Applying SOFL to a Railway Interlocking System in Industry

Juan Luo[1], Shaoying Liu[2(✉)], Yanqin Wang[1], and Tingliang Zhou[1]

[1] Casco Signal Ltd., R&D Center, Shanghai, China
{luojuan, wangyanqin, zhoutingliang}@casco.com.cn
[2] Department of Computer Science, Hosei University, Tokyo, Japan
sliu@hosei.ac.jp

Abstract. This paper describes another application of the SOFL three-step specification approach in specifying a railway interlocking system in industrial setting. We also explore the way of deriving hazard conditions from formal specifications, and propose a way to analyze the conditions for the assurance of the safety of the interlocking system in the early stage of the development. Our experience shows that SOFL is much more accessible by ordinary practitioners than other existing well-known formal methods and effective in helping practitioners deepen their understanding of the system details.

Keywords: Formal specification · Hazard condition · Analysis · Interlocking system

1 Introduction

Railway signaling system is a kind of safety critical system whose failure is likely to cause catastrophic disaster. The reliability and safety of such a system can be achieved not only through the redundant architecture of hardware, but also the high quality of the software deployed for the control purpose in the system. High quality software must function as expected and must not trigger safety problems for the system.

To ensure the high quality for a software system, capturing correct and complete requirements is essential, simply because it is almost impossible to achieve a high quality implementation from incorrect or incomplete requirements. Traditional requirements analysis, design, and testing methods based on natural language descriptions can hardly guarantee that all functional and safety requirements are implemented correctly. In the industrial practice, system functional requirements are mainly documented in natural language and their implementation is verified by testing. Safety requirements are usually ensured by first using hazard log to record potential hazard and then carrying out hazard analysis in different development phases. However, this kind of practice suffers from the following two disadvantages:

(1) Requirements specifications in natural language are likely to cause ambiguity in design and implementation, which may lead to significant errors.

© Springer International Publishing AG 2017
S. Liu et al. (Eds.): SOFL+MSVL 2016, LNCS 10189, pp. 160–177, 2017.
DOI: 10.1007/978-3-319-57708-1_10

(2) Since test cases of traditional testing methods are mainly generated manually, the functional scenarios of an operation may not be considered completely, which is likely to result in the incompleteness of test case design.

It is well recognized that the later the faults are found, the higher the cost for removing the faults will become [1]. This is especially true of railway signaling systems that involve complex operations in both hardware and software.

To detect faults in requirements, especially those related to human decisions on both functional and safety requirements, formal methods are considered to be an effective technique [2]. Formal methods are built on strict mathematical definitions and have precise mathematical semantics. This advantage can help resolve requirements and property ambiguity in natural language descriptions. There are many well-known formal methods, such as VDM [3], Z [4], Event-B [5], SCADE [6], and SOFL [7], and each has its own characteristics. Although they share some common features, such as using the concepts of pre- and post-conditions in specifications, their differences in syntax, style, and requiring different level of mathematical skills provide different accessibility to practitioners, which help them make appropriate choices in practice.

We have been making all kinds of attempts to use several formal methods on our products in CASCO Shanghai. For example, we applied SCADE to the design of a zone controller subsystem, Event-B for modeling and verification of the zone control subsystem, and formal proof for verifying the interlocking system. After these attempts, we derive the following conclusions based on our experience:

(1) SCADE performs well for system design, but when it comes to requirements analysis phase, it becomes unsuitable due to the lack of effective mechanism for functional abstraction.
(2) Event-B can be used throughout the entire development process. The formal refinement adopted in Event-B is an ideal technique that integrates formal verification and design into refinement laws for developing correct programs, but since it requires too much mathematical knowledge and manipulation skills for the developers, our experience suggests that it is beyond our capability and not cost-effective as well.
(3) There are also some formal verification tools (e.g. Gatel and Prover iLock) that can be used to verify the safety and functional requirements, but they do not provide specific guidelines for carrying out formal modeling and formal verification of related properties. They do not seem to be able to guarantee the correctness of the system either, even if the verification is successfully done.

Due to the disadvantages above, we turn to SOFL. SOFL, standing for Structured Object-Oriented Formal Language, provides a formal engineering method for practical formal modeling and verification. In particular, the practicality of the formal modeling mainly comes from the SOFL three-step approach that emphasizes the importance of writing a formal specification based on the construction of an informal specification

and a semi-formal specification. After about fourteen hours training, we realized that SOFL is easy to understand and to use; it also requires much less mathematical skills than Event-B. We therefore decided to apply it to the interlocking system specification and verification as a trial testing project.

Our major contributions in this paper are three fold. Firstly, we explain how practitioners with little experience of SOFL can use the SOFL three-step approach properly to writing formal specifications on the basis of first writing informal and then semi-formal specifications. We chose the interlocking system as the target for specification and discuss how the domain knowledge can be effectively utilized to formalize properly the requirements with different features. Secondly, we describe how hazard conditions can be systematically extracted from a formal requirements specification. A hazard condition is a logical formula whose implementation may cause hazards to the system. Finally, we present a testing-based verification method for analyzing the hazard conditions.

The rest of the paper is organized as follows. Section 2 briefly introduces the interlocking system model to pave the way for readers to understand the subsequent sections. Section 2 focuses on the construction of the informal, semi-formal, and formal specifications of the interlocking system. Section 3 describes how hazard conditions can be extracted from formal specifications. Section 4 discusses our experience of using SOFL and the interesting problems encountered during the application. Section 5 briefly introduces some related applications of formal methods to interlocking systems. Finally, in Sect. 6, we conclude the paper and point out future research directions.

2 Specification for Interlocking System

In this section, we first give a brief introduction to the interlocking system used in our project, and then describe how the formal specification for its functional requirements can be constructed based on an informal specification and a semi-formal specification.

2.1 Introduction of the Interlocking System

In railway signaling system, interlocking subsystem (calls CBI, Computer based interlocking) is a signal control system that completes interactive interlocking check between signal, switch and route to set routes for trains and to prevent conflicting movements of trains. Once the route is set and the other routes conflict with the set route, they are not allowed to set and the associated interlocking operations, such as point move, are not allowed to perform. CBI should be designed to make it impossible to display a dangerous status for signal in any case and to prevent from the mistakenly release of route to ensure the safety of train operations. Only when they satisfy required interlocking relations, are trains to be allowed to proceed to the planned route in order

to prevent accidents or hazards, such as head-on collision, side collision, rear-end collision, inappropriate route entering, switch splitting, or trains derailing during operation. Since interlocking systems are safety-critical and must have safety integrity (meaning the likelihood of a system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time), according to the European standard EN50129, the safety integrity level of interlocking system is defined as SIL4. Safety Integrity Level SIL of a function is determined by the Tolerable Hazard Rate THR per hour. If 10-9 $\leq$ THR < 10-8, then the SIL of the function is defined as SIL4, which is a number indicating the required degree of confidence that a system meets its specified safety functions with respect to systematic failures.

As Fig. 1 shows, the interlocking system used in our project is divided into three layers: *man-machine session layer*, *interlocking computation layer*, and *execution layer*. Each layer is divided into several functional modules according to the partition of the functions. The man-machine session layer is responsible for processing the man-machine interface information by means of three modules, *man-machine interface module, communication module*, and *information indication module*. The interlocking computation layer carries out the interlocking computing through a dispatching module or real time operating system and a group of other modules, such as *basic interlocking module*, *self-diagnosing module*, *special interlocking module*, and *adjacent interlocking system interface module*. The execution layer controls the output of commands to the field devices through the *field device state input module* and the *field device control command output module*.

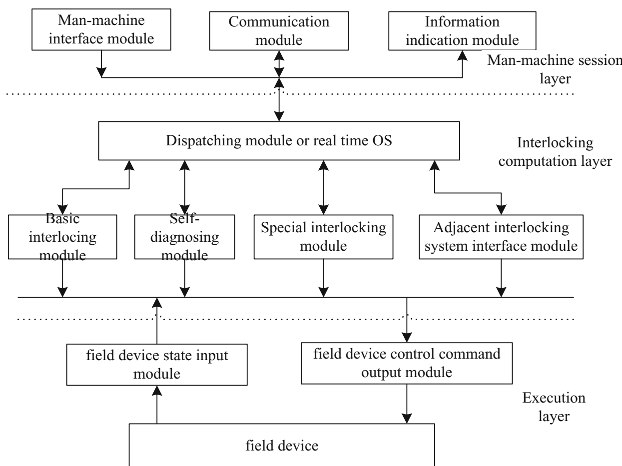

Fig. 1. The structure of an interlocking system

2.2 Basic Interlocking Function

We use SOFL mainly for the basic interlocking model that is used to realize the interlocking relations in the system. The devices controlled are mainly signals, switches, and track circuits, and these devices are controlled in a route or individually. Figure 2 is an example of part of some railway station layout that illustrates how field devices are arranged and related with each other in the interlocking system.

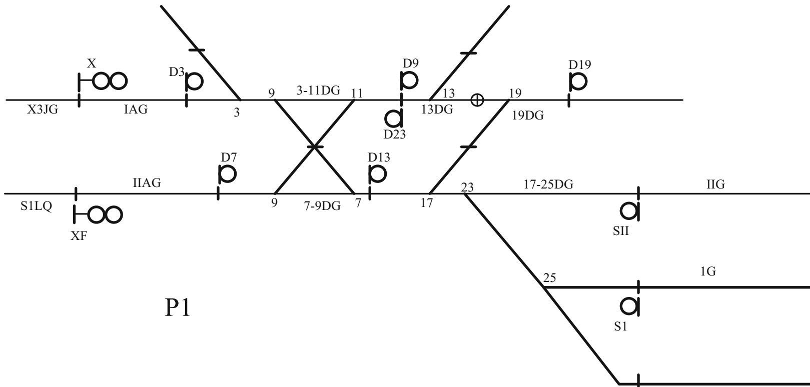

Fig. 2. Station layout example

The basic interlocking function is route controlling, including *route setting*, *route locking*, *signal opening*, *keeping signal opening*, *normal route release*, *abnormal route release*, *manual switch operation* and *general route call-on locking*. Since there are different kinds of routes, such as train route, shunting route, calling-on route, successive route, and special shunting route, and each function has different requirements for each kind of route, we need to first specify the functionality of operations for each kind of route and then investigate how the related specifications are connected to form the whole specification for the entire system.

2.3 Specific Ways to Write SOFL Specifications

As mentioned previously, the final formal specifications of various operations are achieved by means of writing an informal specification first and then refining it into a semi-formal specification, and finally formalizing the semi-formal specification into a formal specification.

2.3.1 Informal Specification

We build the informal interlocking requirements specification as advocated by the SOFL three-step approach. In this section, we focus our discussion on how the informal specification is written. According to the SOFL approach, an informal specification is composed of three sections: *functions*, *data resources*, and *constraints*.

The basic interlocking functional requirements are mainly learnt from the informal interlocking technical descriptions of the controlled devices, system states that need checking, and properties or constraints the system must satisfy. According to the form of SOFL informal specification, we treat the operations for checking the system states as bottom level functions, the devices (e.g., routes, signals, switches) to be controlled by the system as data resources, and the properties that the system must satisfy as constraints. For the sake of both confidentiality of the original specification and space limit, we only give the informal specification of a *switch normal operation* below as an example to show the general structure of an informal specification.

Informal specification for the switch normal operation:

1. ***Functions:***
   1.1 *switch operation*
      1.1.1 *switch normal operation*
         1.1.1.1 *check that the switch is not locked*
         1.1.1.2 *check that the switch has position indication*
         1.1.1.3 *check that there is no reverse operation command output*
         1.1.1.4 *check that time is not out for the switch to operate*
      1.1.2 *switch reverse operation*
2. ***Data resources***:
   2.1 *switch*
   2.2 *route*
3. ***Constraints***:
   3.1 *If the switch is already in normal position when receiving a normal operation request, then the system will not output the normal operation command.*
   3.2 *If the max time for switch operation is expired, the operation for switch move must be stopped.*

In this informal specification, the description of each item is deliberately kept short and its style is not restrictive. However, to make the specification comprehensible, each functional description uses the *verb-object* structure; each data item is described using a noun; and each constraint is presented as a condition. The application of this principle can be flexible for other domains in practice.

2.3.2 Semi-formal Specification

After finishing the informal specification, we refine and transform it into a semi-formal specification. At this step, three things are done to fulfill the task. Firstly, we group the related functions, data resource items, and constraints in the informal specification into SOFL modules. Secondly, we declare all of the necessary constant identifiers, type identifiers, and state variables formally in SOFL. Finally, we define the functionality of each process in the module using pre- and post-conditions properly.

As far as constructing each module is concerned, we take the following guideline to

define the corresponding items in the module. Each function in the informal specification is refined into a process in the SOFL module because each process fulfills a function by defining how its input can be used to produce its output. Each data resource item in the informal specification is transformed into a state variable declaration because it is likely to be shared by several processes. Each constraint in the informal specification is refined into either an invariant or part of some process functionality in the module, considering its role in the system.

For transforming the data resource items in the informal specification to the declarations in the semi-formal specification module, we apply the following principle. For each data resource item, we declare a state variable using a well-defined type in the module. If the type is not defined yet using the SOFL notation, we need to declare it properly in the section named "type" of the same module. For each declared type, its constraints, if any, can be defined as invariants in the section named "inv" of the same module. Each invariant is a condition described in natural language in the semi-formal specification. For each state variable, its properties that must be sustained throughout the entire system can also be defined as invariants in the "inv" section in the similar way to type invariants.

As far as refining each function in the informal specification into a process in the module is concerned, we use pre- and post-conditions to specify its functionality. To this end, we first need to determine all of the necessary input variables, output variables, and the state variables the process uses, and then formally declare them using well-defined types. The pre-condition presents a constraint on the input and state variables before the execution of the process, and the post-condition gives another constraint for the output and the updated state variables to satisfy. In the semi-formal specification, both the pre- and post-conditions are described in a structured natural language in order to strike a good balance between the usability and the rigor for a high cost-effectiveness. The structured natural language expression is actually a disjunctive normal form in which each term is described in natural language but the logical connectors are formally defined operators (e.g. ***and***, ***or***, ***not***).

As an example, below we show part of the semi-formal specification of the process for the normal switch operation. The partial specification is expressed as a disjunction of several functional scenarios (FS). Each FS is a conjunction of terms described in English. Specifically, the semi-formal specification describes how the switch functions when the system receives a route setting request. First it needs to check the position of the switch. If the position is not the same as the route requests, the system should execute the switch normal or reverse operation. After the operation is done, the system should show the result. In this example, we only describe the semi-formal specification of normal switch operation.

Part of the semi-formal specification of the process for switch operation:

```
module switch_operation_Decom/switch_operation;
  type
CLOCK = nat0;/*time type*/

POSITION = composed of
           normal_indicate: bool
           reverse_indicate: bool
           end;

TIMER = composed of
        acc: CLOCK /*the current time value*/
        delay: nat0 /*maximum time delay*/
        start : bool /*timing flag*/
        end;

POINT = composed of
        sw_id: nat0
        track_id: nat0 /*track which the switch is in*/
        pos: POSITION /*switch position indication*/
        lock: bool /*switch lock state*/
        pt_timer: TIMER /*switch operation timer*/
       end;

ROUTE = composed of
        points: seq of POINT/*switches in the route*/
        pt_req_pos: seq of POSTION/*switch requested position by
                                         route*/
        tracks: set of TRACK/*tracks in the routes*/
        start_sig : SIGNAL/*start signal*/
        end_sig: SIGNAL/*end signal*/
        locked: bool/*route lock state*/
        permissive: bool/*route permissive state*/
        idle: bool /*route idle state*/
        end;

var
  rt: ROUTE
  pt: POINT
process switch_normal_operation(normal_request: sign | normal_cmd:
sign)normal_op_ok: sign | trail_alarm1: sign | normal_cmd: sign, nor-
mal_cmd_output: sign
ext wr pt
    wr rt
pre true
post
```

```
/*FS1: receives a normal operation request, outputs normal operation
command and starts the timer*/
normal operation request is received and
switch is not locked and
switch is in reverse position and
normal operation command is sent out and
the timer is started
or
/*FS2: switch is moving but not getting into normal position, time is
not out, output normal operation command and continue timing*/
the normal operation command in the last cycle is sent out and
switch is not locked and
switch is moving and
time is not out and
normal operation command is sent out and
timer is continuing
or
/*FS3: switch is already in normal position, output success flag*/
The switch is in normal position and
The success flag is sent out and
not normal operation command is sent out and
the timer is terminated and reset
or
/*FS4: switch is moving but time is out, not output normal operation
command and reset the timer, output fail flag*/
the normal operation command in the last cycle is sent out and
time is out and
not normal operation command is sent out and
the timer is terminated and reset
end_process;
process switch_position_check(route_set_req: sign)
normal_request: sign | no_operation: sign |reverse_request: sign
…
end_process;

process switch_reverse_operation(reverse_request: sign | reverse_cmd:
sign)reverse_op_ok: sign | trail_alarm2: sign | reverse_cmd: sign,
reverse_cmd_output: sign
…
end_process;
process switch_op_result(normal_op_ok: sign | trail_alarm1: sign |
no_operation: sign | reverse_op_ok: sign | trail_alarm2:
sign)switch_op_ok: sign | trail_alarm: sign
…
end_process;
end_module
```

2.3.3 Formal Specification

To ultimately resolve the ambiguity in the semi-formal specification, we need to completely formalize all of the informal expressions, such as "*switch is not locked*" in the above process for normal switch operation. However, since some processes in the

specification may depend on other processes in terms of data flows, our experience suggests that it can reduce the chances of modifications of the formal specifications of the processes if their dependency relation can first be defined using a the graphical notation called Condition Data Flow Diagram (CDFD). Taking this into account, we need to fulfill two tasks in constructing the formal specification:

(1) *Draw a CDFD to describe the dependency relation between processes.*
(2) *Formalize the pre- and post-conditions of each process occurring in the CDFD.*

The CDFD not only reflects the dependency relation between processes, but also reflects the architecture of the system. In the architecture, the signature of each process in terms of its name, input, output, and the related data store variables is precisely defined, and all of the relevant processes are connected in terms of data flows and data stores.

When formalizing the pre- and post-conditions of each process in the corresponding module of the CDFD, we need to choose appropriate operators defined in the relevant data types to formally express the informal statements in the semi-formal specification. In some circumstances, we may find that some variables cannot be declared using existing types or some type definitions are not complete. In that case, we need to modify or add some type definitions.

As an example, we show the formal specification for the switch operation, which includes the CDFD in Fig. 3 and the corresponding module given below. For the sake of space, we only give the details of the formal specification of the process for switch normal operation.

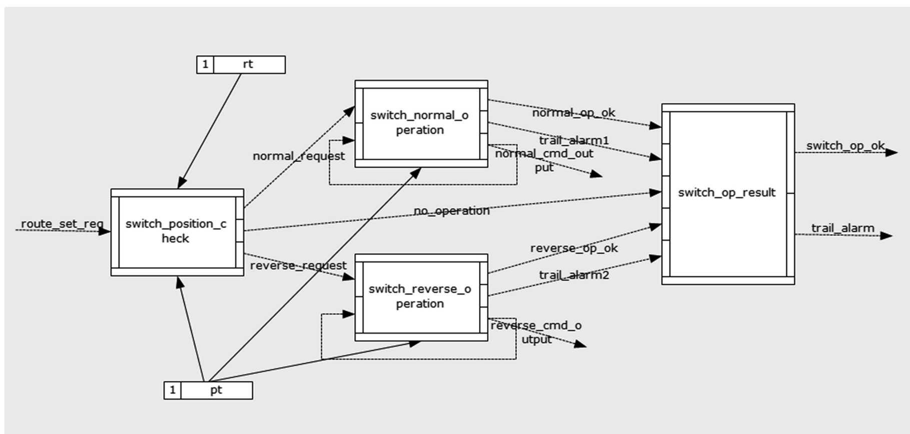

Fig. 3. CDFD of the switch operation module

```
module switch_operation_Decom/switch_operation;
type
… /*inherent from the semi-formal specification.*/
var
…
process switch_normal_operation(normal_request: sign | normal_cmd:
sign)normal_op_ok: sign | trail_alarm1: sign | normal_cmd: sign, nor-
mal_cmd_output: sign
ext wr pt
    wr rt

pre true

post
/*FS1: receives a normal operation request, outputs normal operation
command and starts the timer*/
bound(normal_request)and
not (~pt.locked) and
~pt.pos.reverse_indicate and
not ~pt.pos.reverse_indicate and
normal_cmd and
normal_cmd_output and
pt.pt_timer.start
pt.pt_timer.acc := 0
or
/*FS2: the switch is moving but not getting into normal position, time
is not out, output normal operation command and continue timing*/
bound(~normal_cmd) and
not (~pt.locked) and
not ~pt.pos.reverse_indicate and
not ~pt.pos.normal_indicate and
~pt.pt_timer.start and
~pt.pt_timer.acc< ~pt.pt_timer.delay and
normal_cmd and
normal_cmd_output
or
/*FS3: switch is already in normal position, output success flag*/
~pt.pos.normal_indicate and
not ~pt.pos.reverse_indicate and
success and
not pt.pt_timer.start
or
/*FS4:  the switch is moving but time is out, not output normal oper-
ation command and reset the timer, output fail flag*/
bound(~normal_cmd) and
~pt.pt_timer.start and
~pt.pt_timer.acc>= ~pt.pt_timer.delay and
trail_alarm and
not pt.pt_timer.start
end_process;
… /*inherent from the semi-formal specification */
end_module
```

Since the formal specification preserves the structure of the corresponding semi-formal specification of the same process, we do not repeat the explanation of its meaning here for brevity.

3 Derivation and Analysis of Hazard Conditions

A complete formal specification of a safety critical system should be defined in the way that the functionality of the system must imply the required safety properties. To ensure this point, it is necessary to derive the hazard conditions from the relevant formal expressions that present a potential violation of the safety requirements and to check whether they are valid with respect to the safety requirements. A general distinction between a functional requirement and a safety requirement is that the functional requirement indicates that something must be done, while the safety requirement shows that the result of functional requirement do not lead to hazards [8]. In this section, we present a systematic way to derive hazard conditions from a formal process specification and then discuss how they can be analyzed to determine their validity.

3.1 Derivation of Hazard Conditions

Our previous research [9] shows that any formal process specification can be converted into an equivalent *functional scenario form* (FSF).

Definition 3.1. Let S_{pre} denote the pre-condition and S_{post} the post-condition of process S, respectively. Let S_{post} = G_1 ***and*** D_1 ***or*** G_2 ***and*** D_2 ***or…or*** G_n ***and*** D_n, where $G_i(i = , \ldots, n)$ is known as a *guard condition* containing only input variables and D_i is known as a *defining condition* containing at least one output variable. Then, the following form is called an FSF of S:

S_{pre} ***and*** G_1 ***and*** D_1 ***or*** S_{pre} ***and*** G_2 ***and*** D_2 ***or…or*** S_{pre} ***and*** G_n ***and*** D_n and each S_{pre} ***and*** G_i ***and*** D_i *is called a functional scenario* (FS), defining an independent function.

Our way to derive hazard conditions focuses on each functional scenario. Let T_i ***and*** D_i represents a general functional scenario, where T_i = S_{pre} ***and*** G_i is called *test condition* of the scenario. Our discussions below always refer to this FS. The specific rules for hazard condition derivation are given as follows:

(1) If D_i defines a safety-related operation on some field device, then *Ti* ***and not*** *Di* may describe a hazard condition. For example, suppose

some switch on a route has no position indication ***and*** *the start signal of the route is restrictive*

is a functional scenario in relation to the safety requirements, then we can derive the hazard condition:

some switch on a route has no position indication ***and not*** *(the start signal of the route is restrictive).*

This can further be simplified into the following more intuitive one:

some switch on a route has no position indication ***and*** *the start signal of the route is permissive.*

Obviously, this hazard condition is likely to produce a hazard if it is implemented in the system, because if some switch has no position indication and the start signal of the route is permissive, when the train runs into the route, it will likely derail or roll over.

(2) If T_i describes a critical guard condition (i.e., the violation of it may jeopardize the safety), then ***not*** T_i ***and*** D_i will become a hazard condition. For instance, suppose

(all switches in the route are in right position ***and*** *the route is out of obstacles* ***and*** *no conflicting route is set)* ***and*** *the start signal of the route is permissive*

is a functional scenario, then the following hazard condition can be derived:

not *(all switches on the route are in right position* ***and*** *the route is out of obstacle* ***and*** *no conflicting route is set)* ***and*** *the start signal of the route is permissive.*

It can further be simplified into:

not *all switches in the route are in right position* ***and*** *the start signal of the route is permissive* ***or***
not *the route is out of obstacles* ***and*** *the start signal of the route is permissive* ***or***
not *no conflicting route is set* ***and*** *the start signal of the route is permissive*,

which implies three different kinds of hazards.

To apply these rules effectively, the relevant functional scenarios have to be selected manually based on the safety-related knowledge in the domain in general. The reason is that formal expressions may not make sense if they are not interpreted in the context of the related domain. What our method can help is to systematically and automatically derive a hazard condition after the related specific functional scenario is selected.

3.2 Hazard Condition-Based Testing

After deriving all possible hazard conditions, we need to analyze whether each hazard condition is really implemented into code. To this end, a *hazard condition-based testing* can be carried out.

Specifically, for each derived hazard condition, we generate some test data for the input variables that satisfy the test condition of the hazard condition. Then, we use the test data to run the corresponding program that is supposed to implement the specified functionality of the related process. After obtaining the result of the test, which is the output of the program, we can evaluate the corresponding "defining" condition of the hazard condition. If the defining condition is true, that implies the hazard is already implemented in the code.

Given the hazard condition T_i ***and not*** D_i where T_i is the test condition and ***not*** D_i is the defining condition, applying the above technique, we can generate a test data, say t, to satisfy T_i, and then use t as the input to execute the corresponding program. Suppose we get the result r, then we need to check whether the following condition is true:

$$T_i(t) \Rightarrow \textbf{\textit{not}}\, D_i(r)$$

If the implication evaluates to true, that indicates the fact that the hazard is implemented in the code. For example, considering the hazard condition:

> *some switch on a route has no position indication* ***and not*** *the start signal of the route is restrictive.*

Suppose it is formalized as

$$switch_trail \ \textbf{\textit{and not}} \ signal_restrictive,$$

we generate a test data "true" for the boolean variable *switch_trail*, and use it to run the corresponding program, say P. Assume we get the value "false" as the result for the boolean variable *signal_restrictive*, we then substitute this value for the variable in the hazard condition to check whether the following implication is true:

$$switch_trail \Rightarrow \textbf{\textit{not}} \ signal_restrictive.$$

Obviously, this is true because *switch_trail* is true and ***not*** *signal_restrictive* is true, which means the hazard may happen. This indicates the existence of bugs in the implementation of the related process specification. The same practice can be applied to the other hazard conditions.

As far as test data generation from a hazard condition is concerned, we can treat the hazard condition as a "normal" functional scenario derived from a process specification, and then apply the test data generation criteria proposed in our previous publications [10–12]. Since there is no new discovery about this point in our research, we omit the detailed discussions for brevity.

4 Experience and Difficulties

In this section, we first describe our experience of using SOFL in our project, and then point out some difficulties we have faced. Some of the difficulties have already been resolved through expert consultation, while a few still need to be addressed in the future practice.

4.1 Experience

Our project is planned as a one-year project and our experience of using SOFL so far can be summarized as the following points:

(1) When writing the semi-formal and formal specifications for a process, organizing the post-condition as a disjunctive normal form can significantly help the analyst (i.e., the person who writes the specification) write the specification, achieve its good readability, and check its completeness. The reason is that each conjunctive clause in the disjunctive normal form clearly defines a relatively independent functional scenario, showing under what condition what output is expected.

We found that the way also offers us a clear guideline by which we can rather systematically think about what to write in the specification.

(2) The mechanism for decomposing a high level process into a low level CDFD for defining its functionality in detail is effective to help us formalize some functionally complex processes. In particular, when the formal description of the process functionality inevitably involves the sequential operations, the decomposition of the process into a CDFD is rather straightforward and helpful, because the CDFD notation offers comprehensible graphical representation of sequential operations, parallel operations, and some simple data flow loop structures. One important thing in conducting the decomposition, however, is to keep the consistency between the interface of the high level process and that of the CDFD resulted from the decomposition.

(3) We found that the combination of semi-formal specifications and formal specifications for our system is cost-effective. For some complex processes whose functionality description requires necessary repetition of applying other processes, writing a complete formal specification can be difficult and time-consuming. In this case, we keep the description semi-formal in which only the process signature is precisely defined while the pre- and post-conditions are described in natural language.

4.2 Difficulties

We have also encountered some difficulties in applying SOFL, which include the following aspects:

(1) SOFL does not allow the invocation of another process in the formal specification of a process in order to avoid semantic ambiguity. But this may cause a difficulty for the practitioners who have got used to programming style. How to properly do abstraction in the formal specification to avoid the necessity of calling another process is a challenge to industrial practitioners. To handle this challenge, we turn to SOFL explicit specification. An explicit specification of a process is an abstract program in which the normal program constructs, such as sequence, selection, iterations, and process invocations, can be used to form the program structure and the data types and logical available in the SOFL notation can be used to form conditions and/or statements. However, since the explicit specification involves considerable considerations on the design of algorithm, it may not be suitable for abstract description of process functionality. Another perhaps more balanced way is to use semi-formal statements to express the idea of using another process's functionality in the pre- or post-conditions of the process under specification.

(2) Another problem we have faced is that the formal specification may not be clear enough for the programmer to understand the whole story of the entire system. This will require the programmer to make creative efforts in designing the program structure and the necessary algorithms. To help attack this difficulty, during the process of writing the semi-formal specification, we try to describe the state transitions of each device, which is declared as a data store variable in our

specification, and to get the feedback from the domain expert to clarify the ambiguities and to improve the specification. That is, we take an evolutionary approach to finally complete the formal specification.

5 Related Work

There are some studies about formal methods in railway systems. Haxthausen and Peleska present an abstract algebraic specification and verification for railway signaling system with simple railway network module [13]. The SACEM system [14] used in the RER line in Paris is a successful case of B method. Matra (now is part of Siemens) uses B method in the designing of many similar railway control systems. One of the famous applications is line 14 of RATP (Paris Metro), it used B method to refine the requirement specifications and correct some requirement errors [15]. Zou et al. studies how to formalize and verify the SRS (System Requirement Specification) of CTCS-3 (Chinese Train Control System 3) [16]. HCSP (Hybrid Communicating Sequential Processes) is used to model each basic functional scenario and HHL(Hybrid Hoare Logic) is used to describe the system attributes, and whether the specific HCSP model satisfies the given HLL attributes is formally verified. They also studied how to transform Simulink figures into HCSP and use the HHL to verify HCSP model. The related research results have been applied successfully in the verification of CTCS-3 [17]. Horste et al. formalizes the functional requirements about the ETCS (European Train Control System) [18]. The Ansaldo STS project uses model checking technique to verify the RBC subsystem of ECTS [19]. Many Interlocking systems in lines belonging to RATP (Paris Metro) and NYTC (New York City Transit Authority) were also verified using a model checking tool from Prover technology [20].There have been several years when CASCO started to study and try on formal methods, for the last several years the research is mainly about formal design and verification of ZC subsystem [21, 22]. And from this year, formal modeling and verification techniques have been applied on the interlocking system.

After several years' research before our current project using SOFL, we realized that the formal methods used in the cases mentioned above are quite difficult for practitioners in our company to use, and may not be able to deliver expected results in a short period of time. We also found that the main difficulty for developing a highly reliable and safe system lies in the requirements analysis and specification phases. Our experience so far suggests that SOFL has a much better capability to help us effectively carry out requirements analysis and specification construction, and benefit the subsequent activities in design, coding, testing, and verification of the system.

6 Conclusion and Future Work

We discussed how the SOFL specification language and its three-step approach to writing formal specifications can be applied to an interlocking system in our company. The project is planned for one year and still ongoing. Currently, we have finished the

semi-formal specification and part of the formal specification during which many ill-defined or incomplete requirements in natural language were identified. We are continuing the construction of the formal specification and the derivation of hazard conditions until the end of the project.

After the current project, we will try to carry out specification-based and hazard condition-based testing and verification for the implementation. We will further investigate how adequate test data can be generated from the specification and hazard conditions, and how bugs can be effectively uncovered. If our current project succeeds in terms of providing sufficient benefits or profits to our company, we will extend our experience and practice to more railway signaling systems in the future.

Acknowledgment. This work was supported by CASCO. Shaoying Liu was also partly supported by JSPS KAKENHI grant Number 26240008.

References

1. Boehm, B.W., Basili, V.R.: Software defect reduction top 10 list. IEEE Comput. **34**(1), 135–137 (2001)
2. Bowen, J., Stavridou, V.: Safety-critical methods and systems, formal standards. Softw. Eng. J. **8**(4), 189–209 (1993)
3. Bjørner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language. LNCS, vol. 61. Springer, Heidelberg (1978). doi:10.1007/3-540-08766-4
4. Diller, A.: Z: an introduction to formal methods 23(9), 10–23 (1990). Wiley
5. Abrial, J.-R.: Modeling in Event-B System and Software Engineering. Cambridge University Press, Cambridge (2010), ISBN-13 978-0-521-89556-9
6. Efficient Development of Safe Railway Applications Software with EN 50128 Objectives Using SCADE Suite, 3rd edn.. Esterel Technologies, SA (2012)
7. Liu, S.: Formal engineering for industrial software development using the SOFL method. Springer, Heidelberg (2004), ISBN 3-540-20602-7
8. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language LUSTR. IEEE Trans. Softw. Eng. **18**(9), 785–793 (1992)
9. Liu, S., Chen, Y., Nagoya, F., McDermid, J.A.: Formal specification-based inspection for verification of programs. IEEE Trans. Softw. Eng. **38**(5), 1100–1122 (2012)
10. Liu, S., Chen, Y.: A relation-based method combining functional and structural testing for test case generation. J. Syst. Softw. **81**(2), 234–248 (2008)
11. Liu, S., Nakajima, S.: A decompositional approach to automatic test case generation based on formal specifications. In: 4th IEEE International Conference on Secure Software Integration and Reliability Improvement, Singapore, 9–11 June, pp. 147–155 (2010)
12. Liu, S., Nakajima, S: A "Vibration" method for automatically generating test cases based on formal specifications. In: 18th Asia Pacific Conference on Software Engineering (APSEC 2011), 5–8 December, pp. 73–80. IEEE CS Press, VNU-HCM, Vietnam (2011)
13. Haxthausen, A.E., Peleska, J.: Formal development and verification of a distributed railway control system. IEEE Trans. Softw. Eng. **26**(8), 369–387 (2000)

14. DaSilva, C., Dehbonei, B., Mejia, F.: Formal specification in the development of industrial applications: subway speed control system. In: IFIP Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE), Perros-Guirec, France, 13–16 October, pp. 199–213 (1992)
15. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: a successful application of B in a large project. In: Wing, Jeannette M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999). doi:10.1007/3-540-48119-2_22
16. Zou, L., Lv, J., Wang, S., Zhan, N., Tang, T., Yuan, L., Liu, Yu.: Verifying Chinese train control system under a combined scenario by theorem proving. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 262–280. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54108-7_14
17. Zou, L., Zhan, N., Franzle, M., Qin, S.: Verifying simulink diagrams via a hybrid hoare logic pover. In: International Conference on Embedded Software (EMSOFT), Montreal, QC, 29 September 2013–4 October 2013, pp. 1–10 (2013)
18. Horste, M., Hungar, A., Schnieder, E.: Modelling functionality of train control systems using petri nets. In: FM-RAIL-BOK Workshop, Madrid, Spain, September 23–24, 2013, pp. 46–50 (2013)
19. Cimatti, A., Corvino, R., Lazzaro, A., Narasamdya, I., Rizzo, T., Roveri, M., Sanseviero, A., Tchaltsev, A.: Formal verification and validation of ERTMS industrial railway train spacing system. In: Madhusudan, P., Seshia, Sanjit A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 378–393. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31424-7_29
20. Study cases of Prover technology, http://www.prover.com/company/casestudies/
21. Qian, J., Liu, J., Chen, X., Sun, J.: Formal design and verification of zone controller. In: 21st Asia-Pacific Conference on Software Engineering (APSEC 2014), 1–4 December 2014, pp. 375–382. IEEE CS Press, Jeju (2014)
22. Qian, J., Liu, J., Chen, X., Sun, J.: Modeling and verification of zone controller: the SCADE experience in china's railway systems. In: ICSE Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS), 23 May 2015, pp. 48–54. IEEE, Florence (2015)