Integrity Analysis of Digital ATC Track Database with Automatic Proofs

Natsuki TERADA
Railway Technical Research Institute
Hikaricho 2-8-38, Kokubunji-shi
Tokyo 185-8540, JAPAN

Abstract

We designed the specification of the onboard track database for digital ATC (automatic train control) system with VDM and analyzed its integrity with automatic proofs, which is provided as the pilot proof support for VDM-Toolbox. We describe the precise of the analysis. And we also describe some experiences that gives us a lot of hints to make automatic proofs easier.

The specification is written in VDM-SL. In this specification, track property such as length and location of track circuits and speed limits are modelled as well as routes and stations, etc. In the specification, the requirements are implemented as invariants that should be kept at any time. On top of that, manipulations of the database are defined in such a way that they keep the invariants.

From the specifications, proof obligations are generated automatically. They are requirements to verify the consistency of the specification such as satisfaction of keeping invariants, precondition, etc. Some mistakes are found while looking at the specification.

Automatic proof support is available for the proof obligations. We can use full automatic proofs and interactive proofs. In both cases all we need is just clicking buttons, although interactive proofs need some experience and good understanding of the specification. We found some mistakes of the specification through these proofs. After several modifications, 90% of the proof obligations are proved fully automatically and the remains are proved interactively. This means that the integrity of the specification is proved fully mechanically.

1 Introduction

Railway Signalling is a safety critical system. Signalling systems are responsible for the safety of train. No collision is allowed. Special hardware such as relay has been used for the railway signaling. Recently computer has been widely introduced to the signaling systems. For the hardware aspect of the computer, the safety techniques are being established with special devices and redundant systems. For example two out of three system is used. The comparing circuit for this system has a function to be turned off when the circuit itself fails.

On the other hand, safety techniques for the software are not mature yet in spite that the portion of software getting larger. Formal methods may be a good solution for this issue. With formal methods, automatic and rigorous verification of the specification is enabled. Especially we are interested in the verification with automatic proofs. We applied formal methods with automatic proof to the specification of the database of the digital ATC system. In this report, we describe the analysis and discuss the effect of the formal analysis.

1.1 Application of Formal Methods in the Railway System

There are already railway applications of formal methods. In France, in development of SACEM, a train control system in RER line in Paris, the early version of B[3] is used[2]. B is also used in another ATC system of Paris metro[4]. In Sweden in the verification of Ebilock, an interlocking system (interlocking is a system which controls switches and signals in the station), specification is written in STERNOL, and verified with NP-Tool[5].

The use of formal methods are recommended in the international safety standard IEC61508. So many applications are expected to be appear.

1.2 PROSPER and VDM-Toolbox with proof support

In this report we used VDM. The VDM-Toolbox, provided by IFAD in Denmark, features syntax and type checking, interpreter and test coverage, but proofs are not supported. So IFAD stress on modelling aspect of VDM. Of course IFAD recognizes the demand for proof support and IFAD participated in EU's ESPRIT project PROSPER[6] and developed alpha version of proof facilities. We had a chance to use it, and applied this tool to our project, explained in the following sections. In PROSPER academic research of automatic theorem proving is incorporated in the industrial CASE and CAD tools. VDM-Toolbox is an application of CASE tool. In the development of proof support of VDM-Toolbox, our study is used to improve the performance of proofs.

2 Digital ATC Track Database – Object of The Analysis

We applied formal methods to the specification of digital ATC track database. We explain what this is.

2.1 Digital ATC System

ATC(automatic train control) system has been used in Shinkansen (Japanese high speed train service) and some heavy commuter lines in Japan. It is very reliable system and there is no accident with fatality in relation with this system. But there are some problems in concern with the efficiency of traffic. In the conventional ATC system, the maximum speed is transmitted to the train for each blocks. The signal is called speed signal and its maximum speed is calculated in advance according to the trains with worst braking performances. In case of trains with good braking performance, the braking curve, which means the trace of the train speed against the location while slowing down, is multi steps and, as a result, the running time and headway get longer.

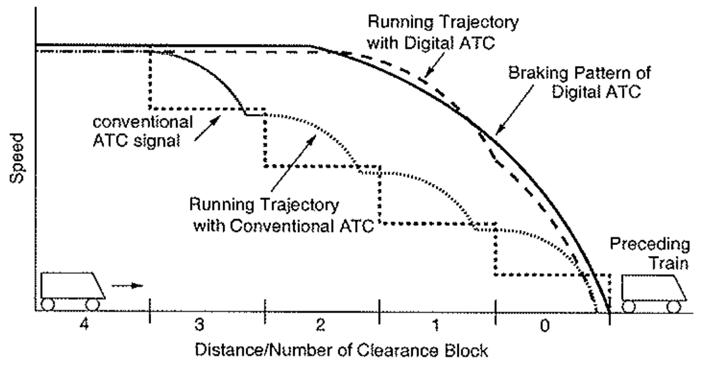

Running Trajectory with Digital ATC
Braking Pattern of Digital ATC
conventional ATC signal
Speed
Running Trajectory with Conventional ATC
Preceding Train
4
3
2
1
0
Distance/Number of Clearance Block

Figure 1: Concept of Digital ATC System

To overcome this problem, digital ATC[7] has been developed. In this system, the signal is a kind of distance signal. The number of clear blocks and the ID of the block where train stays are sent to the train by digital transmission. The onboard system looks up the database from that information, and calculates the braking curve according to the property of the train itself. This enables to calculate the optimum braking curve for each train. This system moves the subject of the calculation of braking curve from ground equipment to the vehicle. This means that onboard system is much more responsible for the safety.

The research phase of this system is finished, and now this system is being introduced to a part of Shinkansen. First section equipped with this system is going to appear in this year.

2.2 Structure of the database

The database is very important for the safety as well[8]. If the database is not correct, it results in dangerous braking curve that allows the collision and derailment. So the high integrity is required for the database. The difference between the database and the actual equipment cannot be examined with formal methods, but the inconsistency of the properties within the database can be checked. We tried to make the structure of the database with consistency. The structure of the track database can be drawn as Figure 2.

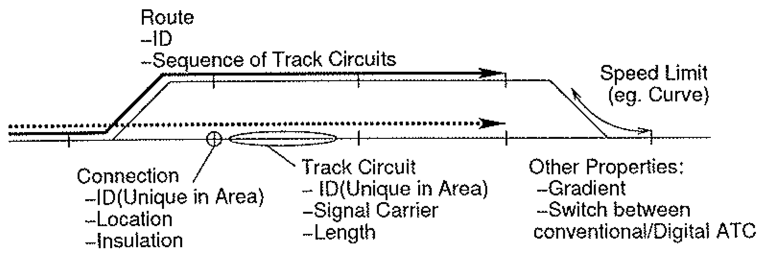


Figure 2: Structure of The Track Database

The basic components of the database are track circuits. A track circuit is an electric circuit which uses the two parallel rails as a part of it and wheels of vehicle as the switch (see Figure.3.) Then existence of a train can be detected. The region where the existence of a train can be detected with the track circuit is also called track circuit hereafter. In this report you can interpret track circuits as blocks. Track circuits have more functions. The current of track circuits carries signals related with ATC systems. The information of track circuits in the database includes that of ATC signal, the location and the boundary information. Each track circuit has a unique ID in the area.

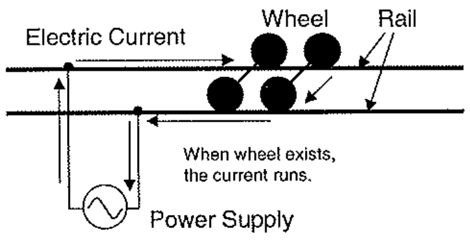


Figure 3: Track Circuit

There may be switches in the area detected by track circuits. On track circuits, paths are defined as a pair of the boundaries to specify from where to where trains can proceed. For example when a track circuit with a switch is shown as Figure 4, one path can be defined as a pair of (1,2) and another path can be defined as a pair of (1,3). To handle the track circuits uniformly, paths are defined in case there is no switch. In this case there is only one path.

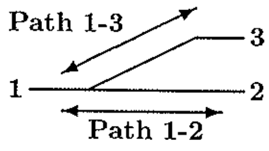


Figure 4: Track Circuit with one switch

On top of track circuits and paths, routes are defined as sequences of paths. Routes express how trains can go along. All of the defined paths are connected according to the definition of routes. Each route has also a unique ID and the IDs are sent to trains in real time to indicate how the train can go along.

The track has speed limitations in relation with curve and track conditions. And gradients of slopes are also an important factor to calculate the braking curve. These properties are also registered in the database.

Among these properties there are many relationships. These relationships must have consistency. We verify the consistency.

3 Formal Analysis

3.1 Writing Formal Specification

Before formal analysis, we need to write a formal specification from informal specification. We described the specification with a formal specification language VDM-SL. First, the basic components such as track circuits and routes are modelled as types. Track circuits are modelled as follows:

```
TrackC:: joints : map Joint_id to Joint
         atc : map Direction to ATC
         td : TD
         atbt : ATBT
```

Joint in the first line expresses the boundary of the track circuit. Two track circuits are connected via Joint. In the second line, ATC signal is defined with a direction. ATC signal is always sent to the front of the train. So, in general, the ATC signals are different between two directions of train movement. TD in the third line is the abbreviation of train detection signal which is used on some track circuits. ATBT in the last line means AT or BT. These show whether the train detection signal is sent from the front of the train or the rear of the train.

Then, we add invariants to the types. Invariant is the constraint to the types. Any values with a certain type should keep the invariants at any time. Invariants are very strong constraint. If the safety requirements are included as invariants, they should be kept at any time. The track circuit type, for example, has following invariants:

```
inv mk_TrackC(joints, atc, td, atbt) ==
   card dom joints > 1 and
   dom atc = {<ADIR>, <BDIR>} and
   TD_Used_NonInsulated_TC(td, atbt, rng joints) and

   (atc(<ADIR>).used and atc(<BDIR>.used) =>
       atc(<ADIR>).carrier <> atc(<BDIR>).carrier)
```

In the second line, it is defined that the number of the joints is more than one. In the fourth line, a predicate TD_Used_NonInsulated_TC is used. This means that train detection signals are used only on the non-insulated track circuits, which are track circuits without any electric insulation at the joints. A predicate is a function to return a boolean value. The use of predicates makes the specification easy to read.

After the structure of the database is defined, we should add the small set of the operations. Here is an example for the operation to add (register) a new track circuit into the database:

```
Add_TrackC : Area * TrackC_id * TrackC -> Area
Add_TrackC(ar, tcid, tc) ==
    mu(ar, trackcs |-> ar.trackcs ++ {tcid |-> tc})
pre tcid not in set dom ar.trackcs and
    forall jid in set dom tc.joints &
      Only_One_Next_TrackC(...) and
      forall tcid1 in set dom ar.trackcs &
        Joint_and_Next_TrackC_Consistent(...)
```

The part following pre defines the precondition, which defines the restrictions among the parameters to keep the calculation consistent. Whenever this function is used, the precondition must be satisfied.

After that postconditions are defined to verify the relationship between input parameters and return value. The specification reaches 985 lines of VDM-SL code.

Post condition is the requirement between the parameter of the fucntion and return value. This is an extension for standardized VDM-SL. VDM-SL has two types of function: implicit and explicit. In explicit functions in which function body is written, post condition is somewhat extra things. In this case it is explicit, but post condition can be added.

Postcondition of Add_TrackC is as follows:

```
post    tcid in set dom RESULT.trackcs and
        RESULT.trackcs = ar.trackcs ++ {tcid |-> tc} and
        RESULT.trackcs(tcid) = tc and
        RESULT.paths = ar.paths and
        RESULT.routes = ar.routes;
```

You can find that the post condition is somewhat identical to the function body. So it may seem strange. But the concept is that if the post condition is written, the function body can be changed as long as the postcondition is satisfied.

3.2 Generating Proof Obligation

When the specification is written in formal specification languages, proof obligations can be generated. These are the requirements to verify the consistency of the specification. For the specification used here, 188 proof obligations are generated.

Proof obligations are classified into many classes. One of the most difficult class is invariant type, which are generated for any values. Most typical obligations are generated for the return values of functions. For example, for the previously appeared Add_TrackC, following proof obligation is generated:

```
forall ar: Area, tcid: TrackC_id, tc: TrackC &
   pre_Add_TrackC(ar, tcid, tc) =>
     inv_Area(Add_TrackC(ar, tcid, tc))
```

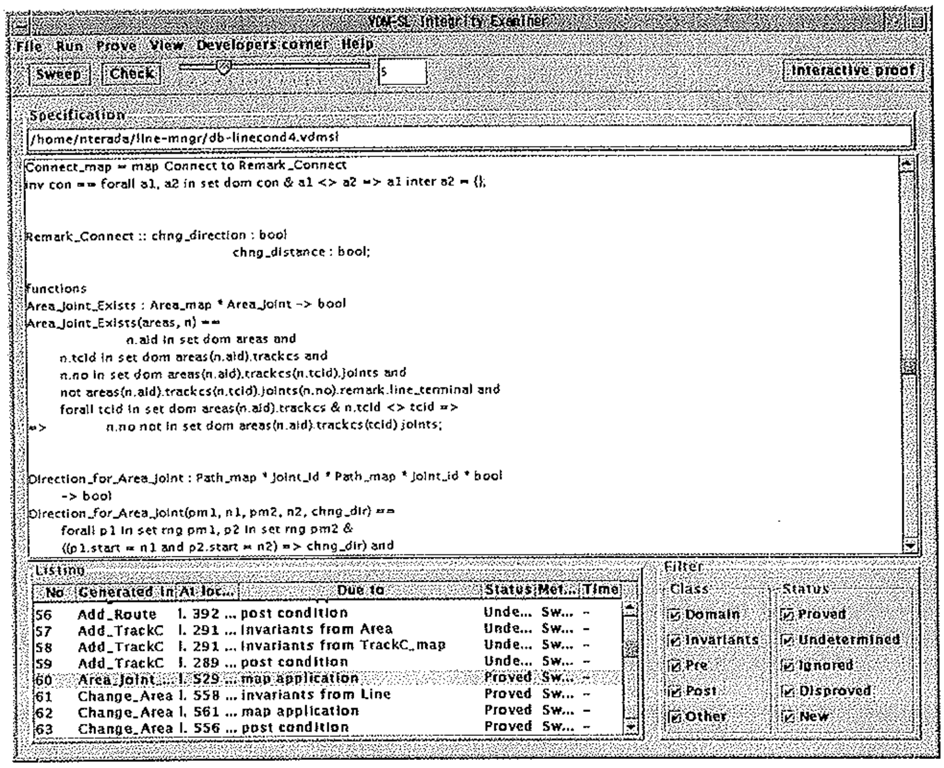
This means that if the precondition of Add_TrackC is satisfied, return value must satisfy the invariants of Area type. Notice that forall is used. This is the requirement for all of the possible values. So the proof obligations are strong requirement and difficult to prove.

Much more proof obligations are classified as domain types. This requires that the domain must be defined at any time. For example, there is a function Line_Add_TrackC which calls Add_TrackC and defined as follows:

```
Line_Add_TrackC : Line * Area_id * TrackC_id * TrackC -> Line
Line_Add_TrackC(ln, aid, tcid, tc) ==
        mu(ln, areas |-> ln.areas ++
                {aid |-> Add_TrackC(ln.areas(aid), tcid, tc)})
pre     aid in set dom ln.areas and
        pre_Add_TrackC(ln.areas(aid), tcid, tc) and
        ...
post    ...
```

Since Add_TrackC is defined with a precondition, we must check that the parameters satisfy the precondition whenever Add_TrackC is called. Then following proof obligation is generated for the call of Add_TrackC

```
forall ln : Line, aid : Area_id, tcid : TrackC_id, tc : TrackC &
```


VDM-SL Integrity Examiner
File Run Prove View Developers corner Help
Sweep
Check
5
Interactive proof
Specification
Listing
Filter
Class
Status

/home/nterada/line-mngr/db-linecond4.vdmsl

```
Connect_map = map Connect to Remark_Connect
inv con == forall a1, a2 in set dom con & a1 <> a2 => a1 inter a2 = {};


Remark_Connect :: chng_direction : bool
                   chng_distance : bool;

functions
Area_Joint_Exists : Area_map * Area_Joint -> bool
Area_Joint_Exists(areas, n) ==
           n.aid in set dom areas and
    n.tcid in set dom areas(n.aid).trackcs and
    n.no in set dom areas(n.aid).trackcs(n.tcid).joints and
    not areas(n.aid).trackcs(n.tcid).joints(n.no).remark.line_terminal and
    forall tcid in set dom areas(n.aid).trackcs & n.tcid <> tcid =>
=>         n.no not in set dom areas(n.aid).trackcs(tcid).joints;


Direction_for_Area_Joint : Path_map * Joint_Id * Path_map * Joint_id * bool
    -> bool
Direction_for_Area_Joint(pm1, n1, pm2, n2, chng_dir) ==
    forall p1 in set rng pm1, p2 in set rng pm2 &
    ((p1.start = n1 and p2.start = n2) => chng_dir) and
```

No	Generated In	At loc...	Due to	Status	Met...	Time
56	Add_Route	l. 392 ...	post condition	Unde...	Sw...	-
57	Add_TrackC	l. 291 ...	invariants from Area	Unde...	Sw...	-
58	Add_TrackC	l. 291 ...	invariants from TrackC_map	Unde...	Sw...	-
59	Add_TrackC	l. 289 ...	post condition	Unde...	Sw...	-
60	Area_Joint_...	l. 529 ...	map application	Proved	Sw...	-
61	Change_Area	l. 558 ...	invariants from Line	Proved	Sw...	-
62	Change_Area	l. 561 ...	map application	Proved	Sw...	-
63	Change_Area	l. 556 ...	post condition	Proved	Sw...	-

Class	Status
✓ Domain	✓ Proved
✓ Invariants	✓ Undetermined
✓ Pre	✓ Ignored
✓ Post	✓ Disproved
✓ Other	✓ New

Figure 5: Snapshot of GUI

```
pre_Line_Add_TrackC(ln, aid, tcid, tc) ==>
    pre_Add_TrackC(ln.areas(aid), tcid, tc)
```

When a function is applied, the precondition of the function must be satisfied. In many cases it is sufficient to include the requirements into the context and they are proved easily.

The proof obligations can be examined by hand. If a proof obligation found false, it means the inconsistency of the specification so it needs to be modified. In the beginning we checked the specification by hand and found some mistakes without automatic proofs. The proof obligation generator provides very powerful support even without automatic proof support. Of course if the automatic proof support is available, we can save time to check the proof obligations.

3.3 Automatic Proofs

The specification and its proof obligations are loaded into the proof engine. This engine is based on HOL(High Order Logic)[9]. The PROSPER project developed an API, called PROSPER Toolkit. The Core Proof Engine of the

PROSPER Toolkit is primitive, but the proof engine can be customized by adding HOL theories, rewriting rules, and proof management and user defined modules. The engine used here is customized for proving VDM-SL specifications.

Automatic proofs are very easy to use. We just need to click a button to execute automatic proofs. Two kinds of full automatic proof procedures are available. The first one checks the proof obligations quickly but gives up fast. There are a lot of proof obligations easy to prove, so many proof obligations are swept out. The other checks more deeply and takes time. Some of the proof obligations are proved with this.

Even if the proof is not finished, the subgoal left undetermined are reported, which helps us to prove by hand or finding mistakes. For the reasons why automatic proof is not successful, please refer section 4. Even if many proof obligations are left undetermined, the number of them are much reduced from that of the original and we can concentrate on the undetermined obligations.

We found mistakes through this tool. For example, if a precondition is short of some predicates to keep invariants, the subgoal says that the conclusion includes some predicate which are not included in the assumption. In another case, the same subgoal remains undetermined among several proof obligations. After the modification of the mistakes, 167(90%) of the proof obligations are proved automatically.

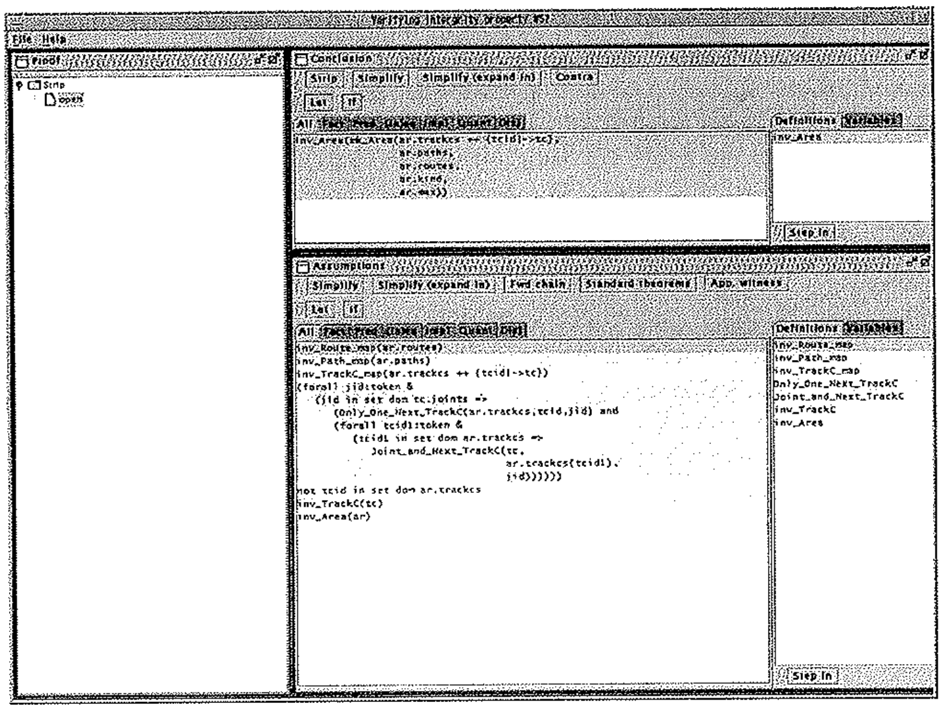
3.4 Interactive Proofs

Interactive proof support is available for the undetermined cases. Figure 6 is a snapshot of GUI for proving the proof obligation shown in section 3.2. The proof is called goal oriented proof. The goals to be proved are split into small subgoals and when all of the subgoals are proved, the original goal is proved.

The other procedure includes unfolding functions, moving left hand of the implication in the conclusion into assumption, combining some assumptions into a new assumption, case splits, etc. In fact, the automatic proof procedure consists of such procedures with the automatic selection. For some precise procedure, please refer Table 1. We can select these procedures by clicking a button on the GUI. This GUI is intended to provide proof facility to those who are not so expert in the proofs. The history of the proof is displayed as the proof tree in the left side of the GUI.

Although the GUI is very easy to use, it is still difficult to do some proofs. Sometimes the proof tree reaches 100 steps and it takes several hours to prove the obligation. To select proper buttons, we should not only get used to the tool, but also well understand the specification itself. We must consider well this if we are to succeed.

Although the interactive proofs are time consuming, the result is very powerful. We proved all of the remaining proof obligations. This means that the proof obligations are proved fully mechanically.



```
inv_Area(mk_Area(ar.trackcs ++ {tcid|->tc},
                 ar.paths,
                 ar.routes,
                 ar.kind,
                 ar.max))
```

```
inv_Area
```

```
inv_Route_map(ar.routes)
inv_Path_map(ar.paths)
inv_TrackC_map(ar.trackcs ++ {tcid|->tc})
(forall jid:token &
   (jid in set dom tc.joints =>
      (Only_One_Next_TrackC(ar.trackcs,tcid,jid) and
      (forall tcid1:token &
         (tcid1 in set dom ar.trackcs =>
            Joint_and_Next_TrackC(tc,
                                  ar.trackcs(tcid1),
                                  jid))))))
not tcid in set dom ar.trackcs
inv_TrackC(tc)
inv_Area(ar)
```

```
inv_Route_map
inv_Path_map
inv_TrackC_map
Only_One_Next_TrackC
Joint_and_Next_TrackC
inv_TrackC
inv_Area
```

Figure 6: Snapshot of GUI for Interactive Proof

3.5 How to Handle Validation Conjectures?

What we proved here is the consistency of the specification. But how can we handle validation conjectures, which are not obligations, but expected to be true? Here is an example of validation conjecture:

```
Del_TrackC(Add_TrackC(ar, tcid, tc), tcid) = ar
```

This means that when a new track circuit with ID tcid is added, then the track circuit is removed again, the data is the same as before. The user interface for handling the validation conjecture is not provided. But the automatic prove is available to add a following predicate to the specification.

```
Conj_Add_and_Del_TrackC :
    Area * TrackC_id * TrackC -> bool
Conj_Add_and_Del_TrackC(ar, tcid, tc) ==
    Del_TrackC(Add_TrackC(ar, tcid, tc), tcid) = ar
pre pre_Add_TrackC(ar, tcid, tc)
post RESULT = true;
```

splitting of the goal	$\dfrac{A \vdash B \wedge C}{A \vdash B \wedge A \vdash C}$
Implication	$\dfrac{A \vdash B \Rightarrow C}{A, B \vdash C}$
for all	$\dfrac{\forall X : C \cdot f(X), Y : C \vdash A_1, A_2, ..., A_n}{f(Y), \forall X : C \cdot f(X), Y : C \vdash A_1, A_2, ..., A_n}$

Table 1: Tactic of the Interactive Proof

Then the following proof obligation is generated in relation with postcondition:

```
forall ar : Area, tcid : TrackC_id, tc : TrackC &
   pre_Add_TrackC(ar, tcid, tc) ==>
    ((Del_TrackC(Add_TrackC(ar, tcid, tc), tcid) = ar)
        = true)
```

If this is proved true, the validation conjecture is true. If you want something to be proved other than proof obligations, it is handled in the same way. This is not so convenient, so more sophisticated tool is required. In fact it is difficult to make GUI to handle these things easy. At least it is very difficult to input the things to prove only by simple buttons.

4 Experiences – Hints to Make Proofs Easier

There are several reasons why the automatic proofs are not successful. In the following sections we show some experiences about the difficulty. This gives us large helps to make proofs easier.

4.1 To Find Counter Examples

The proof engine cannot disprove the obligations. When a proof obligation is false, the automatic engine classifies it as undetermined. We must decide whether it should be disproved or simply difficult to prove. The easiest way to disprove it is to find a counter example because most of the proof obligations are written beginning with `forall`. Only one counter example is enough to disprove this. We should find it by hand, but we can use interpreter to check the counter example violates the invariants, which is very clear disproof. When there is a mistake in the specification, counter examples often help us to find the mistake.

4.2 To Express the Same Thing in a Same Way

When sequence is used, the use of indexes of the sequence causes a problem. Let's consider two expressions. The first one is:

```
forall i in set inds a & f(a(i))                                    (1)
```

where `inds a` means index set of the sequence a and a(i) is the i-th element of a. The other is:

```
forall x in set elems a & f(x)                                      (2)
```

where `elems a` means the collection of all elements of the sequence a. These two expressions mean the same thing, but they are understood in different ways. First one is understood as a sentence of variable x, while the other is understood as a sentence of integer i. Suppose a goal in which the assumption is (1) and the conclusion is (2). Then this is rewrited with the following assumptions:

```
x in set elems a
forall i in set inds a & f(a(i))
```

and the following conclusion:

```
f(x)
```

If the proof engine can find that

```
exists i in set inds a & a(i) = x
```

and substitute a(i) of f(a(i)) with x, this goal can be proved. But it is difficult to do so.

The flexibility is a factor whether it can be written easily. But on the other hand this makes the automatic proofs difficult. In the practical point of view, therefore, if some parts means the same thing, all of them should be written in the same manners and the common part should be written as an additional function. This is important for human as well because it makes the specification readable.

4.3 Prevent Too Deep Tree

When a new assumption is added by combining some assumptions, all of the possible combinations are tried. Automatic prove cannot choose a specific combination. As a result, assumptions get larger fast and it takes time to prove. In some case, this process goes into infinity loop. To prevent this situation, automatic proof is interrupted. If the proof needs to be unfolded deeper, it is hard to be proved.

Although in the previous section we described that the specification should be written smart with functions, the use of the functions sometimes makes the proofs difficult. This is a contradictory fact. We should try make the relationship of the functions simple.

4.4 Contradiction

There are some undetermined cases, in which the contradiction will be satisfied. When

$$a_1, a_2, ..., a_n \vdash f(a_1, a_2, ..., a_n)$$

is to be proved, we can prove it by disproving

$$\overline{f}(a_1, a_2, ...a_n), a_1, a_2, ..., a_n$$

The feature to handle these situations has been added to the GUI. But this procedure is difficult do by automatically.

5 Conclusion

We analyzed the specification of digital ATC track database with formal methods and automatic proofs. The specification was proved consistent fully mechanically.

The proof obligations have very strong property so automatic generation of them provides help us finding mistakes in the specification even without automatic proofs.

The automatic proof support is much more powerful. In this analysis 90% of proof obligation are proved automatically. Even the proof obligations are not proved so much, the tool provides us information of undetermined proof obligations and helped us finding mistakes.

The interactive proof support GUI achieved some points from the point of view of user friendly. But it took a lot of time to succeed in proofs. It is still difficult to succeed in interactive proofs because the good knowledge of the specification is required. We should write formal specification easy to read both for human and computer. In spite of the difficulty, we could interactively prove all of the proof obligation undetermined by the automatic proofs. We could prove the consistency of the specification.

Although the tool still needs a lot of improvement in both proof engine and GUI, we are satisfied with the analysis as an experimental project. And we hope that the analysis with formal methods and automatic proofs become more common method in the near future.

Acknowledgments

This work was done as a cooperative work between RTRI and IFAD. We express great thanks to Peter Gorm Larsen and Kim Sunesen, who helped and commented us our work and gave us a chance to use proof supports.

References

[1] J. Fitzgerald, P. G. Larsen: Modelling Systems – Practical Tools and Techniques in Software Development, Cambridge University Press, 1998

[2] G. Guiho and C. Hennebert: SACEM Software Validation, Proceeding of 12th International Conference on Software Engineering, IEEE Computer Society Press, 1990

[3] Abrial J-R: The B-Book: Assigning programs to meanings, Cambridge University Press, 1998

[4] A. Faivre, P. Benoit: Safety critical software of meteor developed with the B formal method and vital coded processor, Proceeding of WCRR99, Tokyo, 1999

[5] G. Stålmarck and M. Säflund: Modelling and Verifying Systems and Software in Propositional Logic, Proceeding of SAFECOMP'90, Pergamon Press, 1990

[6] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham: The PROSPER Toolkit. Proceedings of TACAS 2000, pp. 78-92. LNCS 1785, Springer-Verlag, 2000

[7] T. Takashige and I. Watanabe: Development of the digital ATC for high speed and high density area. Report of RTRI, Vol. 9, No.1, 1995

[8] T. Ogino, Y. Hirao: The interest in formal methods from Japanese perspective, Proceeding of FMERail workshop #5, FM'99, Toulouse, 1999

[9] M. J. C. Gordon and T. F. Melham: Introduction to HOL: A theorem proving environment for higher order logic, Cambridge University Press, 1993

[10] N. Terada: Formal Integrity Analysis of Digital ATC Database, Proceeding of WCRR2001, Cologne, 2001