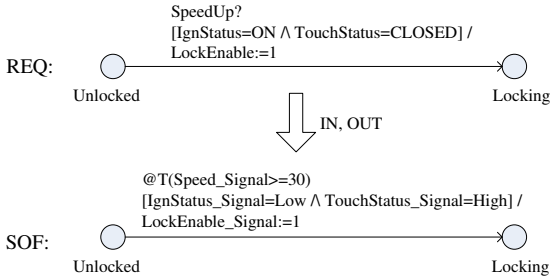

Fig. 6. Derive SOF from (REQ, IN, OUT)

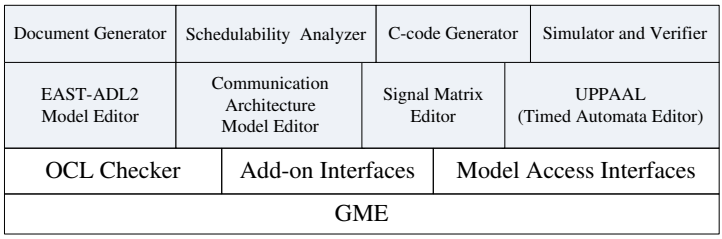

Fig. 7. Structure of the tool

abilities: (1) It allows developers to build aspectual models; (2) It supports developers to establish the linkages between aspectual models; (3) It is able to integrate some external COTS (Commercial-Off-The-Shelf) for model simulation and verification; and (4) It has the abilities of code-generation, document-generation and schedulability analysis.

5.1 Structure of the Tool

The development of this tool is based on GME (Generic Modeling Environment) [10] platform, an UML meta-modeling environment for the development of *domain specific language* for embedded systems. The main strength of GME is that it allows us to develop a modeling environment by the construction of its meta-model, which will significantly reduce the development cost and facilitates the management of changes. For example, to build an EAST-ADL2 modeling editor, we just need to build the meta-model of EAST-ADL2 (just like Figure [3]) with GME tool, define the constraints on this meta-model with OCL language, and choose favorable icons and line-styles to demonstrate the modeling elements and relations. After then, GME Meta-Interpreter can interpret the meta-model as a paradigm file (*.xmp file), with which developers can build their EAST-ADL2 models in GME environment.

The tool architecture is illustrated in Figure 7. All the modeling tools and assistant tools are built on the basis of GME environment. The uncolored blocks stand for the build-in mechanisms or interfaces provided by the GME environment, and the grey blocks for the tools developed by ourselves or integrated from external COTS. The main building blocks include:

- *OCL Checker*, *Add-on Interfaces* and *Model Access Interfaces* are build-in blocks

provided by GME. In GME, constraints are defined on the elements of the meta-model with OCL language. Developers should designate the model element (of the meta-model) where the constraint will be attached, and the time when the constraint checking is triggered. When the constraint is violated in modeling time, an alarm will arise at the triggering time; *Add-on Interfaces* provide developers a means to developing their own programs and integrating them with GME. For example, in our tool support, UPPAAL tool is integrated through these interfaces; *Model Access Interfaces* provide a set of interfaces allowing us to access the model elements through programming.

- *EAST-ADL2 Model Editor* and *Communication Architecture Model Editors* are developed with GME meta-modeling techniques just mentioned previously; *Signal Matrix Editor* is an individual program which allows us to edit, modify and record the signals of an ECU; UPPAAL tool is also loosely integrated in our toolset, it is only invoked from our tool, thus its integration does not violate the copyright of UPPAAL.
- *Document generator* generates the formatted document from models. It extracts necessary modeling information from models by using model access interfaces, and then exports them to a WORD document; *Code generator* takes timed automata as inputs, and generates segments of C-code to implement them; We have also implemented a schedulability analyzer which is able to compute the WCRT (Worst Case Response Time) [11] of signals for CAN bus. In the future, we also want to combine task schedule and bus schedule together, and integrate holistic schedulability analysis into this tool; *Simulator and Verifier* can simulate a functionality and verify its properties based on its timed automata model. In the next subsection, we will illustrate an example of simulation and verification with the help of UPPAAL tool.

At present, this tool has been used in the BCM project. This project includes a total of 8 main functions, more than 40 ports and 90 signals are involved. Obviously, it is difficult to manage such a large model if no tool support was available.

5.2 Simulation and Verification with UPPAAL

In this section, we introduce how to simulate and verify the functionalities with our tool. Since our tool borrows UPPAAL tool to do these tasks, we mainly introduce the procedure of transforming the models to an UPPAAL specification. The transformation procedure will combine the structure model and behavior model together, because the channels, global variables appeared in an UPPAAL specification are actually the ports depicted in the structure model. Therefore, we can also view the transformed UPPAAL specification as the integration of structure and behavior models.

To simulate and verify the requirements with the UPPAAL tool, we should carry out the following steps to transform the aspectual models to an UPPAAL specification:

- Transform the structure model and the behavior requirements of a functionality

to a timed automaton with UPPAAL tool;

- Model the environment of the functionality as timed automata as well;
- Combine the timed automata of both the functionality and its environment together to form an UPPAAL specification, and then perform the simulation and verification activities by using UPPAAL tool.

An UPPAAL specification includes four parts: *global declarations*, *templates* for timed automata, *process assignments* and a *system definition*. Global declarations declare clocks, data variables, channels, and constants, all of which are shared with every timed automaton in a system; templates are timed automata equipped with lists of formal parameters and with local declarations of clocks, data variables, channels, and constants; 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 running parallel.

We propose a number of transformation procedures which transform the structure model and behavior requirements to an UPPAAL specification:

Step1: Declare *server_ports* and *client_ports* of the structure model as global channels;
Step2: Declare *inflow_ports* and *outflow_ports* of the structure model as global data variables or formal parameters of templates;
Step3: Model the behavior requirements and local clocks, local data variables as templates of timed automaton;
Step4: The environments of a functionality are declared as templates of timed automata as well;
Step5: The variables and channels which are expected to be instantiated when creating a process, are declared as formal parameters of the templates;
Step6: Compose the functionality and its environment parallel with system definition.

Example 5.1 The UPPAAL specification for LOCKCTR is illustrated in Table 2 and Figure 8. The structure model and the behavior requirements for LOCKCTR are illustrated in Figure 2 and "Req1-Req5" in Section 2.

Note that, the environment of LOCKCTR may include drivers, lock actuators, and other ECUs such as EMS (Engine Management System). As an example, here we only select drivers and EMS as the environment of LOCKCTR. They initiate the actions which must be synchronized with the co-actions of LOCKCTR's automaton. Because variables "VehicleSpeed, IgnStatus, KeyStatus and TouchStatus" come from the other environment outside drivers and EMS, they are specified as the formal parameters of LOCKCTR's automaton and instantiated when creating the process.

With the help of UPPAAL tool, we can simulate and verify the UPPAAL specification. The simulation explores all possible runs of the combined system (LOCKCTR $\|$ Drivers $\|$ EMS) in a step-by-step or random manner, through which analyzers can detect potential errors in the requirement models. UPPAAL tool also allows us to verify some properties of the specification. Generally, a property may be either a very general purpose one, such as *deadlock freedom*, which must hold for any valid requirement model, or a domain-specific one which should be identified according

Table 2
UPPAAL specification for LOCKCTR

Global declaration

```
chan RKEUnlock, RKELock, DCLUnlock, DCLLock;

chan VehicleCrash, SpeedUp;

const int[0,1] OPENED=1, CLOSED=0, IN=1, OUT=0, ON=1,OFF=0;

int[0,1] UnlockEnable=0, LockEnable=0;
```

Templates

```
LOCKCTR(int[0,254] VehicleSpeed, int[0,1] IgnStatus,

    int[0,1] KeyStatus, int[0,1] TouchStatus)

Drivers();

EMS();
```

System declarations

```
const int[0,254] speed=30;

lockcontrol=LOCKCTR(speed,ON,IN,OPENED);

drivers = Drivers();

ems = EMS();

system lockcontrol, drivers, ems;
```

to the knowledge of an application domain. In this paper, we do not discuss how to verify UPPAAL specifications, and how to detect and correct possible requirement errors through verification mechanism. It is a further topic deserved to be explored in the future.

6 Related Works

This work is inspired by the works [1], [2], [5] and [6]. [1] is a platform-based design methodology for embedded systems, and developed an environment, called Metropolis, to support it. This methodology separates an embedded system as several orthogonal aspects, such as computation and communication, function and architecture, and behavior and performance parameters. Each of the aspects are represented by UML notations. Work [2,5] presents the rCOS method of model driven design of component-based software. It emphasizes on the integrated use of different modeling notations for multidimensional modeling to support separation of concerns and and incremental development. The work presented in [6] defines the different dimensions of structures, data, functionality and interaction and shows how they are represented in UML diagrams. Theoretical study of relationship between

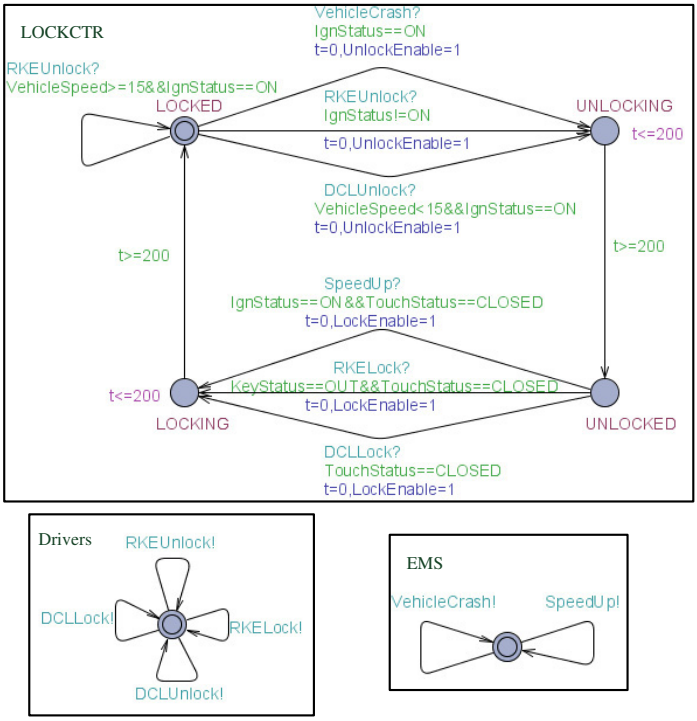

Fig. 8. Timed automata for LOCKCTR and its environments

the structure and behavior aspects in the context of object-oriented programming in [7], where the structure aspect presents the attributes and associations in a class diagram, and the behavior aspect presents the class methods. It investigated how to derive the class methods from the structure aspect by giving a small set of structure refinement rules.

The main ideas of this paper derive from these works, however in this paper, we try to apply them to the domain of automotive software, and try to make them practical for actual development. Domain characteristics and application background make us to choose different modeling notations for automotive software. For example, we choose EAST-ADL2 to describe the structure of software, because on one side EAST-ADL2 is an industrial standard which has been widely recognized by engineers, and on the other side, it is restrictive and expressive enough to present the expected structure information.

SaveCCM [12] and AML (Automotive UML) [14] are works similar to this paper. Like the basic ideas represented in this paper, SaveCCM also separates software into structure and behavior aspects. Its behavior model is specified with *task automata* formalism [13], an extension of timed automata with tasks. Note that, SaveCCM is a model for software *design purpose* rather than requirements analysis, because task automata are mainly used to perform schedulability analysis of tasks, however, there is no tasks have been identified yet at the requirement level. The main weakness of SaveCCM is that it has no notations for describing communication architecture of an automotive system. AML is a modeling language tailored to the development needs

of automotive embedded systems. It provides the abstraction levels and necessary modeling elements for automotive applications, such as function, function variant and function networks. The main weakness of AML is that it does not provide the notation for behavior modeling.

For complicated software, a single formalism usually does not work for all system aspects, therefore many works dedicate on the *integrated formal methods*, which combine several well-built formal methods together to specify the whole system. CSP-OZ [15] is a method which integrates both CSP and Object-Z formalisms. CSP is used to model the behavior aspect and Object-Z for data structures and state-based specifications. To model the time-related behavior, CSP-OZ-DC [16] formalism is proposed, which additionally integrates duration calculus formulas to specify the timing properties of systems. In this paper, the timing behavior is modeled by both timed automata and duration calculus formulas. The former is used to model the top-level operational (reactive) behavior of software, and the latter is used to specify the temporal changes of signal values.

7 Conclusion and Future Works

In this paper, we explored how to extract models from requirement documents to support model-driven approach for automotive software. We also proposed an approach to linking these aspectual models together to form a complete requirement specification. In the future, we want to continue this work in two directions: firstly, we want to extend our framework to include additional modeling concepts and performance characteristics, such as function variants, QoS, safety and reliability, resource and scheduling properties. With these performance parameters, we can deal with perform analysis for the models; the second work is to extend the current tool to support model simulation, holistic schedulability analysis and performance analysis.

References

[1] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli and J. Rabaey. UML and Platform-based Design. UML for Real, 107-126, Kluwer Academic Publishers, 2003.

[2] Zhiming Liu, Charles Morisset and Volker Stolz. rCOS: Theory and Tool for Component-Based Model Driven Development. In F. Arbab and M. Sirjani, editors, International Symposium on Fundamentals of Software Engineering (FSEN 2009), volume 5961 of Lecture Notes in Computer Science, pages 62-80. Springer, 2010.

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

[4] R. Alur and D. L. Dill. A theory of timed automata. Theorectic Computer Science. 126(2):183-235, 1994.

[5] X. Chen, J. He, Z. Liu, and N. Zhan. A model of component-based programming. In F. Arbab and M. Sirjani, editors, International Symposium on Fundamentals of Software Engineering (FSEN 2007), volume 4767 of Lecture Notes in Computer Science, pages 191-206. Springer, April 2007.

[6] X. Chen, Z. Liu, and V. Mencl. Separation of concerns and consistent integration in requirements modelling. In J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack, and F. Plasil, editors, SOFSEM 2007: 33rd Conference on Current Trends in Theory and Practice of Computer Science, volume 4362 of Lecture Notes in Computer Science, pages 819-831. Springer, January 2007.

[7] L. Zhao, X. Liu, Z. Liu, and Z. Qiu. Graph transformations for object-oriented refinement. Formal Aspect of Computing, 21(1), Springer, 2009.

[8] Zhou Chaochen and M.R.Hansen. Duration Calculus: A Formal Approach to Real-Time Systems. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[9] D. L. Parnas and J. Madey. Functional documentation for computer systems. Science of Computer Program. 25(1):41-61, 1995.

[10] G. Karsai, J. Sztipanovits, A. Ledeczi and T. Bapty. Model-Integrated Development of Embedded System. Proceedings of the IEEE. 91(1): 145-164, 2003.

[11] Davis, R. I., A. Burns, R. J. Bril and J. J. Lukkien (2007). Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. Real-Time Syst. 35, 239-272.

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

[13] 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.

[14] Michael von der Beeck, Peter Braun, Martin Rappl, and Christian Schröder. Automotive Software Development: A Model-Based Approach. SAE Technical Paper Series 2002-01–875,Detroit, 2002.

[15] G.Smith. The Object-Z Specification Language. Kluwer Academic Publishers, 2000.

[16] J.Hoenicke. Combination of Processes, Data, and Time. PhD thesis, University of Oldenburg, Germany, 2006.

[17] G. Behrmann, J. Bengtsson, A. David, K.G. Larsen, P. Pettersson and Wang Yi. Uppaal implementation secrets. In Proc. of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, 2002.