Safety checking in an automatic train operation system

G.K. Palshikar

Tata Research Development and Design Centre (TRDDC), 54B, Hadapsar Industrial Estate, Pune 411013, India

Received 27 June 2000; revised 15 November 2000; accepted 16 November 2000

Abstract

Keeping the trains and tracks in a safe state is important for railway systems, which include automated control. ATO-2000 is an automated railway system that plans, operates, monitors and controls a small railway network of driver-less trains within a mine. The formal specifications, design and implementation of Checker Function (CF), a software sub-system responsible for maintaining safety in ATO-2000 are described. CF is an important component in a safety-critical, real-time, distributed, mobile computing system. The formal specifications (in Z) of the core safety requirements in ATO-2000 are presented which include a new representation of the track topology. Some fault tolerance of the data received from the field is achieved through data validation constraints. Command safety constraints conservatively validate outgoing commands so that no possible future system state is unsafe. A simple approach used to integrate formal methods in the industrial software development process is discussed. The paper concludes with a review of the lessons learnt. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Railway systems; Z; Real-time safety-critical systems; Lightweight formal methods; Data validation; Fault tolerance

1. Introduction

Many real-time safety critical systems that have a central controller behave in a periodic (or cyclic) manner. The first part of every cycle is spent in gathering data from field devices. Since there may be faults in the field devices and communication media, some fault tolerance is achieved indirectly by means of data validation to identify errors, inconsistencies and other problems in the reported data. Then appropriate commands for the field devices are computed. These commands need to be validated for safety before they are actually issued, so that any possible future system state is not unsafe or dangerous. Typically, this command safety validation is done by forecasting the effects of the commands to be issued on the current system-state. A command is rejected if any one of the possible future states is unsafe. Effects of commands in a command set may interfere or conflict; individually safe commands may be unsafe when issued together. Also, methods for computing future states and the criteria for safe states must be well defined.

1.1. The ATO-2000 system

ATO-2000 is an example of such a cyclic system having a central controller and involving both data and command validation. The ATO-2000 system aims to automate the operations of trains within a mine; similar small autonomous railway systems can also be found elsewhere, e.g. within a factory complex or docks. The system consists of about 8.5 mile of tracks divided into two main tracks and several sidings. There are up to nine electrically powered goods trains of up to 20 cars each. Each train picks up ore rocks from one of the 3 pockets (which are storage points where the ore rocks dug from the mine tunnels are collected), transports (and unloads) the ore rocks into a crusher complex. The trains are unmanned (i.e. driver-less) and automatic, with an on-board computer (OBC). The track-side equipment include tunnel doors, more than 100 track-circuits to detect track occupancy, about 36 power sensors, about 24 switches to enable trains to change tracks, etc. The track is controlled by 10 Wayside Interface Units (WIU), each of which consists of a small computer (mainly to operate the track-side devices) and an operator console. A human operator can use a WIU to observe the status of the part of the track controlled by the WIU and to issue commands, if necessary.

The Railway Control System (RCS) is the central controller that observes, plans, controls and manages the trains and the track-side equipment so as to meet operational goals like ore mix, productivity and throughput, without compromising on safety requirements. Each train and WIU sends its current status to RCS every second (via a radio-based network). Train status includes current position, direction and speed (as computed on-board). WIU status includes...
switch and door positions, occupancy indicated by track-circuits, etc. Upon receiving this field data, RCS then computes the next course of action for each train and new values for track-side equipment and dispatches the necessary commands to the trains and WIUs.

The ATO-2000 system has a tight 1-second system-wide operational cycle, making it a hard real-time system. This 1-second cycle is divided into several sequential sub-frames that include: receive status messages from trains and WIU, process the received messages and generate new commands, validate the generated commands, send validated commands to the trains and WIU. The ATO-2000 system also has stringent operational safety requirements. It contains spatially distributed mobile units (trains) communicating over a TCP/IP based radio network. A variety of faults can happen in various components in ATO-2000. In this paper, we are concerned with only one sub-system of the ATO-2000 system, the so-called Checker Function (CF) sub-system, which is described next.

1.2. The checker function sub-system

RCS includes a sub-system called the CF, which provides safety management services to RCS by performing the following tasks: (a) validation of the incoming status data sent every second by all the WIU and trains; and (b) ensuring that the commands sent out by RCS to the track-side equipment and OBC do not lead to unsafe conditions in the field.

CF is an important component in a safety-critical, real-time, distributed, mobile computing system. CF is designed and implemented as a functionally and physically independent system. The RCS and CF are connected via TCP/IP over Ethernet. The trains, human operators and WIU computers communicate with each other and RCS via a TCP/IP wide-area network over radio. Every 1 s, CF receives the following: (a) a set of data input messages from the trains indicating their current state (location, speed, direction, etc.); (b) a set of data input messages from WIU indicating the current state of track-side devices (switch and door positions, occupancy reported by track-circuits, power status, etc.); (c) a set of commands from human operators; and (d) a set of commands from RCS to trains and WIU.

CF validates the received data input messages by checking them against specified data validation constraints. CF validates the received commands by checking them against specified command validation constraints. As a result of this checking, CF notifies the RCS regarding: (a) errors, if any, in the current railroad-state as indicated by the received data input messages; and (b) whether it is safe to issue each of the given commands. If CF detects any problems in the data input messages, then it continues to use the old status (rather than guess the real status in the field outside) for at most two consecutive cycles. If CF rejects any command (or if CF fails to validate it within a fixed time limit), then RCS does not forward the command to that unit. If the unit fails to receive a command from RCS for more than two consecutive cycles, then it has a default fail-safe action; e.g. if the unit is a train, then it starts braking and ignores all further commands till it comes to a complete halt.

CF needs to handle several faults that may arise in practice, like missing, duplicate, late, corrupted or out of order messages. For example, if CF fails to receive the data input message from a train (say) for one cycle, then it uses the last cycle’s data and if it fails to receive it for two consecutive cycles, CF rejects any command to that unit. It is important to note that CF does not attempt to reconstruct the system state (in the face of missing data input message from a train, for instance). Thus, CF does not employ rollback or roll-forward fault tolerance in any strict sense. CF simply continues to use the old state for 2 s (cycles) and thereafter (if the arrival of the messages has not resumed) marks the source unit (train or WIU) as out of communication so that no more commands are sent to it. This decision was based on the perception that it is difficult to reconstruct the system state with known bounds on the errors (e.g. error in the location of a moving train which goes out of communication) and that the risk in using the reconstructed state increases with time. The approach chosen here is simpler and more suitable for fast real time implementation which is needed. However, the 2 s latency may be too short and detrimental to the throughput of the ATO-2000 system, depending on how often the messages are missed in practice. For instance, an output of communication train is brought to a halt before new commands can be sent to it, even if it resumes sending status messages after missing them for, say, three cycles.

The importance of formal methods in safety-critical systems is well recognized. We motivate the choice of the Z notation used in this project. Sets, relations, functions, sequences and partial orders are fundamental mathematical structures as well as important data types in terms of which solutions of many computational problems can be expressed. Mixing these mathematical structures with typed first order predicate logic leads to a rich notation, called Z [15] to express formal specifications of complex software systems. A number of other formal notations like VDM-SL, B and Standard ML share these qualities. In addition, Z also provides the schema calculus, which helps in organizing the formal specifications of a system in a modular and hierarchical manner and also helps to improve the reusability of the specification components. Z is a standardized notation and has been widely used as a formal specification notation in various real-life industrial projects. Z is supported by a number of special tools such as type-checkers, pretty-printers, simulators and code generators. Z is also amenable for refinement, although refinement was not used in this project. The mathematical types in Z are particularly suitable to describe track topology as well as logical safety constraints.

In this paper, we focus on the functional safety requirements of the ATO-2000 system rather than dynamic behavioral,
real time timing requirements nor any user interface requirements (there is very little of it in CF). In particular, the Z specifications do not capture: (a) the application level protocol of message exchange (and associated fault conditions) between CF and RCS as well as CF and trains and WIUs; and (b) any real-time requirements and timing properties of the CF sub-system; e.g. those related to the time limits on the generation of CF responses. The motives for this decision were as follows: (a) stretching the Z notation to include timing properties (which are dynamic and behavioral in nature) would have distorted the natural functional structure of the Z specification; and (b) most (but certainly not all) of the important safety constraints were functional in nature.

1.3. Related work

We now briefly describe some related work. The importance of formal methods in safety critical systems and in particular in railway systems is well recognized. The proceedings of several special workshops (available at http://www.ifad.dk/Projects/merail.html) devoted to the use of Formal Methods in railway applications are a good source for the current work in the field. Another source for the relevant literature is the bibliography at http://liinw-w.w.ira.uka.de/bibliography/SE/rail.html for applications of Formal Methods in Railway domain. Much work is concerned with modeling the railway signaling and interlocking systems and their verification. For instance, the work of Hansen [5] describes VDM models of Danish railway interlocking systems containing graph oriented representations of track topology (somewhat similar to the one in this paper) and logical statements of safety constraints. This work also uses fault trees augmented with duration calculus to model dynamic safety properties of the system; it is also a valuable source of relevant literature. Ref. [6] discusses the use of VDM in Japanese railway signaling systems. Ref. [1] describes an application where NP-tools were used to automatically verify that an interlocking program (specified in a special-purpose interlocking programming language Stermol) satisfies the safety constraints (specified in first order predicate logic). Cimatti et al. [2] have used the SPIN model-checking tool to verify a reactive interlocking program (specified in a special purpose Safety Logic), and Wong [17] and Moniget [11] deal with formal representation of track topology as graphs and in logic, respectively. The SACEM project [4] in France describes the use of formal methods in a full software development life cycle for a train speed control application in Paris Metro. There is considerable interest [3,10,12] in understanding the role of formal methods in industrial scale development methodologies. The Z specification notation has been widely used in practical systems and in particular in safety-critical systems; Jacky [7] and Johnson [8] describe case studies where Z specifications were used in safety-critical systems. Paul Amman has also used Z to express requirements of a traffic control system. Ref. [9] contains thorough discussions of the basic issues and techniques for safety analysis and requirements specifications for safety-critical systems. It is a commonly held practical view that most safety-critical systems have a safety kernel (as expressed by John Rushby) that needs to be analyzed and developed with much greater care and the CF corresponds to such a safety kernel in the ATO-2000 system.

The prototyping of the Z specifications was an important aspect of the CF requirements analysis phase. Z is an abstract specification notation without any direct relationship with a model of execution. Since Z allows working with infinite sets as well as partial and non-deterministic specifications, it is possible to write Z specifications that cannot be executed directly. However, the Z notation is sufficiently close to logic programming and functional programming languages, so that a number of attempts have been made to (manually) translate Z specifications to such languages for execution. In Ref. [16] two approaches to animate Z specifications using Prolog are discussed: (a) formal program synthesis; and (b) structure simulation. They argue about the limitations of the first approach, and recommend the later approach, which is also the basis of the work in this paper. However, they do not describe a complete tool to do a faithful animation of Z facilities. In Ref. [14], a similar approach is followed, except that it describes a (manual) translation of Z specifications into the functional programming language Haskell. Rudimentary set management predicates are discussed in Ref. [13]. However, they are insufficient to be directly usable for prototyping Z specifications since they lack facilities for implicit set definitions, sets of sets as well as specific Z operators to manipulate these objects.

The rest of the paper is organized as follows. Section 2 contains the formal specification in Z of some of the core safety requirements in CF, using the Z notation [15]. Section 3 analyses some system-wide safety properties and the role of CF in maintaining the overall ATO-2000 system in a safe state. Section 4 describes an approach used for the design and implementation of CF and its relationship with the formal specifications. Some lessons learnt from this project are discussed in this section. Section 5 presents the conclusions.

2. Formal specifications of safety requirements

CF maintains two types of data: static and dynamic. Static data includes relatively unchanging information; e.g. the track layout, number, and type and placement of track-side devices. Dynamic data changes frequently during the system’s operation; e.g. speed and location of trains, and current values of various tracks-side devices. The system-state includes both static and dynamic data. We now discuss a formal model of both these types of data. Formal specifications of the data validation and command safety constraints (presented later) make use of the formal data model developed here. We use the Z notation for these tasks. For brevity, we give only fragments, not complete
specifications, in this paper. The specifications here are also
simplified from the actual system specifications, so as to
allow a focus on the essential safety issues in the system.
These simplifications and omissions are often described in
the paper by saying  

\textit{we ignore .} \ 

2.1. Track layout

We describe a novel method of modeling track layout
and illustrate it using the simple track in Fig. 1. Rather than
as a graph, we describe the track as a binary relation.
Various domain concepts associated with the track (e.g.
route) are formally specified based on this model of
the track layout. A physical track is continuous. We \textit{discretize}
the track by introducing track points. A \textit{track point} is a
logical marker (of zero physical width) assumed to be
present on the track. Its only purpose is to serve as a marker
for dividing the track. The set \textit{TRACK\_POINT} consists of
all track points in the system; for Fig. 1, this set is enumerated
as:

\text{TRACK\_POINT} = \{a, b, c, d, e, f, g, h, i, j, k, l, c', j'\}

in the \textit{normal} position, the train continues on the current
track. When the switch is in the \textit{reverse} position, the train
changes track. For the train to complete such a \textit{crossover},
the switches at both ends of the crossover track must be in
the reverse position. A \textit{siding} may have only one switch for
entry and exit of the train; some sidings may of course have
two separate switches for entry and exit. In addition, due to
the physical structure of the tracks and trains, a switch
places restrictions on the permitted changes of tracks. For
example, in Fig. 1, a train on track 1 coming from \(f\) and
moving southward cannot use the switch to cross over to
track 2. The track layout has to naturally reflect these
geometrical restrictions.

To handle this, we place two \textit{co-located} track points at
each switch (\(c, c'\) and \(j, j'\) in Fig. 1). The track points \(c\)
and \(j\) (\(c'\) and \(j'\)) correspond to the normal (reverse)
position of the corresponding switches. It is easy to check that the follow-
ing irreflexive and anti-symmetric binary relation layout (on
the set \textit{TRACK\_POINT}) reflects all the physical restrictions
on the movement of trains on the tracks in Fig. 1. The track
interval (\(c'j'\)) denotes a \textit{crossover track interval}. We adopt
the convention that if a pair of track points \((x,y)\) belongs to
layout, then \(x\) is more southward than \(y\).

\begin{align*}
\text{layout} & : \text{TRACK\_POINT} \leftrightarrow \text{TRACK\_POINT} \\
\text{layout} & = \{(a,b),(b,c),(b,c'),(c,d),(d,e),(e,f),(g,h),(h,i),(i,j),(j,k),(k,l),(c',j'),(j'k)\} \land \\
\text{irreflexive layout} & \land \text{antisymmetric layout}
\end{align*}

Each tuple of adjacent track points denotes a physical
\textit{track interval} on the track; e.g. the track portion \((b,c)\)
between the track points \(b\) and \(c\) is a track interval and so
is \((j,k)\). There may be zero, one or more track points
associated with a physical point on the track. A device called
\textit{switch} (or \textit{point} as it is known in Europe and India) is used
to allow a moving train to change track. When the switch is

We assume the axiomatic definitions of certain useful
predicates and functions. The predicate contiguous: \texttt{iseq}:
\text{TRACK\_POINT} \to \text{BOOLEAN} \text{ is \textit{true} if the given non-
empty sequence \(S\) is a valid route. A \textit{route} is a sequence
of track points ordered from south to north and contiguous
as per the given track layout; e.g. in Fig. 1, contiguous
\((\langle b,c,j'k\rangle) = \text{true} \text{ and contiguous}(\langle b,c,j,k\rangle) = \text{false}.)
The predicate northward: TRACK_POINT \times TRACK_POINT \rightarrow BOOLEAN is true if the first track point is more northwards than the second track point; predicate southward is defined similarly. The function route_len: TRACK_POINT \rightarrow \mathbb{N} associates a length (in meters) with every track interval. Function route_len: iseq TRACK_POINT \rightarrow \mathbb{N} returns the length in meters of the given route; e.g. in Fig. 1, route_len((a, b, c', j, k)) = 2850 m. Note that the function route_len requires the entire route — and not just start and end track points of the route — since in general there may be several valid routes of differing lengths between the two given track points.

2.2. Track-side devices

A number of devices of different types are placed along the track. The set DEVICE gives unique ID for each device in the system. The set DEVICE_TYPE lists the different types of devices in the system. Function device_type: DEVICE \rightarrow DEVICE_TYPE associates a device type with every device in DEVICE. A door controls access to a portion of track. A train can pass through a door only if it is open. A track circuit is a device that is spread over a contiguous portion of a track and reports whether or not that portion is occupied by some object; it can also detect and report when a part of the physical track is broken. The set DEVICE_VALUE defines the values that devices can have. We ignore the additional check that the value of a device is consistent with its type; e.g. a switch cannot have a value occupied. We get such consistency properties for free with notations like VDM SL which are more strongly typed. The value ooc denotes the out-of-correspondence situation, when a switch is neither normal nor reverse (or a door is neither open nor closed), as happens when the device is in transition to change its current value to a new value.

DEVICE = = \{s1, s2, d1, o1, o2, o3, o4, o5, o6, o7\}
DEVICE_TYPE = = \{switch, door, tc\}
DEVICE_VALUE = = \{normal, reverse, ooc, open, closed, occupied, unoccupied, broken_rail\}

Each device is associated with one or more physical points on the track, given by the function d2tp: DEVICE \rightarrow F; TRACK_POINT. We require that at least one track point be placed at the physical location of each device on the track. A device will be associated with all the track points co-located at the physical location of the device; e.g. in Fig. 1, the switch s1 is associated with two track points c and c' and door d1 (at track points e and k) controls the access to the track on either side. A track circuit is spread over a contiguous region of track and all the track points within this track region are associated with the track circuit. For example, the track circuit o2 is spread over the contiguous region of track given by the sequence of track points (b, c, d).

A simple predicate schema TPControl specifies the logical condition associated with each track point, which has a device associated with it. If this condition evaluates to true for a given track point, then it is safe for a train to move over the track point; otherwise, the train should not be allowed to proceed across that track point. In Fig. 1, for a track point like c (c') the condition evaluates to true if the current value of the switch is normal (reverse). For the track points e and k, the condition evaluates to true if the door d1 there is open. If the given track point has no device associated with it, the schema evaluates to true for that track point.

2.3. Trains

The set TRAIN consists of the trains that can occupy the track. Every train has associated with it a direction of movement from the set TRAIN_DIR. The constant max_speed indicates the maximum speed (in meter per second) with which a train can be commanded to move; of course, it is possible that a train may report its speed to be higher than max_speed. In addition, there are speed limits that can be associated with parts of track, which we ignore for the sake of simplicity. The partial function safe_braking_distance: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} indicates the maximum distance (in meters) required for a train to stop if the current train speed is within a specific range. A partial function braking_distance: \mathbb{N} \rightarrow \mathbb{N} can be easily defined using the partial function safe_braking_distance to return the braking distance required for a train moving at the given speed: e.g. braking_distance(23) = 1000 m if safe_braking_distance(20, 30) = 1000.

TRAIN = = \{t1, t2, t3, t4, t5, t6, t7, t8, t9\}
TRAIN_DIR = = \{northward, southward, stopped\}

The location of a train end-point is a tuple (TP, d) where TP is the southward track point of the track interval within which the train's end-point falls and d is the distance of the end-point from TP (in meters). At each instant, the location of an entire train is given as two locations: north location and south location i.e. the locations of the northward and southward end-points of the train. Note that the locomotive may be at the south end-point of the train even if the train is moving northward, as when the locomotive is pushing rather than pulling the train. For example, in Fig. 1, if the
south and north locations of a train are (b,1000) and (c,250) then (i) the train’s southern end-point is located in the interval (b,c) and it is 1000 m away from the track point b and (ii) the train’s northern end-point is in the interval (c,d) and it is 250 m away from the track point c. It can be computed, using the function ti_length, that the distance between these two locations, i.e. the observed train length, is 450 m. We omit the simple check that d is less than the length of the corresponding track interval.

```
TrackState
device_value : DEVICE → DEVICE_VALUE
ti_state : layout → F TL_STATE
device_cmd_value : DEVICE → DEVICE_VALUE

-- if a track circuit reports occupied then the associated track is marked so
∀d:dom device_type • device_value(d) = occupied ⇒
(∀p,q:TRACK_POINT | {p,q} ∈ layout ∧ {p,q} ⊆ d2tp(d) • occupied ∈ ti_state(p→q) = true)

-- several other integrity checks for validating the track state
```

We assume axiomatic definition of some more useful predicates and functions. The predicate northward_loc: (TRACK_POINT × N) × (TRACK_POINT × N) → BOOLEAN is true if the first location is north of the second location. The predicate loc_within_route: (TRACK_POINT × N) × iseq_l TRACK_POINT → BOOLEAN is true if the given location is within the given route. The partial function distance: (TRACK_POINT × N) × (TRACK_POINT × N) × iseq_l TRACK_POINT → N returns the distance between the given two locations, both of which are within the given route. We assume that the first location is more southward than the second. We omit the actual axiomatic definition of this function. The constant max_length_uncertainty indicates the maximum dynamic variation (in meters) expected in the train length due to various operating conditions (e.g. stretching or shrinking of the train).

2.4. Dynamic track state

TL_STATE is the set of the possible status values associated with a track interval. The value ‘blocked’ states whether any operator has blocked the track interval, say for repairs. The dynamic state of the track-side devices and track intervals is given by the schema TrackState. CF periodically updates the values for devices and track interval status. We ignore the operational modes for the various devices, e.g. under manual command, under WIU command or under central RCS command. We also need to detect and record the faulty status for devices — a task ignored here for simplicity. Usually, faulty devices cannot be trusted or manipulated. We also ignore the question of how to handle missing status (value) from devices. Since there can be considerable time lag between commanding the device to a value and the device actually achieving that value, commanded values for devices are also recorded.

```
TL_STATE = { power_on, blocked, occupied, broken_rail }
```

Simple schema can be defined to update the dynamic track state, as recorded in the schema TrackState, as per the incoming data input messages received from trains and WIU.

2.5. Dynamic train state

The frame schema Φ TrainState represents and records the dynamic status of each train as maintained by CF. The occupancy is not actually reported by the train, but it is calculated by CF from reported locations and track circuit data. The enforcement authority (EA) of a train refers to the part of the track cleared and allocated for the train; it is specified as a sequence of track points ordered from south to north. Only the device values are reported by WIU to CF; the track-interval status is computed by the CF. Overall state containing reported values is defined by the schema ReportedSystemState.

```
ΦTrainState

cars : N
dir : TRAIN_DIR
north_loc, south_loc : TRACK_POINT × N
speed : N
occupancy : iseq_l TRACK_POINT
cmd_dir : TRAIN_DIR
cmd_speed : N
cmd_EA : iseq_l TRACK_POINT
```
TrainState Δ [train_state : TRAINS → ⌜TrainState]

The overall state of the ATO-2000 system, as seen by CF, is defined as:

SystemState Δ TrainState ∧ TrackState.

The reported state is actually given by the ReportedSystemState schema, which is only partially filled by decoding the incoming data input messages (since some fields of the TrainState and TrackState schema are computed by CF). However, we ignore this complication.

ReportedSystemState Δ TrainState? ∧ TrackState?

2.6. Data validation

Data validation consists of identifying any inconsistencies in the reported state of the system. Some examples of such rules, called data validation constraints, for validation of the data received from the train and WTU are given below. Note that some of these data validation constraints need to compare the reported system state with previous system states (i.e. with system history). For each train:

1. The reported north location should really be more northward than the reported south location.
2. All the contiguous track segments from the reported south location till the reported north location should be reported as occupied and all these segments should be included in the reported train occupancy.
3. The south and north locations should be within the reported train occupancy.
4. The distance between the reported south and north locations should be the same as the train length, within some permitted limit (to account for stretching of a train, for example).
5. The length of its reported occupancy should be more than the train's length.
6. The reported train occupancy should be consistent with the reported values for the switches within the occupancy.
7. If the train is moving, then the reported train locations should be displaced (with respect to the previous train locations) in the reported direction of train movement. For example, if L1s and L1n (respectively L2s and L2n) are the previous (respectively currently reported) south and north train locations and if the currently reported train speed is non-zero and if currently reported train direction is northward, then L2n should be more northward than L1n (similarly for L2s and L1s). Moreover, the distance between L1n and L2n (respectively, L1s and L2s) should approximately equal to reported train speed × time interval elapsed between previous and last train data.

We represent each data validation constraint by a schema. The input variables indicate the new (reported) state data about trains, devices and track. As an example, the following schema specifies one of data validation constraints stated above. In addition, we have restricted the scope of the constraint by requiring that constraint can be applied to a train only if constraints 1 and 3 are satisfied (i.e. they are pre-conditions or prerequisites for this constraint). The abs function stands for the absolute value (modulus) function. We have assumed that length of each car is 15 m and that of a locomotive is 30 m.

```prolog
∀t : TRAIN |
- Reported north location is indeed more northward than reported south location
northward_loc(train_state?(t).north_loc, train_state?(t).south_loc) ∧

- Reported train locations are within reported (contiguous) train occupancy
loc_within_route(train_state?(t).north_loc, train_state?(t).occupancy) ∧
loc_within_route(train_state?(t).south_loc, train_state?(t).occupancy)

- Distance between reported train locations is close to known train length
• abs(distance(train_state?(t).south_loc, train_state?(t).north_loc, train_state?(t).occupancy)
  - train_state?(t).cars * 15 + 30) ≤ max_length_uncertainty
```
2.7. Command safety validation

RCS can send commands to various units in ATO-2000 (e.g., trains and WIU). When a unit executes a command, it causes changes in the state of trains, track intervals or track-side devices. Examples of commands include changing the speed of a train, changing the value of a switch or a door and blocking a portion of track. Frame schema \( \Phi_{\text{CmdTrainState}} \) and \( \Phi_{\text{CmdTrackState}} \), respectively, represent a command to a train or a WIU and contain the new commanded values for the trains, devices and track segments. We assume that RCS always computes commands for all trains and devices in each cycle.

\[
\text{WIU} = \{ w1, w2, w3, w4, w5, w6, w7, w8, w9, w10 \}
\]

1. The commanded EA of each train is contiguous and contains the train's current occupancy.
2. No track interval included in the commanded EA of any train should be blocked, occupied or contain a broken rail.
3. The commanded EA of all trains in the system are mutually non-overlapping (except perhaps for the end-points).
4. The switches within the commanded EA of each train are correctly aligned.
5. The doors within the commanded EA of each train are open.
6. The commanded and current EA of a train are consistent with the reported direction of movement of the train.
7. A device to which a command is to be issued must be either a switch or a door; moreover, it should not be included in the newly commanded or current EA of any train.
8. A track segment to which a command is to be issued is not included in the newly commanded or current EA of any train. Note that this restriction holds even if the command is unblocking a track segment.
9. For each train, the commanded EA is long enough, in the direction of train movement, to allow the train to stop within the EA.
10. For each train, the commanded speed is \( \leq \text{max\_speed\_limit} \).

\[
\begin{align*}
\Phi_{\text{CmdTrainState}} & \\
\text{cmd\_dir} : \text{TRAIN\_DIR} & \\
\text{cmd\_speed} : \mathbb{N} & \\
\text{cmd\_EA} : \text{iseq\_} \text{TRACK\_POINT} & \\
\text{cmd\_speed} \leq \text{max\_speed} & \land \\
\text{cmd\_speed} > 0 \Rightarrow \text{cmd\_dir} \neq \text{stopped} & \\
\end{align*}
\]

\[
\begin{align*}
\Phi_{\text{CmdTrackState}} & \\
\text{cmd\_device\_value} : \text{DEVICE} & \rightarrow \text{DEVICE\_VALUE} & \\
\text{cmd\_ti\_state} : \text{layout} & \rightarrow \text{FTI\_STATE} & \\
\end{align*}
\]

\[
\begin{align*}
\text{CmdTrains} & \triangleq \{ \text{cmd\_trains} : \text{TRAINS} \rightarrow \Phi_{\text{CmdTrainState}} \} & \\
\text{CmdWIU} & \triangleq \{ \text{cmd\_wiu} : \text{WIU} \rightarrow \Phi_{\text{CmdTrackState}} \} & \\
\text{CmdState} & \triangleq \text{CmdTrains} \land \text{CmdWIU} & \\
\end{align*}
\]

We now formally specify a number of restrictions, called command safety constraints, on the commands to be sent to trains or WIUs in every cycle. Every cycle, CF checks that the set of commands for trains and WIUs (as prepared by RCS) satisfies each safety constraint. Some examples of the safety constraints are as follows.

\[
\begin{align*}
\exists \text{SystemState} & \\
\text{CmdState} \end{align*}
\]

\[
\begin{align*}
\forall w : \text{WIU}; d : \text{DEVICE} & \bullet w \in \text{dom} \ \text{cmd\_wiu} \land d \in \text{dom} \ \text{cmd\_wiu}(w).\text{cmd\_device\_value} \land \\
(\text{device\_type}(d) = \text{switch} \lor \text{device\_type}(d) = \text{door}) \land \\
(\text{device\_value}(d) \neq \text{cmd\_wiu}(w).\text{cmd\_device\_value}(d)) & \land \\
\neg (\exists \text{tp} : \text{TRACK\_POINT}; t : \text{dom} \ \text{cmd\_trains} \bullet \text{tp} \in \text{d2tp}(d) \land \text{tp} \in \text{cmd\_trains}(t).\text{cmd\_EA} ) & \\
\end{align*}
\]
11. For each pair of adjacent trains, their EAs should be separated by minimum 40 m.

We model each safety constraint as a Z schema. For example, the above schema specifies the constraint 7 above.

3. Safety and fault tolerance in ATO-2000

Safety in the entire ATO-2000 system is a complex issue. There are many kinds of unsafe situations that may occur in the ATO-2000 system. For example, train collision, derailment, mechanical failures (e.g. broken rail, overheated bearings or axle dragging), device failures (e.g. door failing to open), speed limit violations, brake failure, train entering into closed tunnels or blocked tracks, mis-aligned switches causing derailment or unexpected change of track for a train, unexpected/uncommanded train movement (e.g. rollback on hill slope), unexpected objects on track (e.g. rocks), fire in WITU, etc. We ignore possible unsafe situations in the loading and unloading activities in pockets and crusher.

Many of these unsafe situations happen due to failures of the individual hardware devices or communications network. For this reason, many components in ATO-2000 have individual fault-tolerance facilities, either based on simple replication or corrective measures. For example, each track circuit occupancy sensor is duplicated and is required to report a fail-safe value. Each train location is computed in two different ways — from a count of wheel rotations and by recognizing radio-tags placed along the tracks. Sequence numbers and double checksums are used to identify old and corrupted data and command messages. The on-board software stops the train if it misses commands from RCS for more than two cycles or if the commanded speed is more than known speed limits.

However, in spite of some fault-tolerance in the individual field units, there is still a possibility of them reporting incorrect status. It is the purpose of the data validation constraints in CF to validate the status data received from devices and trains. The first question in the safety analysis, from the point of view of CF, is as follows. An important class of unsafe situations arises when the status of the ATO-2000 system as reconstructed and maintained by CF is significantly different from the actual real status in the field. For example, the values of switches or doors, or train speeds and locations may be different in CF and in reality. Two reasons why this can happen are missing messages or time delays for message transfers. In addition, there are some physical reasons why this can happen. In switch chattering, the switch value may be wrongly reported as out-of-correspondence due to the vibrations caused by a train passing nearby. Even if we ignore these, there is still a need to prove that the CF always has a reasonably correct (or up-to-date) picture of the outside reality.

Due to the complex and multiple faults that can cause distortion in the CF view of the system reality, it is not possible prove that the CF will always have a reliable view of the system status. For example, if a train loses communication with CF, CF does not have the correct location data for it and has to rely on the track circuit occupancy readings to prevent accidents. We have, however, performed extensive simulations to test the data validation constraints in CF.

Similar questions arise about the formalization of the role of CF in maintaining the ATO-2000 system in a safe state. We look at the question of preventing train collision. Suppose that the all the data truly reflects the outside reality of the system state, that there are no missing or delayed messages and that no data or command validation checks are violated at any instant. Even ignoring mechanical and brake failures and assuming the appropriateness of the safe braking distance table, other reasons why the system may end in a collision include: CF has incorrect location/speed of one or more train or occupancy data is incorrect (for some instants). If there are no distortions in reported data or faults of any other kind, then of course, CF prevents the overlap of EA for any pair of trains. However, we have not formally modeled the situations containing faults and their effects in terms of data distortions. So, we have not attempted a formal proof that the CF command safety constraints are able to prevent train collisions under faults. We have, however, performed extensive simulations to test the safety validation constraints in CF.

4. Development process for CF

4.1. Safety requirements analysis

The safety requirements in ATO-2000 to be implemented by CF were prepared in two sequentially organized phases: user requirements specification (URS) written in English and formal specifications written using the Z notation. First, the URS was prepared based on a series of interviews with the railway engineers and systems safety specialists. Preliminary hazard analysis and some level of fault trees were also prepared during this process to gain insight into the safety requirements. The URS was reviewed by the railway safety experts, systems engineers and the CF/RCS software development teams. The URS underwent a series of revisions to take into account the review comments at each stage. A final version of the URS was then reached and approved.

Once the URS was approved, the formal specifications of the safety requirements were prepared. The process of the preparation of the formal requirements for the CF system included the following steps:

1. **Data model.** In this stage, the global data store maintained and used by the CF system was studied, its structure was identified and integrity constraints for the data were identified and formally specified. As mentioned earlier, this data was of two types: (a) static data regarding track layout,
number, type and placement of devices, etc.; and (b) dynamic data regarding speed and location of trains, values of various devices, etc. The data model also included a formal definition of the interface data, i.e. the data reported by trains and WIUs as well as the commands sent by RCS. The steps in data model definition roughly followed this sequence: system constants (e.g. speed limits), given and other basic sets (e.g. TRACK_POINT), basic data types (e.g. layout) and operations on them, axiomatic definitions of required functions (e.g. routelen), hierarchical groups of data items (e.g. train data and device data) along with operations to manipulate them (e.g. update train state), integrity constraints on data items and finally, definition of the global ATO-2000 system state as reconstructed and maintained by CF.

2. Data validation constraints. In this stage, each data validation constraint was formally defined.

3. Command safety validation constraints. In this stage, each command safety validation constraint was formally defined.

4. Prototyping. In this stage, the formal specifications in Z were manually translated into the ZLOG model and prototyped using the ZLOG tool developed in-house by the author. The ZLOG specifications were extensively tested under a wide range of test conditions. In fact, the realization of the value added by the ability to prototype the formal specifications (using the ZLOG tool) was an important aspect of the project. The ZLOG prototyping provided invaluable feedback and helped gain the user's confidence.

5. Reviews and Analysis. Both the URS and formal specifications were reviewed and approved; system safety specialists and developers were both trained in the Z notation so that they could directly review the Z specifications. There was no attempt to formally prove the correctness and completeness of the safety constraints; e.g. to prove impossibility of collisions or guaranteed detection of specific problems in the reported data.

4.2. Prototyping Z specifications

We now present some examples of how the Z specifications were prototyped and tested using the ZLOG tool. ZLOG specifications can be incrementally developed, tested, queried, analyzed for properties and executed as prototypes. ZLOG helps to interactively improve problem understanding and detect at the earliest functional shortcomings — a frequent reason causing delayed client acceptance and large overheads in rework.

The ZLOG tool can be used to prototype and interactively execute formal specifications written in Z. Specifications written in Z need to be manually translated into executable ZLOG programs. ZLOG is essentially a portable Prolog toolkit for uniform definition and symbolic manipulation of mathematical types (e.g. sets, relations, functions, and sequences) of Z in the logic programming framework of Prolog. ZLOG also provides dynamic type checking and operators in the Z schema calculus. ZLOG supports both explicit and implicit definitions of sets and ZLOG operators work for both representations. ZLOG provides a uniform treatment of the mathematical objects in Z; e.g. a function can also be manipulated as a relation or as a set as necessary. By imposing the following simple structural restrictions on Z specifications, we can bring the Z specifications 'sufficiently' close to Prolog's execution model underlying ZLOG.

- Given (i.e. unspecified) sets are not permitted; all sets must be defined.
- Infinite sets are not permitted; all sets must be finite.
- The Z specifications must be divided into a common data store and schema.

In addition, the logical conditions in the Z specifications may be written with a more operational approach so as to keep the translation to ZLOG easier. Also, we have divided the Z specifications into a common global data-store and schema that manipulate this data-store so as to correspond to the common division of Prolog programs into facts and predicates.

The sets in the Z specification presented in this paper are easily defined in ZLOG as:

```prolog
set 'TRACK_POINT' :: =
\{a,b,c,d,e,f,g,h,i,j,k,l,c,l,j\}.
set layout :: = \{(t(a,b), t(b,c), t(b,c1), t(c,d),
\{d,e\}, t(e,f), t(g,h), t(h,i), t(i,j), t(j,k),
t(k,l), t(c1,j1), t(j1,k)\}.
set ti_length :: =
\{(t(c,a), 400), t(c,b,1200), t(c1,b,c1), 1200),
\{(t(c,d), 2000), t(c,d,e), 3000), t(e,f), 1400),
\{(g,h), 800), t(h,i), 600), t(i,j), 5000),
\{(j,k), 1200), t(k,l,450), t(c1,j1), 50),
\{(j1,k), 1200),\}.
set 'DEVICE' :: =
\{d1, d2, d3, o1, o2, o3, a4, a5, o6, o7\}.
set 'DEVICE_TYPE' :: =
\{switch, door, tc\}.
set 'DEVICE_VALUE' :: =
\{normal, reverse, ooc, open, closed, occupied, unoccupied, broken_rail\}.
set device_type :: = \{(t(a1, switch), \{t2, \{switch\},
\{d1, door\}, \{o1, tc\}, \{o2, tc\}, \{o3, tc\},
\{o4, tc\}, \{o5, tc\}, \{o6, tc\}, \{o7, tc\},
\{d2, tp\} :: =
\{t(a1, {c1,c2}, \{t2, \{j1\}, \{d1, e,k\},
\{t1, a,b\}, \{t2, b,c,d\}, \{o3, d,e,f\},
\{o4, c1,j1\}, \{t06, \{g,h,i\}, \{t06, i,j,k\},
\{t07, (k,l)\}.
set 'TRAIN' :: = \{(t1, t2, t3)\}.
set 'TRAIN_DIR' :: =
\{northward, southward, stopped\}.```
set safe_braking_distance ::=
{t(t(0,10), 300), t(t(10,20), 700), t(t(20,30), 1000),
t(t(30,40), 1500), t(t(40,50), 2300), t(t(50,60), 3000),
t(t(60,100), 10,000)}.  
set 'WIU' ::=  
{w1, w2, w3, w4, w5, w6, w7, w8, w9, w10}.

It is easy to check some elementary properties of these sets. For example,

?- dom (set d2tp) seteq (set 'DEVICE').  
  yes
?- ran (set device_type) subset (set 'DEVICE-TYPE').  
  yes
?- irreflexive (set layout), antisymmetric (set layout).  
  yes
?- transitive (set layout).  
  no

The axiomatic Z definitions of constants, predicates and functions as well as Z schema can be easily defined as a Prolog predicate in ZLOG and they can be tested by executing with different parameter values. Each Z predicate has two parameters: list of input variables with types; and list of output variables with types. A variable with type has the form VariableName: Type, where VariableName is an atom containing the name of the variable and Type is a Z type expression; e.g. 'TP': set 'TRACK_POINT'. Note that the parameters to the ZLOG predicate contiguous are typed, whereas usual Prolog predicates have un-typed parameters. For example, the predicate contiguous defined in Z can be defined in ZLOG as given below (this predicate does not have any output parameters).

contiguous([Seq; seq set 'TRACK_POINT'], [1]):-  
  zeval (seqlen(Seq) >= 2, true),  
  zeval (seqlen(Seq) - 1, true),  
  forall([J: set 1..N1] true, t(seq_element  
          (Seq,J), seq_element (Seq,J + 1) belongs_to_set  
          layout)),  
  true.

ZLOG provides a special predicate zcall, which can be used to execute a Z predicate. The values for input parameters are passed in the first list and Prolog variables to hold the output values are passed in the second list. It is easy to test the above predicate using the track layout; e.g.

?- zcall (contiguous([b,c,l,j,l,k]), []).  
  yes
?- zcall (contiguous([b,c,l,j,k]), []).  
  no
?- zcall (contiguous([b,c,l,j,l,l,k]), []).  
  Type-check failed! Type=seq set TRACK_POINT;  
  Value=[b,c,l,j,l,k].

A Z schema can also be easily defined as a Prolog predicate in ZLOG and they can be tested by executing in different data states. Each Z schema has three parameters: list of the names of schema included in this schema, list of input variables with types, list of output variables with types. A variable with type has the form VariableName: Type, where VariableName is an atom containing the name of the variable and Type is a Z type expression; e.g. 'TP': set 'TRACK_POINT'. For example, the schema 'TrainLocationMatchesTrainLength' is defined in ZLOG as follows (this schema does not include any schema and has no input/output parameters):

'trainLocationMatchesTrainLength'([], [], []).  
forall ([T: set 'TRAIN'],  
true,
  t(T, S1) belongs_to set reported_train_state(_),  
%
reported north location is more northward than  
reported south location  
arg(4, S1, t(TP1,D1)), % north location =  
(TP1,D1)  
arg(3, S1, t(TP2,D2)), % south location =  
(TP2,D2)  
zcall (northward_loc([TP1,D1, TP2,D2],[])),  
%
reported train locations are within reported contiguous occupancy  
arg(6, S1, Occupancy),  
zcall (contiguous([Occupancy],[])),  
zcall (loc_within_route([TP1, D1, Occupancy], [])),  
zcall (loc_within_route([TP2, D2, Occupancy], [])),  
%
distance between reported train locations is  
close to known  
%
train length  
arg(1, S1, NumCars), % no. of cars in trainT  
zcall (distance([TP2, D2, TP1, D1, Occupancy], []), Len1),  
abs(Len1 - (NumCars * 15) + 30, Delta),  
max_length_uncertainty(U),  
Delta = <U  
], []).  
%
}.

ZLOG provides a special predicate zcall, which can be used to execute a Z schema predicate. The values for input parameters are passed in the first list and Prolog variables to hold the output values are passed in the second list. It is easy to test the above schema when executing as follows:

?- zcall ('TrainLocationMatchesTrainLength'([], [])),  
  yes
4.3. Guidelines for development of formal specifications

While the use of formal methods is gaining acceptance in software industry, a need is felt for practical guidelines about making the best use of the formal specifications technology. During this project, we identified a few pragmatic tips for people involved in the industrial use of formal specifications. As a general model, we assume that a systems analyst is responsible for producing the informal URS document whereas a specification engineer is required to prepare formal specifications based on the URS. More generally, the specification engineer is also entrusted with the job of making effective use of the formal specifications technology within a given project. Towards that end, it is important to construct formal specifications, which are readable, well structured, reusable, validated and correlated with the URS.

We do not describe a methodology (in the form of steps, milestones, deliverables, etc.) to develop formal specifications and to adopt a software development life-cycle process to incorporate formal specifications. Instead, we discuss a set of useful, practical guidelines to the specification engineer. These guidelines are divided into two classes: (i) for the process of developing and using formal specifications; and (ii) for contents of formal specifications themselves and are summarized below. These guidelines are not exact nor are they the only ones; their only purpose is to highlight some essential aspects of the process of development of formal specifications. The specification engineer will get a better appreciation of these guidelines (and perhaps add a few of his own), only when he consciously applies them while actually developing formal specifications.

Process of developing and using formal specifications

1. Clearly define the role of formal specifications in your software development process, addressing the issues of users, intended uses, training, change management, reviews, validation, traceability etc.
2. Define the steps in the process of developing formal specifications.
3. Carefully divide specifications among broad classes like functionality, operations, behavior and interface.
4. Select the appropriate parts (classes of requirements, modules, ...) of your application for formal specifications.
5. Choose appropriate formal specification notations and tools for each class of requirements.

Contents of formal specifications

6. Maintain high level of abstraction by avoiding design decisions (data structures, tables, algorithms, ...) and implementation details (classes, messages, ...).
7. Avoid over and under specifications as well as non-deterministic and partial specifications.
8. Build modular specification containing 'good' and 'natural' structure.
9. Always try to come up with many alternate representations and then choose the one best suited for the problem.
11. Be true to the spirit of the notation: focus on declarative (what) aspects rather than behavioral aspects (how) when using Z.
12. Review and test the specifications thoroughly; document the test cases.
13. Document the specifications well and provide English explanations.
14. Plan and make effective use of available tools.
15. State and prove (or argue about or at least demonstrate) all the necessary properties of the specification.

4.4. Design and implementation of CF

Design and implementation of CF followed a standard software development life cycle and did not use any formal methods. In this sense, the project followed what has been called the lightweight formal methods approach. The formal techniques were used only to write precise descriptions of the safety requirements; the rest of the project methodology concentrated on faithfully but manually implementing the formal specifications. Moreover, the formal specifications were restricted to a safety kernel that concentrated on validation of field data and outgoing commands.

In the high-level design stage, CF was split into several modules. The communications manager module in CF dealt with the receiving, sending, buffering and translating messages over TCP/IP. These messages included the data input messages from the trains and WIs, commands from RCS for the trains and field devices as well as control messages exchanged between RCS and CF (e.g. responses of CF regarding results of data and command validations, and health-status messages). A constraint manager dynamically executed constraints based on a fixed precedence structure (specified as a directed precedence graph) among the constraints; e.g. if the EA allocated by the RCS for a train was not contiguous, then several other constraints were not checked for that train. A miscellaneous functions module handled CF activities like bookkeeping, logging etc. Requirements for none of these modules were not specified formally.

The validation manager module in CF implemented the data structures for representing the track layout as well as current and historical dynamic data regarding devices, trains etc. It also contained algorithms to implement the data validation and command validation constraints, which were formally specified. During low-level design of the validation manager, the data structures and algorithms were manually derived from and manually validated against the Z specifications.

The CF code was written in C and implemented as a single process on the QNX real-time OS. The code was
profited and analyzed for timing issues and improved as per the findings.

Extensive simulations of realistic traffic and failure scenarios, using a specially built train and track simulator, were used to validate the final CF implementation. The simulator could be programmed to simulate a wide range of operational scenarios. The simulator, when running, generated and sent to CF the data input and command input messages, exactly as if they were coming from real trains, WTUs and RCS. The timing response of CF was measured and was satisfactory. Currently, CF is being integrated with RCS and thereafter the system is soon expected to go into trial runs. Note that these simulations are performed on the final implemented CF C code and they are different from the validation of the Z specifications using the prototyping facilities of the ZLOG tool.

It is intended that the source code and the formal specifications will be maintained together under an integrated change management procedure, including versioning. For this purpose, special attention was paid during the coding stage to keep the source code of the validation manager module as close as possible to their formal specifications. Several measures were defined to document and relate this code against the formal specifications. For example, there is (almost) one-to-one correspondence between each Z schema and the related C function in the source code. Names of the functions and variables are carefully related to those of the schemas and the variables therein. Source code of the safety constraints contains comments that relate the functions to the corresponding Z specifications. Comments within a function relate to assertions within the corresponding Z schema. Cross-reference tables were built to quickly relate the formal specifications to the corresponding C code fragments. Such tricks considerably enhanced the review of the code against the formal specifications. It has to be noted, however, that this is a rather weak and in fact informal approach to traceability. Additional rigorous measures, properly supported by appropriate tools, like those reported in the literature, could have definitely improved the traceability. However, this approach was considered adequate considering the moderate size of the formal specifications as well as source code, as also because there were sizeable CF components that were not formally specified.

4.5. Evaluation of the formal specifications’ contribution

The project has definitely demonstrated the feasibility as well as value of using lightweight formal methods as part of the industrial software development process. The formal specifications were definitely more concise and of course very precise compared to the URS written in English. The process of formalization of the requirements brought clarity and rigour to the statement of requirements. The ability to prototype the formal specifications was also appreciated; it helped in identifying problems with the requirements and also improved the confidence in the stated requirements. At a broader level, ability to test the formal specifications against realistic scenarios in fact helped to understand many issues regarding the exact nature of the safety in the system. The formal specifications were also used to annotate and comment the code; this practice considerably eased the task of manual code inspection to check conformance with requirements. It is intended that both the code and the formal specifications will be maintained in tandem so that they closely correspond to each other.

Unfortunately, we have no quantitative measurements for the gains obtained by employing formal methods (e.g. savings in costs or efforts). However, the general feeling is that it has helped improve the clarity, accuracy and coverage of the requirements. There were no excessive penalties in terms of time and efforts spent in the requirements analysis stage.

Some concrete gains obtained by using formal methods are as follows. A concise and efficient relational representation of the track topology was defined formally and this allowed a natural but formal definition of many domain concepts (e.g. route and EA). This representation was sufficiently efficient so that algorithms for implementing individual constraints were directly defined and implemented using this representation. Moreover, the formal specified parts of the ATO-2000 project were observed to be much more stable throughout the project and underwent much less number of changes compared to other requirements (e.g. message interfaces) which were not formally specified.

5. Conclusions

This paper reported the use of formal specifications in small automatic train operation system ATO-2000. ATO-2000 is a safety-critical, real-time, distributed, mobile computing system. The formal specifications, design and implementation of the safety constraints in ATO-2000 were described. A new method for track topology representation was described. The logical notation Z was successfully used (rather than behavioral notations based on state machines or temporal logic) to formally model data and safety constraints.

The lessons learned from this exercise can be summarized as follows. The use of formal specifications definitely helped to clarify and even enhance the safety requirements stated in English. Type-checking and prototyping of formal specifications helped in quick and interactive validation of formal specifications in a variety of situations, particularly because the complexity of multiple faults made it impossible to formally prove the adequacy of CF functionality in maintaining safety. After training, end users and design teams were able to read and comment upon the Z specifications. Good specification practices were drawn up as a result of this interaction. All this helped in the use of Z specifications as a single reference point during development. The approach of restricting formal specifications only to a
well-identified critical functionality proved useful in controlling the complexity and improving the utility of the formal methods approach. Simulation tools were useful to test the final CF software, by programming realistic scenarios, including creation of faults.

The project demonstrated that with proper support tools, training and restriction to core functionality, lightweight formal methods can be integrated in mainstream industrial software development processes and can improve the quality of the software produced. Sometimes, it is not necessary to go fully formal.

Acknowledgements

I wish to thank Prof. Mathai Joseph for his perceptive guidance and encouragement throughout this project. Thanks are also due to Prof. K.V. Nori, Prof. S. Ramesh, Dr Purandar Bhaduri, Nambi Ganesh, Aditya Nori and the entire project team for stimulating discussions. Special thanks are due to Charlie Gipson for his constant and enthusiastic support for formal methods. I appreciate the thorough reviews and suggestions from the reviewers. I would like to sincerely thank Dr Manasee Palshikar for her patience, hopes and confidence.

References