DEVELOPING THE TRAM CONTROL SYSTEM BASED ON SIMULINK/STATEFLOW AND B METHOD

CHENG PENG[1,2], WANG KEMING[1,2], HOU XILI[3], LIU NING[2,4] & WANG ZHENG[1,2]
[1]School of Information Science and Technology, Southwest Jiaotong University, China
[2]National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University, China
[3]TongHao GBA (Guangzhou) Smart Control Co., Ltd., China
[4]Graduate School of Tangshan, Southwest Jiaotong University, China

ABSTRACT
The huge gap between the requirements and the system model is an obstacle to the application of formal methods in industry. To reduce this gap, as well as to enhance the consistency and completeness before implementation, we proposed an approach integrating Simulink/Stateflow and Atelier B for developing the tram control system. The Simulink/Stateflow was used to quickly model the requirements for the system. Moreover, we analysed and debugged the logic of the system with the fast-iterative simulation of Simulink/Stateflow. Afterwards, we outlined the analysis in which the B architecture is chosen and manually built the B model according the process flows of the Stateflow model, to further explore through proof of safety invariants. Finally, we introduced the approach by developing the point controlling module of our projects. In this paper, following the approach we presented, not only can the consistency between the requirements and formal specification be improved, but the safety of system model is strengthened.
Keywords: Simulink/Stateflow, B method, formal verification, simulation, safety-critical, tram control system.

1 INTRODUCTION
In recent years, the formal method, a technology with rigorous mathematics foundation, has been successfully applied to the development of the railway safety-critical system. For example, ClearSy [1], a French Company, developed the driverless system of Paris metro line 14 using Atelier B [2], a B method-based IDE tool [3].

However, there are remaining challenges on the application of the B method. The challenges consist of correctness of specifications, correctness of implementation and correctness of proofs, etc. [4]. To revise the incorrect specifications, which lead to the inconsistency between the informal requirement specifications and formal modelling, the UML [5] is used to explore the potential designs before modelling. It can help the coder to understand the design of the system, but is unable to validate the correctness of the functional flow analysis of the requirements. The Matlab Simulink/Stateflow [6] provides the graphical diagrams, which not only can quickly model the behaviors of the control system, but also enable the developer to analyze and debug the logic flow of the model. Nevertheless, the correctness of the model cannot be guaranteed by simulation testing alone. Although the Simulink Design Verifier [7] can use Bounded Model Checking [8] algorithm to generate test cases to improve test coverage, it lacks the sufficient support for formal verification (e.g. The state space explosion of the model). Therefore, we propose an approach which combines the advantages of both Simulink/Stateflow and B method.

We apply the integrated approach to the developing of the tram control system (TCS) of Guangzhou Huangpu Line 1 in China. As far as we know, this is the first application of B method on the tram control system in China. The approach boosts our confidence in design consistency and completeness before implementation.

The rest of this paper is organized as follows. The developing map of our project presented in this paper follows the discussion of Section 1. By introducing a case, point controlling module (PCM) of TCS, the processes of Stateflow modelling and translating that are described in Section 1 are developed into a concrete activity in Section 2. In Section 3, the simulation steps are listed to explain how to do iterative simulating; the example of formal verification is given to illustrate the benefits between simulation testing and formal technologies. Finally, we discuss the related work and conclude our work.

2 THE PROCESSES OF OUR APPROACH

Developing the safety-critical systems in the railway domain usually follows the V model. Here we define an enhanced V model in our approach, as depicted in Fig. 1.

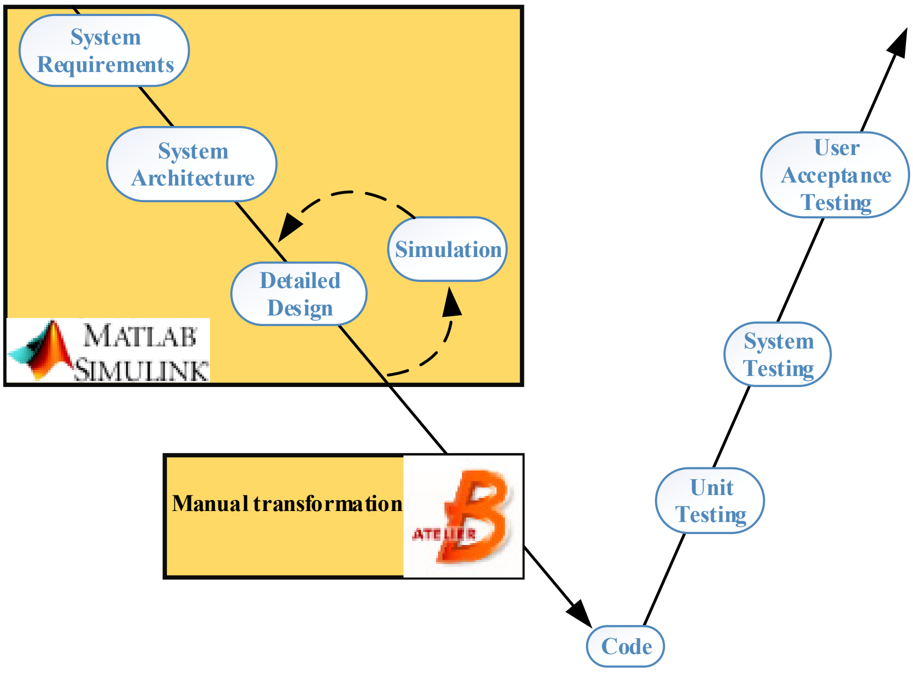


Figure 1: The developing map.

In Fig. 1, the approach consists of two processes, which are described below:

1. Modelling and validation based on Simulink/Stateflow: the flow charts and state machines of Stateflow are used to model the system functions, which are derived from the requirements. After the Stateflow model was established, the correctness of its function flow is validated through fast iterative simulation, which is a cyclic control represented by the dashed line in Fig. 1.

2. Modelling and verification based on Atelier B: we manually build a B model following the architecture and logic flows of Stateflow model, which will be proved in the proof environments of Atelier B.

3 MODELLING AND TRANSLATING THE PCM

In this section, along with the steps of this integration approach, the PCM and its development will be illustrated.

3.1 The brief introduction of PCM and its simulink/stateflow

The TCS includes many modules such as *PCM*, *Route*, *Manual*, etc., which are required to achieve safety integrity level 4 (SIL4) and comply with the European standard for railway software, CENELEC EN50128 [9]. Fig. 2 shows the connections between PCM and other modules, which cooperate with the data-flows to send and receive the commands.

Simulink contains a set of blocks, sub-systems, and wires. Stateflow can be encapsulated as blocks or sub-systems, where Stateflow can be modelled by a hierarchical structure. Fig. 3 is presented to offer a real hierarchy snippet of our Stateflow model and it is also an extension of the POL diagram. The wires connect among the different blocks or sub-systems to cooperate with the data-flows. To provide the actual connections, Fig. 4 draws a Simulink project on the basis of the Fig. 2.

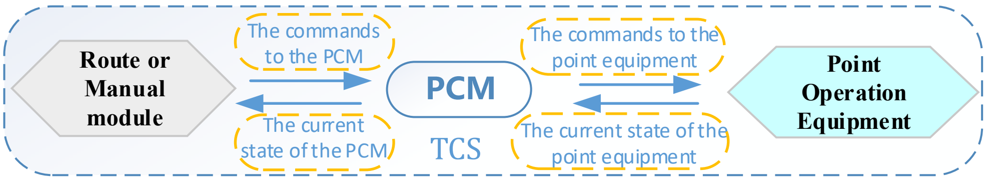

Route or Manual module
The commands to the PCM
The current state of the PCM
PCM
TCS
The commands to the point equipment
The current state of the point equipment
Point Operation Equipment

Figure 2: The framework of the PCM.

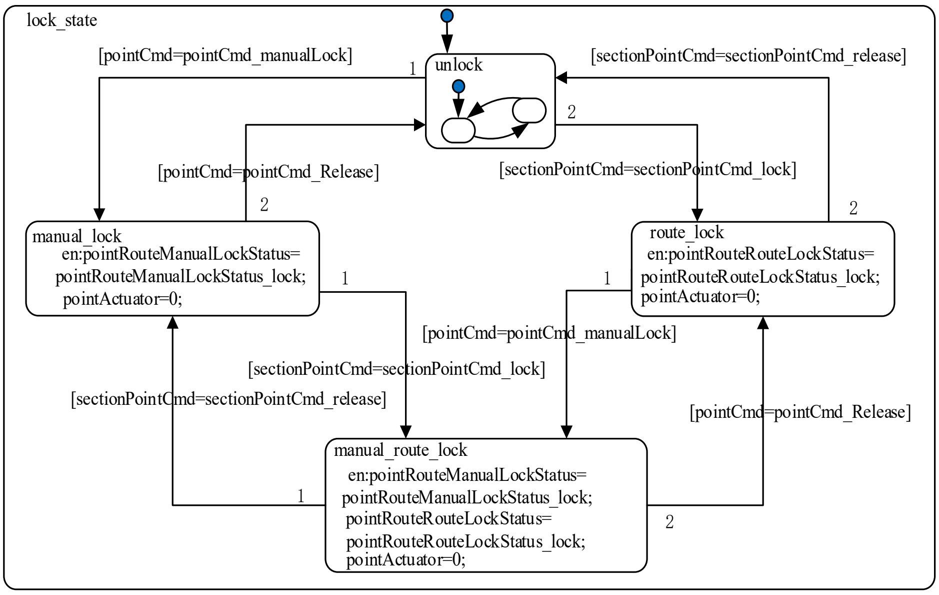

lock_state
[pointCmd=pointCmd_manualLock]
1
unlock
[sectionPointCmd=sectionPointCmd_release]
2
[pointCmd=pointCmd_Release]
[sectionPointCmd=sectionPointCmd_lock]
2
2
manual_lock
en:pointRouteManualLockStatus=
pointRouteManualLockStatus_lock;
pointActuator=0;
1
1
route_lock
en:pointRouteRouteLockStatus=
pointRouteRouteLockStatus_lock;
pointActuator=0;
[pointCmd=pointCmd_manualLock]
[sectionPointCmd=sectionPointCmd_lock]
[sectionPointCmd=sectionPointCmd_release]
[pointCmd=pointCmd_Release]
manual_route_lock
en:pointRouteManualLockStatus=
pointRouteManualLockStatus_lock;
pointRouteRouteLockStatus=
pointRouteRouteLockStatus_lock;
pointActuator=0;
1
2

Figure 3: The picture depicts the *unlock* state, it has a sub-diagram as well as the extension of POL.

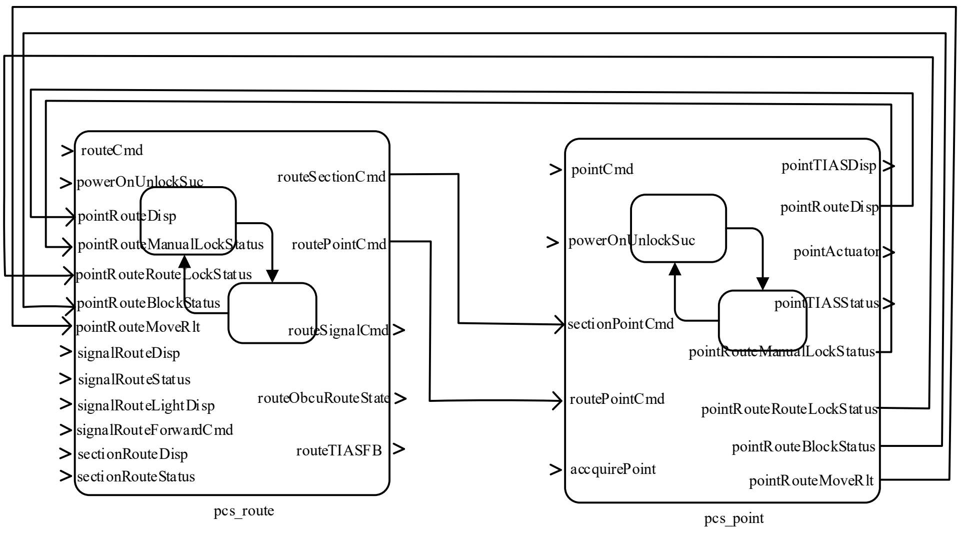


Figure 4: The connections between Route and PCM.

Fig. 2 ignores the command *PowerOnUnlockSuc*, which is the input of *pcs_point* of Fig. 5. In this paper, *pcs_point* is approximately equivalent to PCM.

3.2 Modelling Stateflow of PCM

In Section 1, the first step of our approach is modelling and validation based on Simulink/Stateflow. As a correspondence to the first step, modelling PCM is dedicated in this part and the validation of PCM is discussed in Section 3.

Stateflow acts as a bridge between the requirements and system design. We use Stateflow to create a multiple-level hierarchical model to represent the system functions that is decomposed by the domain experts. The different levels indicate the different development stages of the model, which are portrayed as a development views, from abstract to concrete.

The Stateflow diagram of each level can be an AND diagram, for which states are distributed in parallel and all of them are actually executed in sequential order according to their priority once AND is triggered; or an OR diagram, where only one of the states is active when OR is activated. Fig. 5 gives the Stateflow diagram of the *pcs_point*, containing the *PowerOnlock* module and PCM module. The Stateflow of PCM is an AND diagram, which is composed of three parallel states: point operating and locking (POL), point blocking (PB), and the display of the train integrated automated system (TIAS), respectively. Because of hierarchy, POL is modelled as a sub-diagram and it is also an OR diagram, comprising four states, *unlock*, *manual_lock*, *route_lock*, and *manual_route_lock* respectively.

After the Stateflow model was established, the correctness of its functional flow was validated in the simulation environments. In Fig. 4, the data-flows that are connected between the PCM and Route constitutes closed loop control. The loop can be iteratively simulated to debug and analyze the correctness of function of the model, which are discussed in Section 3.

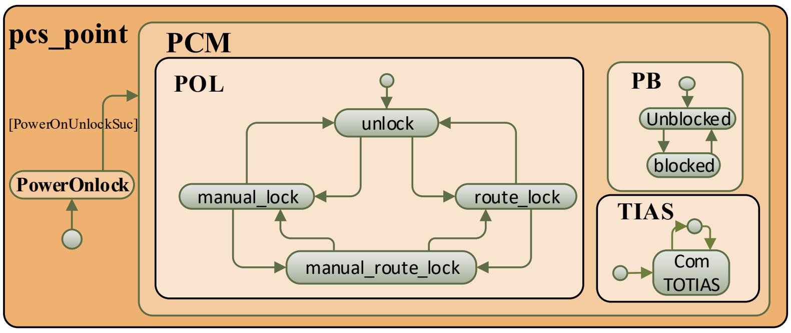


Figure 5: The overview of Stateflow of pcs_point.

3.3 Translating Stateflow to B

Owing to the correctness of the Stateflow model cannot be guaranteed by simulation alone, formal technologies are introduced into the development process to deal with this issue. In Section 1, the formal activity is the second step of our approach, which is divided into the modelling and verification based on Atelier B. Modelling PCM is also the translating of PCM. As a correspondence to the second step, translating is analyzed in this part and the verification is discussed in Section 3.

In terms of translating of PCM, two aspects should be inherited by B modules, which are the structure of the hierarchical diagram of Stateflow model and its detailed design respectively.

3.3.1 Translating the structure of Stateflow to B module

The first aspect concerns the inheritance of the architecture of Stateflow. Fig. 6 provides the mathematical relation which represents a mapping, from Stateflow diagram to B modules. The mapping is defined as a total surjection:

$$Transition \in \{D1, D2, ..., Dn\} -->> \{B1, B2, ..., Bn\}.$$

In Fig. 6, the set $\{D1,\ D2,\ \ldots,\ Dn\}$ denotes all diagrams of this module and the set $\{B1,\ B2,\ \ldots,\ Bn\}$ denotes the all B modules, where a B module can be made up of three components – abstract machine specification, refinement, and implementation [1].

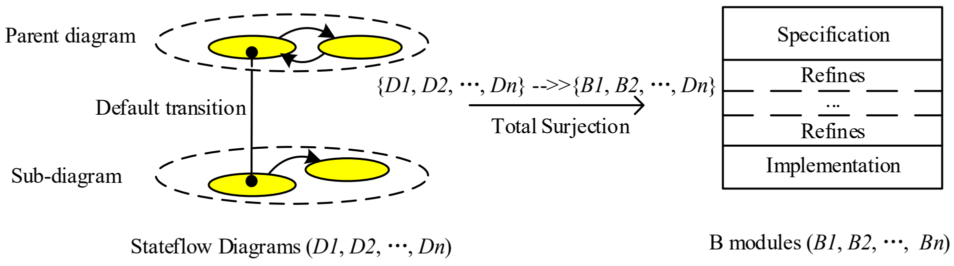


Figure 6: The mathematical relation between Stateflow and B.

Since these B modules can build a B project through different architectures, hand in hand with the decision on how to translate the Stateflow architecture goes the choice of B architecture to develop a B project. The brief analyses are outlined for choosing the translations.

Translation analysis of B development: as far as the B architecture of PCM is concerned, we believe there are three mechanisms of translation (In fact, there are only two for this paper).

(1) Translating the structure of Stateflow to system modelling option.

System modelling has always been a development option when it comes to using B method for system level. The mapping should be defined as a concrete total surjection:

$$\mathrm{Re}\,finement\_structure \in \{D1, D2, ..., Dn\} -- >> \{B1\}.$$

On the basis of the definition, all diagrams of PCM are coupled into a B module. It actually adopts a coupling translation paradigm. To construct final software systems, one issue should be taken which is that the specification of each B module needs comprise the inputs and outputs of all diagrams of a Stateflow module. However, this mechanism can demonstrate system but does not generate codes for system. To this end, B method provides software development option for code generation. Software development mainly adopts INCLUDES and IMPORTS mechanisms to develop system [1].

As a preliminary statement below, the mapping of (2) and (3) both are defined as a bijection in this module:

$$INCLUDES\_IMPORTS \in \{D1, D2, ..., Dn\} > - >> \{B1, B2, ..., Bn\}.$$

(2) Translating the structure of Stateflow to INCLUDES mechanism (software develops). INCLUDES mechanism is used to bring together the components, where the information of the *included* component can be obtained by the *inclusion* component. It is much fitter for developing specification text. According to the bijection, each diagram of PCM corresponds to a B module that includes a specification. INCLUDES mechanism is first used to link among the specifications, then they are implemented separately.

(3) Translating the structure of Stateflow to IMPORTS mechanism (software develops).

IMPORTS mechanism links between an implementation and a specification. The implementation can use specification's data to implement its own data and operation. It is more suitable for constructing software systems. Based on the bijection, these specifications are first independently developed into the implementations, where each implementation uses IMPORTS mechanism to link corresponding specification.

Result for developing PCM: In terms of Stateflow of PCM, the specifications have been divided into different hierarchical diagrams. Intuitively, such a Stateflow specification offers exceptional advantages for software development. From our opinion, for choosing (2) or (3), the main criteria are how the relations between different hierarchical diagram in PCM are linked. By the composition of the specifications, INCLUDES mechanism can build a refinement structure of PCM. For instance, if POL *includes pcs_point*, the guard conditions of each transition of POL should *include* the condition *PowerOnUnlockSuc* in *pcs_point*. IMPORTS mechanism divide the guard conditions of the refinement structure into different implementations through decomposing an implementation into a number of specifications. For example, the *PowerOnUnlockSuc* is not *included* POL in case of *pcs_point imports* POL, but the *PowerOnUnlockSuc* is still the guard condition of the transitions of POL as the substitution of the implementation of *pcs_point* is decomposed into the operation of the specification of POL. Consequently, both (2) and (3) can be used to develop PCM.

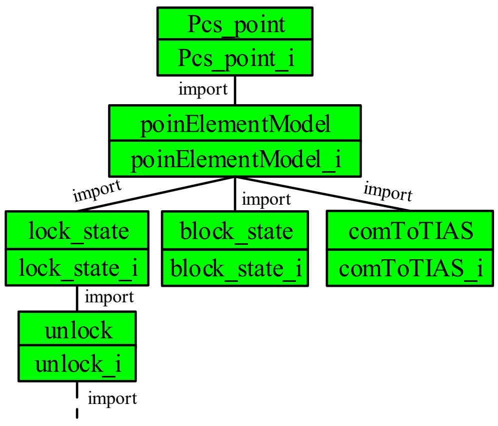


Figure 7: Snippets of the B architecture of PCM.

Fig. 7 shows that (3) is used to develop the architecture of PCM in this paper.

In Fig. 7, the four abstract machines – *pointElementModel*, *lock_state*, *block_state*, and *comToTIAS* respectively denote PCM, POL, PB, and TIAS. The *unlock* abstract machine denotes the state *unlock* of POL, which has a sub-diagram.

3.3.2 Translating the contents of Stateflow to B module

Once the architecture is determined, another aspect will concern how the contents of each specification and its refinement and implementation (*B1*, *B2*, …) should be developed. The main elements of a Stateflow diagram include a finite set of states, transitions, and variables.

States: except for having the hierarchy like *unlock* state in POL, a state may be involved with the three optional types of actions: An *entry action* executed when the state is activated; A *during action* executed when no transition is satisfiable; An *exit action* executed immediately before the source state is marked inactive.

Transitions: to select an active state of the OR diagram on condition that the OR diagram was activated, default transitions with no source states or source junctions are allowed. For instance, the *unlock* state becomes active as the POL diagram was triggered. A transition might consist of several transitions joined by junctions such as in TIAS diagram. A transition label is a line with an arrowed, from source state to target state; It can contain an *event*, a *condition*, a *condition action*, a *transition action*, which are formed as follows:

$$transition\_label = event[condition]\{condition\_action\}/transition\_action \quad [17].$$

The valid transition label can be an *event* only; or a *condition* only; or an *event* and a *condition*; or NOT specified; etc. The *condition_action* will be executed when the *event* is triggered and the *condition* becomes true. The *transition_action* is executed after the relevant transitions were determined to be valid. For instance, the *transition_action* is triggered after the *exit action* of source state was exectuted.

Variables: the variables may be a set of input data-flows, output data-flows, or local, and them exist in the *condition*, *action*, etc.

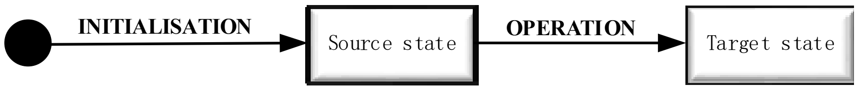
We translate the main elements of a Stateflow diagram into the OPERATIONS CLAUSE and the INITIALISATION CLAUSE of an implementation of a B module. The relation between them and ELSIF definition are depicted in Fig. 8. Apart from the ELSIF structure, other substitution structures are allowed as long as the substitution can be implemented.

Following the previous definition that each diagram corresponds a B module, the translation items are presented as shown below:

1. The States in each diagram are defined as a variable to denote the states such as the variable *pointLockstate* in Fig. 9;
2. The Variables of each diagram are defined as the input and output parameters (e.g. the *pointCmd* in Fig. 9) of OPERATIONS. The parameters will be replaced by the temporary variables to verify the model with INVARIANT CLAUSE;
3. INITIALISATION is coded by translating the default transitions of Stateflow.
4. The predicates of OPERATIONS may be expressed by the source state itself (e.g. the variable *pointLockState* in Fig. 10) and *event* or *condition* in the transition label;
5. The substitution of OPERATION can be *condition_action*; or *transition_action*; or relevant *entry action*, *during action*, and *exit action*; or a target state itself. Besides the substitutions above, the OPERATIONS of the specifications to be *imported* by the implementation are also substituted.

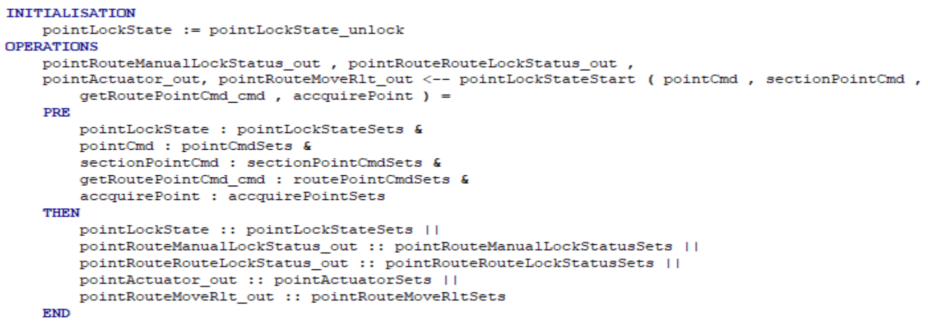
Based on the translation, the snippets of code of the PCM are given to explain the above discussion. The snippets of code of *lock_state* as shown in Fig. 9, where the sets like *pointCmdSets* are defined in the Context file, in which the data-flows such as *pointCmd* are derived.

The snippets of code of *lock_state_i* as shown in Fig. 10, where the transitions and states in POL diagram are inherited. In Fig. 10, the operation *unlockStart* comes from the unlock machine. There is no refinement machine as an interim from *lock_state* to *lock_state_i*.



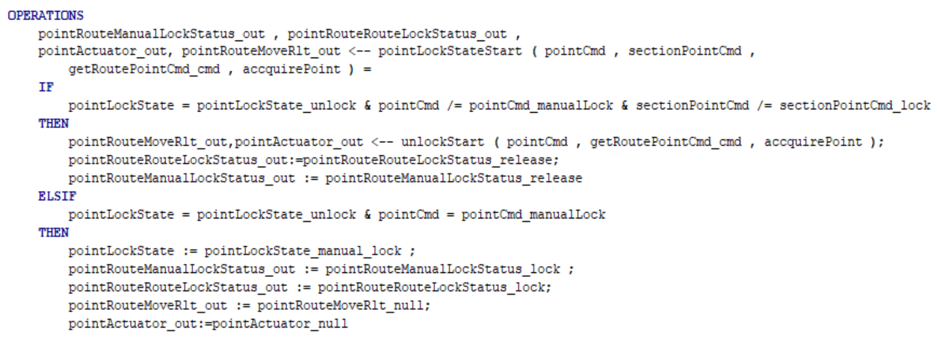
INITIALISATION : BEGIN S END;
OPERATION: IF P THEN S ELEIF Q THEN T ELSE U END;
Where P and Q are predicate and S, T , and U are substitution

Figure 8: The relation and ESLIF definition in the implementation of B module.



```
INITIALISATION
    pointLockState := pointLockState_unlock
OPERATIONS
    pointRouteManualLockStatus_out , pointRouteRouteLockStatus_out ,
    pointActuator_out, pointRouteMoveRlt_out <-- pointLockStateStart ( pointCmd , sectionPointCmd ,
        getRoutePointCmd_cmd , accquirePoint ) =
    PRE
        pointLockState : pointLockStateSets &
        pointCmd : pointCmdSets &
        sectionPointCmd : sectionPointCmdSets &
        getRoutePointCmd_cmd : routePointCmdSets &
        accquirePoint : accquirePointSets
    THEN
        pointLockState :: pointLockStateSets ||
        pointRouteManualLockStatus_out :: pointRouteManualLockStatusSets ||
        pointRouteRouteLockStatus_out :: pointRouteRouteLockStatusSets ||
        pointActuator_out :: pointActuatorSets ||
        pointRouteMoveRlt_out :: pointRouteMoveRltSets
    END
```

Figure 9: Snippets of the B code of *lock_state*.



```
OPERATIONS
    pointRouteManualLockStatus_out , pointRouteRouteLockStatus_out ,
    pointActuator_out, pointRouteMoveRlt_out <-- pointLockStateStart ( pointCmd , sectionPointCmd ,
        getRoutePointCmd_cmd , accquirePoint ) =
    IF
        pointLockState = pointLockState_unlock & pointCmd /= pointCmd_manualLock & sectionPointCmd /= sectionPointCmd_lock
    THEN
        pointRouteMoveRlt_out,pointActuator_out <-- unlockStart ( pointCmd , getRoutePointCmd_cmd , accquirePoint );
        pointRouteRouteLockStatus_out:=pointRouteRouteLockStatus_release;
        pointRouteManualLockStatus_out := pointRouteManualLockStatus_release
    ELSIF
        pointLockState = pointLockState_unlock & pointCmd = pointCmd_manualLock
    THEN
        pointLockState := pointLockState_manual_lock ;
        pointRouteManualLockStatus_out := pointRouteManualLockStatus_lock ;
        pointRouteRouteLockStatus_out := pointRouteRouteLockStatus_lock;
        pointRouteMoveRlt_out := pointRouteMoveRlt_null;
        pointActuator_out:=pointActuator_null
```

Figure 10: Snippets of the B code of *lock_state_i*.

Section summaries: Stateflow can quickly generate the logic model of the specification. Moreover, a well-defined architecture of the software system and detailed design are established by Stateflow, which helps to better understand the behaviors of the system during the formal modeling activity.

4 SIMULATION AND FORMAL VERIFICATION

4.1 Simulation

In this part, by integrating PCM and the Test Sequence within the Test Harness, the test-specific simulation environments are built as shown in Fig. 11.

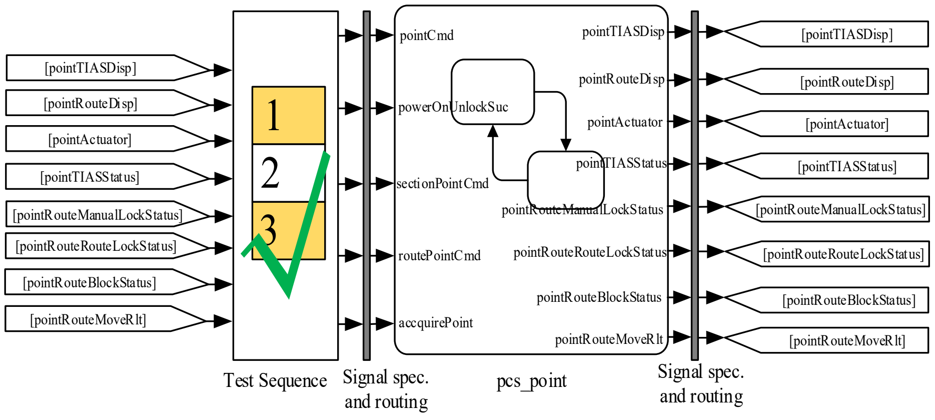


Figure 11: The simulating environment of PCM.

In Fig. 11, the outputs of PCM is linked with the inputs of the Test Sequence. A Test Sequence contains test steps, where a test step consists of actions and transitions. Action executes at the starting of the step. Transition defines when the step stops executing. The entire simulation progresses observe the logical transformations within PCM by periodically performing the steps defined in the Test Sequence. A step comprises the following three processes:

1. The first is to assign values to the inputs of PCM through executing the outputs of Test Sequence; The outputs are the actions of a step, which can be edited in the Test Sequence Editor.
2. After the outputs of PCM was returned to the Test Sequence, it can be assessed whether to comply with the expectation;
3. The current transition condition is then validated. If it is satisfiable, it jumps to the next step to execute. Otherwise, it stays at the current step.

When the simulation model is running, the results are sent to the Test Sequence. Based on the results, the model is analyzed to validate whether it meets the requirements.

One concern for our work is how does the result of simulation conducive to assist formal proving. Indeed, the simulation information only are used to correct the Stateflow model while not work on proving. However, based on the checking of the simulation results, the accuracy of the formal modelling can be improved.

4.2 Formal verification

Verification with invariants: after the B model was built, the hidden safe defects can be further explored by the obligation proving of the safety invariants.

Note that there is no gluing invariant in *lock_state_i* since the variables *imported* from the other abstract machines are not refined relation for the variables in *lock_state*. In addition, the variable name is consistent from abstract machine to its implementation.

Example: we assume that the safety requirement is whether the reachability of the state *manual_route_lock* in Fig. 3 can satisfy the requirement, then the invariant is:

$$\begin{aligned}&pointLockState = pointLockState\_manual\_route\_lock \Rightarrow\\&verify\_section = sectionPointCmd\_lock \wedge\\&verify\_manual = pointCmd\_manualLock\end{aligned}$$

Notice also that the two temporary variables *verify_section* and *verify_manual* separately represent the two input commands *sectionPointCmd* and *pointCmd*, as the two input commands are parameter that INVARIANT CLAUSE is unable to identify. In Fig. 3, the three variables *pointRouteManualLockStatus*, *pointRouteRouteLockStatus*, and *pointActuator* respectively become equal to *pointRouteManualLockStatus_lock*, *pointRouteRouteLockStatus_lock*, and 0 when the entry actions of the state *manual_route_lock* are executed. In the invariant, the value of the variable *pointLockState* is to represent the values of two variables *pointRouteManualLockStatus_lock* and *pointRouteRouteLockStatus_lock*.

In the proof window, the deduction result of the invariant does not involve violations which indicates that the value of the variable *pointLockState* becomes equal to the state *pointLockState_manual_route_lock* when the *sectionPointCmd* and *pointCmd* hold true.

Section analyses – Benefits analysis between simulation and formal verification: model-based simulation technologies can validate the functional flow correctness of the model from requirements by editing test cases that are executed logically. Thus, we conclude that there is an improvement for the correctness of requirement specifications under the above simulation testing. Nevertheless, the effectiveness of them cannot satisfy the criterions of high-test coverage and completeness specifications in safety-critical applications. For this, formal technologies are utilized to fill the gap and complement verification mechanism.

5 RELATED WORK

The huge gap between the informal requirements and formal specifications has been an issue of interest in the safety-critical domains over past decades. It promotes quite a few works to reduce this gap. For example, two approaches are presented to integrate the structured analysis and VDM in Fraser et al. [8], and Object Constraint Language specifications from UML cases are used as a guidance for customer-side to develop formal specifications in Giese and Heldal [9], and so on. For B method, using UML has been a de-facto standard for modelling software systems in informal stage. Many researches have been done on the transition from the UML to B language such as Laleau and Mammer [10], [14]. In Fotso et al. [12], SysML/KAOS, a requirement engineering method of extending SysML UML profile, is used to derive B system specifications; Being apart from extracting the elements of the model, it translates the ontologies, refinement structure etc., of the model into B system specifications to build more links between models.

Besides the concern of bridging the gap, two issues have been widely discussed at the intersection of the software development community and formal method. The first one is related to the benefits of software ecosystems. Historically, software engineering has been also seeking a high level of accuracy at earlier stages of development. According to the statistics [13], about one-third of the errors in the product are buried in the requirement stages, and the earlier they are corrected, the lower the cost. In our work, by modelling the requirements with Stateflow and using the simulation in Simulink environments to correct the errors of the informal model, not only is the gap filled but also the accuracy of formal modelling activities has an improvement.

Another issue attaches the industry development challenge of formal methods, and different formal technologies have their own challenges. In this paper, we focused on the challenge of introducing the B method into the industry. In Lecomte [3], although the B method are successfully used in the railway domain, the main challenge is poor spreading in the safety-critical industrial community; One main reason is that it is too difficult to modify the developing approaches which has been established over many years in the aeronautics, energy, etc. [14]. As a possible future direction, the approach we presented would be interesting to extend formal methods to more safety-critical domains by combining the advantages of Simulink project and B method.

6 CONCLUSION

Industrial developing including formal method exist a huge gap between requirements and formal specifications. In this paper, aimed at this gap, we proposed an approach integrating the advantages of Simulink/Stateflow and B method on the developing of the safety-critical system. The process of this approach was introduced in an improved V model, with its function being illustrated by developing PCM of TCS. After the functions of PCM from requirements were modelled by Stateflow, two aspects are concerned towards the transformation from Stateflow to B: (1) the analyzes are outlined to choose suitable mechanisms for the structure transformation; and (2) the elements of Stateflow diagram are described to explain the corresponding translation relation. Afterwards, orienting the higher industry standard, we clarified the advantages of formal verification over simulation.

While bridging the gap, our approach contributes to increasing the benefits of software development. According to our developing experiences, the approach we proposed not only can check the correctness of the requirements, but also significantly improves the accuracy of the formal modeling processes which consequently enhance safety of the system.

Model-based engineering involving formal method still are not common. As a spread, the integration may enable more industrial applications to apply formal method. The possible future work is to translate Stateflow model automatically into the B model, so as to establish a highly practical approach based formal method for the development of the industrial control systems.

ACKNOWLEDGEMENT

We would like to thank the National Natural Science Foundation of China (No. 71502146, 61673320).

REFERENCES

[1] Clearsy. www.clearsy.com/en. Accessed on: 6 Apr. 2020.
[2] Lecomte, T., Deharbe, D., Prun, E. & Mottin, E., Applying a formal method in industry: a 25-year trajectory. *SBMF 2017, LNCS*, eds. S. Cavalheiro & J. Fiadeiro, Springer: Cham, pp. 70–87, 2017. Clearsy. www.clearsy.com/en. Accessed on: 6 Apr. 2020.
[3] Abrial, J.R., *The Book title: Assigning Programs to Meanings*, Cambridge University Press: Cambridge, 1996.
[4] Batra, M., Formal methods: Benefits, challenges and future direction. *Journal of Global Research in Computer Science*, **4**(5), pp. 21–25, 2013.
[5] Rational Software Corporation: OMG Unified Modeling Language Specification – version 1.4., Sep. 2001.
[6] Mathworks Automotive Advisory Board (MAAB): Control Algorithm Modeling Guidelines Using Matlab, Simulink and Stateflow, Version 2.0, 2007.
[7] The Mathworks, Simulink Design Verifier. www.mathworks.com/products/simulink-design-verifier.html. Assessed on: 6 Apr. 2020.
[8] Clarke, E. et al., Bounded model checking using satisfiability solving. *Formal Methods in System Design*, **19**(1), pp. 7–34, 2001.
[9] CENELEC – EN 50128, Railway applications-communication, signaling and processing systems-software for railway control and protection systems, 2011.
[10] The Mathworks, Help Center, Transitions. www.mathworks.com/help/stateflow/ug/transitions.html. Assessed on: 6 Apr. 2020.
[11] Fraser, M.D., Kumar, K. & Vaishnavi, V.K., Informal and formal requirements specification languages: bridging the gap. *IEEE transactions on Software Engineering*, **17**(5), pp. 454–466, 1991.
[12] Giese, M. & Heldal, R., From informal to formal specifications in UML. *International Conference on the Unified Modeling Language*, pp. 197–211, 2004.
[13] Laleau, R. & Mammar, A., An overview of a method and its support tool for generating B specifications from UML notations, *ICS*, pp. 269–272, 2000.
[14] Snook, C. & Bulter, M., UML-B: formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodolgy*, **15**(1), pp. 92–122, 2006.
[15] Fotso, S.J.T. et al., Event-B expression and verification of translation rules between SysML/KAOS domain models and B system specifications. *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Cham, pp. 55–70, 2018.
[16] Leffingwell, D. & Widrig, D., *Managing Software Requirements: A Use Case Approach*, Addison-Wesley: New Jersey, pp. 10–40, 10–40.
[17] Abrial, J.R., On B and event-B: Principles, success and challenges. *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Cham, pp. 31–35, 2018.