Design of ATP (Auto Train Protection) Software Architecture Based on Hierarchical Component Model

Xin Bao¹, Wei Dong², Yindong Ji² and Xinya Sun²
1 Department of Automation, Tsinghua University, Beijing, China, 100084
Email: celeeinago@hotmail.com
2 Tsinghua National Laboratory for Information Science and Technology, Tsinghua University
Beijing, China, 100084
Email: weidong@tsinghua.edu.cn, jyd@tsinghua.edu.cn, xinyasun@tsinghua.edu.cn

Abstract—In this paper, by introducing hierarchical component model into design of ATP (Auto Train Protection) software architecture, an applicable railway signal system software modeling scheme is proposed. For detail discussion, the design of software hierarchical architecture, hierarchical component model, dynamic component insert mechanism and fault injection mechanism have been also presented. As result, a simulation model of ATP in a railway signal simulation system is established by using the methodology proposed in this paper, to verify its feasibility and advantages in real software development.

Index Terms—hierarchical component model, message bus, ATP, fault injection

I. INTRODUCTION

With the rapid development of high speed railway, more and more chances and challenges have been brought to the research and development of railway signal systems, as well as the design, simulation and test of relevant software systems. ATP (Auto Train Protection) equipment plays an important role in safety issue of modern railway system, whose functions, algorithms and communication implements vary dramatically with the change of application environment. As a result, the improvement of maintainability and reusability of ATP Software becomes a new challenging field. Although traditional design methodology of ATP software, which highly depends on platform, process and codes, is applicable due to its application environment, however, its complexity of software maintenance, update and re-development becomes great obstacle of its functional evolution.

The methodology of software component technique is derived from hardware component technique which has been used for years. Software component is a set of encapsulated, reusable codes. According to the interface standards of components, anyone can implement them using any approaches he wants the user of components can easily make use of the services provided by components regardless the implement detail of component, abiding to the interface standards; strictly following the interface standards, any independent updates made by components or users would not bring problem of incompatibility [1-3]. To standardize and formalize the architecture of ATP software, component technique may be introduced into the design process of ATP software, so to fit the features of multi-platform, multi-standard of ATP system, as well as to raise the reusability and reliability of ATP software and minimize the cost and time of development.

In this paper, a design scheme of ATP software architecture based on hierarchical component model has been presented. As a verification of proposed scheme, an ATP simulation model based on a railway signal simulation system is developed, which shows great advantages in developing and re-using process. As the original target, this scheme also can be used in development of real ATP software to enhance development effect and software reusability.

II. ATP SOFTWARE COMPONENT MODEL BASED ON HIERARCHICAL MESSAGE BUS [4]

The most serious disadvantage of traditional ATP software structure is the strong coupling among function units. These units connect to each other by directly calling communication methods embedded in every unit, which may provide a poor solution to software structure; but such tightly coupled software architecture, may become obstacle to further maintenance, update and reuse of ATP software. In order to eliminate the strong coupling relationship among function units and build functional independent components, the software design model of message bus, which suggests connecting every independent component to a specified message-dispatcher to exchange messages with generic message-transceivers, may be adopted. In this scheme, every component works independently, and they share data with message-dispatcher, so that the degree of coupling between components can be greatly reduced. Fig. 1 has illustrated the overall architecture of ATP software model based on hierarchical message bus, and the detail of Fig. 1 will be discussed in following sections.

A. The Division of ATP Components

Analyzing main functions of ATP system according to Ref. [5-8], 4 main function components can be divided from the ATP application:
(1) Information processing component: collecting and releasing data from/to all kinds of outer data source, such as balise messages, track circuit code and braking command, etc;

(2) Mode managing component: managing the mode switching logic, according to relevant track data and operation command;

(3) Profile computing component: computing profile using specified algorithm(s), such as fixed profile, multi-braking-stage profile and single-braking-stage profile, etc;

(4) Speed supervising component: supervising train speed in real time, with reference to SP (Supervising Profile) and speed supervising rule under different application environments and standards.

B. Stack Structure of ATP Component Model

For the decoupling of ATP components and software structure, a protocol-stack-like hierarchical structure is used to illustrate ATP component model, as shown in Fig. 2. The model is divided into 4 layers. The top layer is ATP function layer, which only deals with implement of ATP application. The layer beneath ATP function layer is component structure layer, which provides the basic services of component, such as message subscribing and component registering, etc. Message-dispatcher and the generic framework of component all belong to this layer.

The third layer from the top is transceiver layer, which deals with the implement of connection between components. This layer provides communication service to component structure layer, so that component structure layer may not concern with the detail of connection and send/receive messages freely. The bottom layer is supporting environment layer. This layer is not a part of ATP software, and it presents the software runtime environments.

Each layer connects to other layers in different ways. ATP function layer is embedded into component structure layer in the form of implement codes in order to call service methods provided by component structure layer. Component structure layer uses reference of transceiver to access communication services of transceiver layer. Transceiver layer may implement communication by utilizing any available approach of supporting environment layer, such as function call, network and system message, etc. Each layer provides independent functions, i.e. communication link, software structure and ATP functions, so that developers can maintain or update content of each layer independently, which could provide a convenient interface for software evolution.

C. Message Transceiver

Transceivers provide communication services to components. These transceivers are encapsulated uniformly to embed into components, and their details of communication implements are transparent to components. So that each component may not care the link type of transceiver, and the only thing they need to do is call the interface methods of sending/receiving messages provided by transceivers. The basic requirements of bus-like communication, transceivers need to provide interface of sending message and receiving message, etc, as well as a message queue to cache messages sent by other components. The following Java codes are used to define the interface form of abstract class “Trscv” for transceivers:

```java
public abstract class Trscv {
    /** send message using this transceiver, for component calling */
    public abstract void send(Message p_msg);
    /** get a received message from message queue, return null if queue is empty */
    public abstract Message get();
    /** send message to this transceiver, whose essential is to push proposed message into the tail of message queue, for neighbor */
}
```
transceiver */
public void transfer(Message p_msg)  
{  
m_buf.offer(p_msg);
}

/** message queue used to cache received messages */
protected Queue<Message> m_buf;

The design of embedding transceivers into components enables the division of component functions and communication methods, as well as the variety of transceiver implements. For components with strong connection and frequently data exchanging, such as those supplying ATP application functions, the transceivers based on function call are recommended, for their high runtime performance; on the other hand, for external components, such as diagnosis component and fault-inject component, it is better to use transceivers based on inter-process communication, such as pipes and sockets, etc., to enable distributed testing and system analyzing conveniently. The implement details of transceivers based on function call and network are presented as follows.

(1) Transceiver based on function call

This transceiver is used to transform tightly coupled structure into loose-coupling structure based on function call. It keeps a reference of transceiver of neighbor component, so that it can call the public interface method to send messages. The essential of this public method is to push messages into the tail of the neighbor’s message queue; by encapsulating technique, the detail of message queue in neighbor component can be hided, as well as other structure details of transceivers. Strictly speaking, transceivers based on reference and function call cannot eliminate the coupling between components thoroughly, however, because the components cannot see the implements of messages delivery, so this kind of transceivers can also help to do some decoupling work to optimize software structure; what’s more, using method call to transfer messages is much more effective comparing to network or system messages, so that it is ideal for the link between components with frequently data exchange. As the disadvantage of this transceiver, it needs transferring a reference of neighbor transceiver on initializing stage to set local link between transceivers, which restrict itself only can be used in components link with relatively stable structure. Fig. 3 illustrates the principle of this kind of transceivers. Methods of components which need data exchange can call the message sending method provided by transceivers to send data; by function call between transceivers, messages are pushed into the tail of the neighbor’s message queue; the neighbor component will periodically check new messages by calling the message receiving method provided by transceivers.

(2) Transceiver based on network

As the limit of real ATP hardware platform, this kind of transceiver is mainly used in ATP simulation software. Adopting communication method based on network link, it can decrease the degree of coupling further; and due to network-based interface, it can introduce new component to software without modifying the structure of software, which dramatically increases the extendibility of software. There are two kinds of implements of network transceivers: server transceiver and client transceiver. Server transceiver listens to configured IP address and port, while client connect to server transceiver according to configured IP address and port. In order to avoid block of program, each network transceiver uses an independent thread to keep network communication. Furthermore, network link is based on data stream, so the content of messages should be processed using serialization operation before sending. By embedding this kind of transceivers, components are no longer restricted in one process, which means some components may enter the running program dynamically; nevertheless, as network links bring along problems of higher resource cost and lower performance than inner-process communication, this kind of transceivers is not suitable for circumstance of close-related or frequently data exchange. Fig. 4 presents the principle of this kind of transceivers.

In order to establish a mechanism of automatic network configuration, the transceivers should comply with link-setup rule as follows. In the situation of local loopback network communication, server transceiver opens a default port to listen to the connection request from other components, while client transceiver connects to this port by default. When the connection is established, the component holding server transceiver re-generates a server transceiver using a new port, and then sends this port number to client. Receiving new connecting detail, client releases the former link, and re-connects to new address, so that the communication link can be established. When disconnecting, client send a request of
releasing link. Server disconnects the link when receiving the request, and then the component holding this server will release this server transceiver.

D. Message Dispatcher

Message dispatcher is the core component of message bus structure, whose responsibility is to receive messages from all components, and to deliver messages by switch table and subscribing table. Fig. 1 presents the structure of message dispatcher. Dispatcher connects to other components using a set of transceivers. When receiving messages, dispatcher will classify messages by type: system messages such as subscribing request and register request will be dealt locally, other messages will be sent according to the reference of relevant transceiver registered in subscribing table.

The default delivery strategy of dispatcher is to deliver message using message-type based subscribing information, for most components only concern the content of messages, not the publishers. On the other hand, the message types from a certain component is fixed, such as train speed message can only be given by speedometer component and speed limit message is only from profile computing component, so that the publisher of a certain type of messages can be concluded from the type itself; as a result, even if a component concerns the publishers of messages, it can deduce this information from message type. However, considering some publishers may send a specified message to a specified publisher, in this circumstance, the dispatcher would deliver message according to its destination address, avoiding receiving of this message by unwanted component.

Switch table is the basic components of message dispatcher for implementing data exchange between components. Dispatcher can obtain the reference connected to relevant component by checking the information of switch table, and then deliver messages. Binary tuple (i, r) is used to describe records of switch table, where i presents the identity of component, r presents the reference of transceiver connected to this component. When delivering message, dispatcher will search the record of relevant component identity, and then use the reference of the corresponding transceiver to send message. Similarly, the record form of subscribing table is binary tuple (t, r), where t is the type of message and r is a list storing references of transceivers connecting to the components which have subscribed this type of message. When subscribing message, a component will send a subscribing request to dispatcher; dispatcher will search relevant record in subscribing table according to the message type that the component want to subscribe; finally insert the reference recorded in switch table into the corresponding reference list.

Considering the variety of message types used in ATP application, a hash-table-like list array is used to store subscribing information, in order to enhance the efficiency of searching record in subscribing table. As for ATP systems, the message type won’t be modified while system running; as a result, enumeration can be considered to present message type, so that the ordinal number of enumeration can be defined as the subscript of array. Taking Java platform for example, first an enumeration of message type is defined, which contains all known message type in it. In runtime, the program will initialize the record number of subscribing table with regards to the total records in type enumeration. When sending message, dispatcher take ordinal number of message type enumeration as the index, so to find relevant record to send message.

E. Hierarchical Message Bus of Information Processing Component

One of the main differences of different ATP implements, is the difference of the track/operating data source and contents. In order to increase the reusability of different data processing units, hierarchical message bus structure is introduced to design information processing component [4, 8]. Data processing units inside this component are defined as communication plug-ins, and outside these plug-ins, a sub-dispatcher is set to manage the data exchange between those plug-ins. The sub-dispatcher’s structure is similar to the top-level dispatcher; the difference is that it uses a specified transceiver to connect with the top-level dispatcher.

As illustrated in Fig. 1, the external characteristic of information processing component which adopts hierarchical message bus is the same to other components: they all use a transceiver to connect with the top-level dispatcher. So in the sight of dispatcher, information processing component is a common component. In sub-dispatcher, there are switch table and subscribing table similar to the top-level dispatcher, and it provides similar data exchanging service to plug-ins. Messages sent by plug-ins will be forwarded to top level dispatcher by sub-dispatcher; subscribing request from plug-ins will first be registered by sub-dispatcher, then sub-dispatcher will send a new subscribing request with itself as the subscriber to the top-level dispatcher. Through this double subscribing, outer messages will first be received by sub-dispatcher, and then the sub-dispatcher will deliver messages according to its subscribing table.

F. Dynamic Component Insert Mechanism

For response to request of inserting a component in runtime dynamically, dynamic component insert mechanism is introduced [4]. This can be realized by using network transceivers. As mentioned in section 3.3, following the connecting rule of network transceivers in dispatcher, a listening server should be set to listen to register request. When receiving register request from external components, dispatcher will generate a network transceiver to establish a communication link; when external components releasing link, dispatcher releasing the relevant transceivers immediately.

G. Fault injection Mechanism

Faults often happen in the internal area of components, though the behaviors of fault always happen at the interface of components. Usually the relationships between interfaces and faults’ behaviors can be classified into two ways: (1) faults behaviors stimulated by
specified correct input, and (2) fault outputs stimulated by faults of component. According to this, fault injection mechanism can be concluded in two aspects: (1) capturing common output of a certain component, and substituting it with modified or specified data, to simulate the circumstance of output of component with fault; (2) capturing common input of a certain component, and substituting it with modified or specified data, to stimulate the fault mode of the component. Therefore, a certain component used to capture common data and inject fault data according to certain rules is needed. Because of the variety of fault behaviors and fault stimulations, the component should supply an approach to describe any behaviors of fault input/output data. What’s more, in order to support message capturing function, the dispatcher should provide service of fault-oriented delivery, i.e. delivering certain messages to fault-inject component, and taking modified messages to deliver to common components.

In order to implement fault injection based on capturing/modifying messages, a specified fault-inject component and fault-inject subscribing table of dispatcher should be considered. The fault-inject component mainly provides 2 services: one is capturing messages, the other is publishing modified or specified data according to fault-inject rules. The rules are described in fault-inject script, which uses a quaternion group to describe a certain fault behavior:

<Message Type, Operation, Time, Duration>,
where Message Type represents the message type needed to be captured; Operation represents actions after capturing messages, such as shielding, modifying messages and sending specified messages; Time is the beginning time point of a certain fault behavior or stimulation; Duration represents the lasting time of a certain fault behavior or stimulation. For common fault, such as failure of certain component resulting in no output, the script can be described to capture the output of that component; for output with error, such as the disturbance of speedometer communication plug-in, the script can be described to capture, modify and then publish the output of speedometer; for fault with complex behavior series, a set of several scripts can be used to combine the whole behavior of fault.

The structure of fault-inject subscribing table is the same to common subscribing table, though message dispatcher deals with them differently. First, the priority of fault-inject subscribing table is higher than common subscribing table. When dispatcher receiving messages, it first searches fault-inject subscribing table. If there are records of this message type, dispatcher will ignore the common table, and deliver the messages only according to fault-inject subscribing table; if not, dispatcher will deliver the messages according to common subscribing table. Second, dispatcher will filter messages that have been fault-inject subscribed according to the source address of messages, i.e. if messages are from common components, dispatcher will send them directly to fault-inject subscriber; if they are from fault-inject subscriber, dispatcher will send them normally according to common subscribing table.

A typical fault injection process based on message modifying is as follows. At first, fault-inject component subscribes a certain message type to fault-inject subscribing table according to fault-inject script. The dispatcher receives the subscribing request of fault injection, and insert a record in fault-inject subscribing table accordingly. When messages from other components passing through, dispatcher will first see if they are from common component, and if the type is subscribed by fault-inject subscribing table. If there are not relevant fault-inject record, dispatcher will deliver it according to common subscribing table; else, dispatcher will deliver it directly to fault-inject component. The fault-inject component will deal with captured messages according to fault-inject script: if script requires no output of certain component, fault-inject component will release the message directly without replying; if script requires modified or specified content of message, fault-inject component will process the content of message, and send it back to dispatcher. When receiving modified messages, dispatcher will deliver them normally according to common subscribing table, for the source of messages is fault-inject component. Therefore, the behavior, as well as the stimulation of certain fault, in the form of specified input/output message series, can be controlled to further system simulation and test.

III. RESULTS

As verification of proposed design scheme of ATP software, an ATP simulation model is developed using the methodology presented in this paper, based on an existing railway signal simulation platform. Adopting HLA technique, the implement of simulation platform is independent from that of ATP simulation model; as support environment, the simulation platform can provide all external data demanded by runtime of ATP model, such as balise messages and train speed, etc; ATP model can output brake orders by utilizing the interfaces of simulation platform. This ATP model, implemented with scheme presented in this paper, can change control logics and algorithms of different standards easily to adapt different application requirements, without modifying software architecture; furthermore, by using data observer component and fault injection component, it is convenient to observe ATP operation status, as well as to apply test of different purpose.

IV. CONCLUSIONS

As important development of modern software engineering methodology, the component technique boosts the reusability and agility of codes dramatically. In this paper, we propose a scheme of ATP software modeling methodology based on component technology, which is utilized and succeeded in a railway signal simulation system. As the original intention, it is also available in development of real ATP software, as well as software for other railway signal equipment.
For next step, MDA technique will be introduce into ATP software as attempt, to experiment the possibility of ATP software development based on model transforming. By using MDA technique, the component model of ATP software will be more powerful to enhance the efficiency of development as well as to decrease costs. What’s more, it will set evidence to swift development of railway signal software.

ACKNOWLEDGMENT

This work was supported in part by the Research Foundation of the Ministry of Railways and Tsinghai University (RFMOR & THU) under Grant T200705 and the National Key Technology R&D Program under Grant 2009BAG12A08.

REFERENCES


