# Specification and Test Case Generation for the Safety Kernel of the Naples Subway[′]

**Antonio Casazza (+), Dario Comini (*), Angelo Morzenti (o),**
**Matteo Pradella (o), Pierluigi San Pietro (o), Fabio Schreiber (o),**
**(o) Dipartimento di Elettronica, Politecnico di Milano**
**(*) Metropolitana Milanese**
**(+) Ansaldo Segnalamento Ferroviario**

**Proceedings of**
**World Multiconference on**
**Systemics, Cybernetics and Informatics**
**and**
**5th International Conference on**
**Information Systems Analysis and Synthesis**

**Orlando (Fl.), 31/7 - 4/8, 1999**

**Vol. 1, pp. 533-540**

′ Work partially supported by MURST, Progetto Scienza e Tecnica per la Qualitá della Mobilitá Locale.

# Specification and Test Case Generation for the Safety Kernel of the Naples Subway'

**Antonio Casazza (+), Dario Comini (*), Angelo Morzenti (o), Matteo Pradella (o), Pierluigi San Pietro (o), Fabio Schreiber (o),**
**(o) Dipartimento di Elettronica, Politecnico di Milano**
**(*) Metropolitana Milanese**
**(+) Ansaldo Segnalamento Ferroviario**

## ABSTRACT

We report on an experience in the application of formal methods to the specification, validation and verification of a railway signaling system: the safety kernel of the Naples Subway. The activity was performed *ex post*, several years after final system delivery, based on the design documentation [4, 1]. We first illustrate the requirement specification using the object-oriented temporal logic TRIO. Then we relate on the use of the specification, by means of suitable support tools, to validate the requirements and to generate some scenarios to be employed as test cases in the verification phase.

**Keywords:** Safety-critical systems, Validation and Verification, Testing, Formal Methods Application.

## 1. INTRODUCTION

The design and construction of the signaling system in the Naples subway presented an interesting integration among a few conventional components, such as the ground apparatus that realizes command and control functions for signals and switches, and some more innovative ones, such as the ground apparatus that determine the free/blocked track status and imposes the minimum distance among trains. These features differentiate it from other plants constructed until then in Italy, as they were realized using microprocessor-based technology. In particular, the safety critical functions of Automatic Train Block were accomplished through an ATIS (Audiofrequency Transmission ad Interlocking System), a system whose main functions are: train detection on the tracks, check of track integrity, and computation and transmission to the trains of the information regarding the state of the signaling devices, to allow the on board instrumentation to regulate the train running.
The ATIS was structured into a central component called Topological Interface, which was connected, through a fiber optic network, to a set of Peripheral Posts displaced on the tracks. These act as an interface to the track circuits, which constitute the medium for information transmission to the trains. The component of the Topological Interface that selects the codes to be sent to the trains for each automatic block section or for each track circuit, and that communicates with the Peripheral Posts is called the Safety Kernel.
The present paper reports on the modeling of the Safety Kernel through a specification written in the formal specification language TRIO [3], and on the production of functional test cases based on the TRIO model, for the purpose of validation of the specification itself and for implementation verification. The activity was in fact performed after the Naples Subway had been constructed, validated, and in operation since several years. This was an experiment in the framework of a cooperation between Metropolitana Milanese and Politecnico di Milano in the application of formal methods to the specification, validation and verification of signaling systems for trains and subways.
The specification is not so strictly related to the Naples plant, since it is parametric with respect to the plant topology. Thus, it can be easily reused to model different plants by means of a simple redefinition of the specification module describing the plant topology.
The specification is based on some simplifying assumptions, which however do not limit its completeness and generality. In particular, we ruled out some of the information that the Topological Interface sends to the track circuits (the transmission frequency of the next track, the braking profile and the remaining section length), which can however easily be added. The considered aspects concern the running direction, the section entrance and exit speed, which are assumed to be the same for all the track circuits of a given section. In other words, from the viewpoint of the Topological Interface we do not distinguish the various circuits inside the same section. These simplifying assumptions can be easily removed whenever necessary.
The document is structured as follows: Section 2 presents a brief overview of the TRIO formalism; Section 3 includes the TRIO specification of the Safety Kernel; Section 4 describes the test case generation activities carried out on the specification.

## 2. THE SPECIFICATION LANGUAGE TRIO

TRIO is based on classical predicate calculus, extended with temporal operators to refer to time instants different from the current one, which is left implicit in the formula. To allow a specifier to describe time related entities, TRIO variables, functions, and predicates are divided into *Time Dependent* (TD) and *Time Independent* ones. Time dependent variables represent physical quantities or configurations that are subject to change with time; time independent variables represent quantities or configurations that are unrelated with time. Time dependent functions and predicates denote relations, properties or events that may or may not hold at a given time instant, while time independent functions and predicates represent facts and properties that are assumed not to change with time. TRIO is a typed language, in that every variable is associated to the domain of the values that it can assume, every function is associated with a domain/range pair, and a domain is associate to each of the arguments of every predicate. Among the domains there exists a distinguished one, called the *Time Domain*, that is numeric in nature: it can be, for instance, the set of integers, real, or rational numbers. We assume as predefined, for all numerical domains (and hence for the Time Domain) all the functions representing the common arithmetic operations, such as +, -, *, /, DIV, MOD, etc., and time independent predicates representing common relational operators, such as =, ≠, <, ≤.
Besides variables, functions, and predicates TRIO includes the propositional operators, '~' (NOT), '–>' (IMPLIES), '&' (AND), '|' (OR), '≡', 'XOR', and the quantifiers 'EXISTS' and 'FORALL'. TRIO formulas can also be composed by using primitive and derived temporal operators, as explained next.

### Primitive Operators

There are two temporal primitive operators, *Futr* and *Past*, that allow one to refer, respectively, to events occurring in the future or in the past with respect to the current time, which is left implicit in the formula. For any given formula *F*, the composed formulas Futr(*F*, *t*) and Past(*F*, *t*) hold at the current time if and only if the property denoted by *F* holds at the instant *t* time units after (respectively, before) the current time.

### A First Example

Let us consider a simple railway track, where a train enters at one end and exits at the opposite end after a given maximum time, say 10 seconds. The event of train arrival at the entrance end is

' Work partially supported by MURST, Progetto Scienza e Tecnica per la Qualitá della Mobilitá Locale.

represented by the boolean TD variable *in*; the train exit event is denoted as *out*. The requirement that every train exit exactly 10 seconds after its entrance is expressed by the following formula:

in -> Futr (out, 10)

The formula expresses, through a logical implication, the constraint that if a train arrives at the current time, a train will exit exactly 10 time units after.

**Derived Operators**

To specify more complex relations TRIO provides a set of *derived* temporal operators. In a more realistic example, the entrance of a train into a railway track is followed by its exit *within* a given time, not *exactly after* that time.
To express this constraint it is useful to employ the derived operator *WithinF* ("within in the future"), defined as follows:

WithinF(A,t) $=_{def}$ exists d (0<d<t & Futr(A,d))

The meaning of WithinF(*A*, *t*) is that *A* will be true at a distance *t* or less in the future. Then the requirement expressed informally before on the railway track can be formalized as follows:

in -> WithinF (out, 10)

An even more realistic example is that the train can not exit before a given time, such as 5 seconds (i.e., the train must run through a certain track with a given speed limit). A derived operator useful for describing this situation is *Lasts*, defined as follows:

Lasts(A,t) $=_{def}$ forall d (0<d<t -> Futr(A,d)).

The meaning of Lasts(A, t) is that property *A* holds for all next *t* time instants (present time and instant at distance *t* excluded). The previous constraint on the minimum time to exit the railway track is therefore expressed as follows:

in -> Lasts(~out,5) &Within(out,10)

The meaning of the formula is that if *in* occurs at the current time then *out* will be false for at least 5 five time units (that is, the train will not exit in the first five seconds) but it will become true before 10 second elapse.
TRIO includes many other derived temporal operators, which allow a specifier to express even the most complex timing requirements. For a thorough discussion on the derived temporal operators, we remind the interested reader to [2], as no other temporal operator is necessary to describe the time requirements in the topological interface.

**Axioms**

Every axiom is a TRIO formula that expresses an invariant property of the specified system. As opposed to methods based on states and transitions, which describe what the system must do by indicating how it must be done, the TRIO language allows one to describe the system by specifying the properties that it must satisfy. Each property is described by a formula, called axiom. Therefore, in general, a TRIO specification is a collection of axioms.

**Modularity**

TRIO specifications are usually organized in modules. The possibility of structuring a specification into modules provides a support to an incremental top-down approach, to the specification activity through successive refinements but also permits to construct reusable specifications of independent (sub)systems which can be composed in a different manner depending on the application context. With TRIO it is also possible to describe a system at different abstraction levels and to focus with greater attention and detail on some more critical and relevant aspects, without specifying formally, or providing only a partial specification of other parts that are considered as less critical or more standard.
TRIO modules are called *classes*. A class can be simple or structured. A simple class is a set of axioms prefixed with the declaration of the class *items*, i.e., of the variables, predicates, and functions, both time dependent and time independent, that occur in the axioms. As an example of a simple class, let us consider class "topografia", a fragment of the specification of the topology in a subway plant:

**class** topografia
*// Sba denotes the set of automatic block sections in the plant.*
*Outputs:*
**TI Items** *//declaration of time indep. functions and predicates*
**functions**
velMax : Sba -> Integer
*// the maximum speed, in km/h, allowed in a sba*
**predicates**
succ(Sba, Sba);
*// succ(a,b) holds iff a is a topological successor of b*
pred(Sba, Sba);
*// pred(a,b) holds iff a is a topological predecessor of b*
**Axioms**
**Vars** s, s1, s2: Sba
1: succ(s1,s2) <-> pred(s2,s1)
2: forall s1 exist s2 succ(s1,s2) | pred(s1,s2)
*// every sba has at least one successor or one predecessor*
*// (there is no isolated sba)*
3: forall s (velMax(s) IN { 0, 15, 30, 45, 65, 77})
*//possible speed values*
**end** topografia

**Structured classes**

Classes that have components, called *modules*, belonging to other classes are called structured classes. Structured classes support the description of modular TRIO specifications, suitable to describe systems whose *parts* must be clearly identified.
For instance, the specification of the Topological Interface can be structured into three parts: a route manager (module *gestioneItinerari*) a topology manager (module *topografia*) for a particular plant and a module that computes the information to be sent to the track sections (module *calcoloCurva*). Such a specification can be represented by a figure as follows.

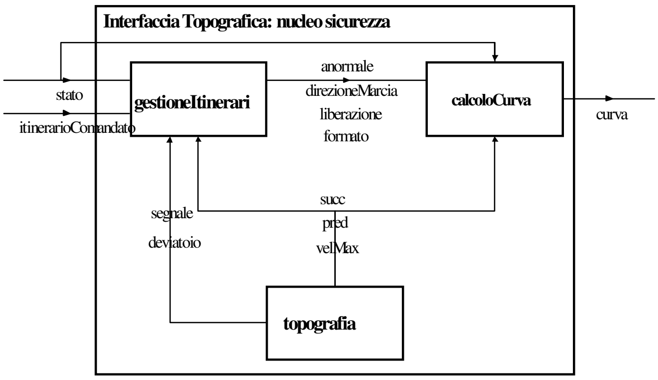



**Figure 2.1**. The structure of the Interfaccia Topografica.

The module Interfaccia Topografica receives information on the status of the plant (item *stato*) and on the driven route (item *itinerarioComandato*) and provides the speed curve (item *curva*) for the track circuits of the route. The three inner modules shown in the figure above, *gestioneItinerari, topografia*, and *calcoloCurva,* correspond to the three parts previously outlined. The connections among the modules represent the exchange of information in the directions shown by the arrows. For instance, the maximal speed in a section (item *velMax*) is provided by module *topografia* both to module *gestoreItinerari* and to module *calcoloCurva*.

## 3. THE SPECIFICATION

### 3.1 The specification entities

We now introduce the various entities that will be formalized in the specification.
A *track circuit* (*circuito di binario*, or *cdb*) is an electrical circuit whose purpose is to transmit suitable messages, called *codes*, to the on board devices. A track circuit is the abstract representation of the physical entity constituted by a piece of track. Each cdb is in relation with a *peripheral post* (*Posto Periferico*) and is a component of an *automatic block section* (*sezione di blocco*

*automatico*, *sba*).

An automatic block section is the piece of track referenced by the messages that the ground devices send to the on board devices. Each sba includes one or more track circuits, with two signals (entrance signals in the two possible running directions) and zero, one, or more switches. An sba is the piece of track of minimal length for which it is certain that, upon occurrence of a train block, the train will reach a stop without overcoming its end. Each sba receives form the topological interface a *code*: that is, the information on the *entrance speed* (VISBA) and on the *exit speed* (VUSBA) of the sba itself, and on the currently set running direction (DIMAR). The *running direction* (*direzione di marcia*) of an sba can be *forward* (*normale*) or *backward* (*inversa*). The running direction of a route is, by definition, unique for the entire route, and is determined by the first sba composing the route. Each sba is associated to two *final points* (*punto finale*) corresponding to its two extremities.

A *switch* (*deviatoio*) can be in three positions *forward blocked* (*bloccato normale*), *backward blocked* (*bloccato rovescio*), and *not blocked* (*non bloccato*).

A *route* (*itinerario*) is composed of an ordered sequence of consecutive sba, by a *driven signal* (*segnale comandato*) and by the state of the final point of the last sba in the sequence. The route is provided by the ACEI (see below for its definition) upon request by the train operators and it always includes, when it is constructed, at least two *automatic block sections*. The first section is the one where the train that is requesting the route is currently positioned; the second one is the immediately successive sba. The final point of a route can be either *free* (*libero*), if the sba does not belong to any route, or blocked (*bloccato*) otherwise. In general the final point of a route should be blocked, to indicate that the last sba of the route is not used for any other route, a situation to be avoided because it can cause train collision.

A *signal* (*segnale*) is a semaphore that can be in one of three positions: *stop* (*via impedita imperativa*) when no train can be authorized to go beyond the signal, *dark* (*spento*), *warning* (*rosso permissivo*) when the trains must stop but may be authorized to proceed in some particular cases, and *drive* (*via libera*). The main link between a route and a signal is the *driven signal* (*segnale comandato*). A driven signal is the signal at the entrance of the second sba of a route: this signal is important because the second sba is the first following the sba where the train is running. The peripheral post handles the signal based on the operational state of the related track circuit.

A *stop* (*fermata*) is a subway station.

The ACEI (*Apparato Centrale Elettrico a pulsanti di Itinerario*) is the subsystem that receives from the on board devices the request of a route, and that is in charge of reserving (by *blocking* them) the necessary sections, switches, and signals.

The *peripheral post* (*Posto Periferico,* or PP) is the set of electronic appliances that define, through the track circuits, the free or blocked state for the piece of track under control. It can exchange data with the ACEI and the topological interface.

The *Topological Interface* (*Interfaccia Topografica*, or IT) is the central subsystem that manages and selects the codes. It receives information from the peripheral posts on the state of the track and on the requested routes; it generates all the codes to be sent to each track circuit. Such codes are necessary to send a train on a given route.

The main purpose of the specification is to describe the requirements of the safety kernel of the topological interface. Under the simplifying hypotheses reported in the introduction, the individual track circuits of a given sba are not distinguished in the present specification, and are therefore ignored.

### 3.2 The TRIO Specification

The specification is parametric with respect to the system, which is actually defined by the sets: *Deviatoio, Segnale, Sba, PuntoFinale, Itinerario, and Codice.*

**Some useful functions defined on *Itinerario* (route):**

Function *length* of a route
lung: Itinerario -> Integer
*returns the number of Sba in the route;*
Function *rest* of a route
  rest: Itinerario -> Itinerario
  *rest(i) returns the route i without its first Sba*
Function *element* of a route
  sba: (Integer , Itinerario) -> Sba
  *sba(k,i) returns the k-th Sba of the route i*
Function *first* of a route
  prima: Itinerario -> Sba
  prima(i) ≡ sba(1,i)
Function *last* of a route
  ultima: Itinerario -> Sba
  ultima(i) ≡ sba(lung(i),i)
Function *last-but-one* of a route
  penultima: Itinerario -> Sba
  penultima(i) ≡ sba(lung(i)-1,i)

**The Structure of the Topological Interface**

We now describe the structure of the class *interfacciaTopografica* (the Topological Interface). The input items of the class are the function *stato (*the current state of switches, block sections, sba signals and cdb) and the *itinerarioComandato* signal (i.e. the driven signal), which corresponds to a route request/block.

The output is the speed curve associated to every sba. The overall structure of the system is represented by the modules' diagram, which consists of three modules: *gestioneItinerari*, *topografia* and calcoloCurva.

```
class interfacciaTopografica
Inputs:
TD Items
   functions
   stato is overloaded over the sets of devices of the rail network:
      stato: deviatoio -> {bloccatoNormale, bloccoRovescio,
                              nonBloccato}
      stato: segnale ->  {viaImpedita, spento, rossoPermissivo,
viaLibera}
      stato: Sba -> {libero, bloccato, occupato}
      stato: Cdb -> {libero, occupatoDaTreno, occupatoDaDisturbo}
      stato: PuntoFinale -> {libero, bloccato}
   predicates
      itinerarioComandato(Itinerario)
   The set of the new routes ( requested routes).
   Every  requested route includes at least two Sba's.
Outputs:
TD Items
   functions
      curva: Sba -> Codice
Modules
//The module decomposition is reported in Figure 2.1
end interfacciaTopografica
```

**The topological data management module**: *topografia*

The *topografia* module provides the main topological data to the other modules, *gestioneItinerari* and *calcoloCurva*. It contains all the static and embedded information about the topological structure of the system: its physical displacement and the network characteristics.

The network structure is represented by *succ*, *pred* e *velMax*. The *succ* predicate identifies the topological successor(s) of a given section, with respect to the forward running direction. The *pred* predicate identifies the topological predecessor(s) of a given section (likewise, *succ* identifies successors). The maximum speed allowed for a section is represented by the *velMax* function. Moreover, *gestioneItinerari* uses the predicate *segnale*, representing the semaphore signal at the entrance of a section (as usual with respect to the forward running direction), and the predicate *deviatoio*, representing a section's switches. The complete specification is reported in Section 2, while the definition of the topology of the Naples plant is reported in Section 4.

**The module *gestioneItinerari***

The module for route managing (gestioneItinerari) provides the module calcoloCurva with important information about the various routes. Such items of information are called *formato, liberazione*, and *direzioneMarcia.*

The time-dependent item *formato* is the set of formed routes at

every instant.
The time-dependent item *liberazione* is the set of routes whose number of sba has been reduced, because the train in the route has completed its first sba: such sba is then freed and made available to form other routes; in the particular case that a route *i* was composed of just one sba, *liberazione(i)* means that route *i* has been completed, that is the train arrived at the end of the route.
The time-dependent item *direzioneMarcia* supplies the direction of every formed route, which can be either *normale* (forward) or *inversa* (backward).
An internal time-dependent set of *gestioneItinerari*, called *anormale*, is used to denote the routes whose behavior becomes abnormal, for instance because a switch is not correctly blocked.

**class** gestioneItinerari
*Input:*
**TD Items**
**functions**
*The function* stato *is overloaded*
*(it is described also in module Interfacccia Topografica):*
stato: deviatoio -> {bloccatoNormale, bloccoRovescio, nonBloccato}
stato: segnale -> {viaImpedita, spento, rossoPermissivo, viaLibera}
stato: Sba -> {libero, bloccato, occupato}
stato: Cdb -> {libero, occupatoDaTreno, occupatoDaDisturbo}
stato: PuntoFinale -> {libero, bloccato}
**predicates**
itinerarioComandato(itinerario)
segnale: (Sba, {normale, inversa}) -> Segnali
segnale(s,d) *is the signal of the sba d for the running direction d (*normale *is forward,* inversa *is backward)*
*Output:*
**TD items**
**predicates**
liberazione(Itinerario) *set of freed routes at every instant*
formato (Itinerario) *set of formed routes at every instant*
direzioneMarcia(Itinerario, {normale, inversa}) *running direction of routes*
*Internal:*
**TD items**
**predicates**
anormale(Itinerario) *set of abnormal routes at every instant*
**Axioms**
vars i, i1: Itinerario, s: Sba, d: deviatoi, dir: {normale, inversa}

liberazioneItinerario:
*a route is freed (*liberazione*) when the train has completed its first sba*
liberazione(i) <-> past(stato(prima(i)) = occupato, 1) &
stato(prima(i)) = libero

formazioneItinerario:
*a route is formed* (formato) *when it is not abnormal and one of the following conditions holds:*

- *it is a requested route (*itinerario comandato*);*
- *or at the previous instant it was formed and not yet freed (liberazione has a delayed effect);*
- *or it derives from a freed route which is not yet empty.*

formato(i) <-> ~anormale(i) &
(itinerarioComandato(i) |
past(formato(i), 1) & ~ liberazione(i) |
exists i1 (past(formato(i1) &
liberazione(i1), 1) & i = rest(i1) &
lung(i)>0))

direzioneDiMarciaComandati:
*The running direction of a requested route can be derived from the topology of the system, since its length is at least 2:*
itinerarioComandato(i) ->
(direzioneMarcia(i, normale) <->
pred(penultima(i), ultima(i))) &
(direzioneMarcia(i, inversa) <->
succ(penultima(i), ultima(i))))

direzioneDiMarciaEsistenti:
*the direction of a formed route does not change in time; moreover, if a formed route is freed, the new corresponding formed route keeps the same direction of the old one:*
formato(i) & ~ itinerarioComandato(i) ->
(forall i1 (past(formato(i1),1) & (i = i1 |
past(liberazione(i1),1) & i = rest(i1) ->
direzioneMarcia(i,dir) <-> direzioneMarcia(i1,dir))

segnaleComandato:
*the definition of the driven signal of a route: the signal, in the appropriate running direction, of the first sba of the route:*
formato(i) -> exists dir (direzioneMarcia(i,dir) &
segnaleComandato(i) = segnale(sba(2,i), dir))

anormalita':
*a route is abnormal (*anormale*) when:*
*its final point is not blocked (*bloccato*)*
*or one of its switches (*deviatoio*) is not blocked*
*or its driven signal (*segnaleComandato*) is in the barred position (*viaImpeditaImperativa*) or it is off (*spento*).*
anormale(i) <->
(stato(puntoFinale(i)) <> bloccato |
exists s exists d (s IN i & deviatoio(d,s) &
stato(d) = nonBloccato) |
stato(segnaleComandato(i)) = viaImpeditaImperativa |
stato(segnaleComandato(i)) = spento)
**end** gestioneItinerari

**The speed curve module: *calcoloCurva***
The speed curve module (*calcoloCurva*) calculates and provides the speed signal for every sba involved in a route. The speed signal is coded using the record items VISBA, VUSBA e DIMAR, which are Italian acronyms for entrance speed, exit speed and running direction, respectively.
The DIMAR item is fixed in every formed route, and corresponds to its first sba's default running direction: this information is provided by *gestioneItinerari,* using the *direzioneMarcia* predicate.
Now consider the case DIMAR = *normale* (forward running direction). The backward-direction case is immediately obtained by swapping VISBA and VUSBA. The speed curve is computed as follows.
Let *i* be the current formed route, and let *s* be its current sba.

- VISBA is the max speed allowed in s, verifying the equation:
  *curva(s).VISBA = velMax(s).*
- VUSBA depends on the next, with respect to the route, sba's maximum speed limit. If *t* is the next section, the constraint is:
  *curva(s).VUSBA = velMax(t).*

We now consider the case of the last sba of a formed route. Since it is assumed that the train must stop, after finishing its route, the speed in the last sba is set to 0.
Another constraint is given by the presence of another train in the next sba: it is necessary to keep at least one free sba between two trains, as a security measure. In this case the speed curve must be 0, from the last-but-one sba, to the end of the current route.
*Topografia* sends to *calcoloCurva* all the topology information about the sba's (this is done using the *succ* and *pred* predicates) and their speed limits (function *velMax*).
The module for route managing supplies the following information about routes: *liberazione*, *direzioneMarcia* and *formato*.
The output of calcoloCurva is the speed curve, computed for every section.
The state of the network (*stato*) - provided by the Peripheral Posts - is used to determine if there is a train in the following sections.

**class** calcoloCurva

*Inputs:*

**TI Items**

**functions**

```
velMax : Sba -> Integer;
```
*sba's speed limit*

**predicates**

```
succ(Sba,Sba);
pred(Sba,Sba);
```

**TD Items**

**predicates**

```
formato(Itinerario);
liberazione(Itinerario);
direzioneMarcia(Itinerario, {normale, inversa});
```

*Outputs:*

**TD Items**

**functions**

```
curva : Sba -> Codice;    the speed curve
Codice : record
   VISBA : Integer;       sba's entrance speed
   VUSBA : Integer;       sba's exit speed
   DIMAR : {normale, inversa}; sba's current running direction
   end;
```

*Internals:*

**TD Items**

**predicates**

*trenoProssimo(i) holds iff there is a train in at least one of the next adjoining sections*

```
trenoProssimo(Itinerario);
```

**Axioms**

**vars**

```
i : Itinerario;
s : Sba;
k : Integer;
d : {normale, inversa};
```

**Speed curve for the sections not in formed routes**

*Every sba not belonging to formed routes is in a default rest state: VISBA = VUSBA = 0.*

```
axNonFormati:
forall s
 (~exists i (formato(i) & s IN i) ->
  (curva(s).VISBA = 0 &
   curva(s).VUSBA = 0)
  );
```

**Running direction for formed routes**

*It is supplied by the gestioneItinerari module, to send the current DIMAR to very sba.*

```
axDirezMarcia:
 forall i forall d   (formato(i) & direzioneMarcia(i,d)->
            forall k    (1 <= k <= lung(i) ->
                         curva(sba(k,i)).DIMAR = d)));
```

**Axioms for trenoProssimo**

*These axioms manage the last sections of a formed route: if there is a train in at least one of the next adjoining sections, then the route speed curve is set to 0 in the current section. This assures the fact that betweed two running trains there is at least one completely free section. When a route has a length less than or equal to 2, calcoloCurva must consider even the axioms concerning "liberazione". trenoProssimo(i) holds iff there is a train in at least one of the i route's following adjoining sections.*

```
axTrenoProssimo:
 forall i (formato(i) ->
  (trenoProssimo(i) <->
   (direzioneMarcia(i,normale) &
    exists s (succ(s,ultima(i)) & stato(s) = occupato) |
    direzioneMarcia(i,inversa) &
    exists s (pred(s,ultima(i)) & stato(s) = occupato))));
```

*There is no close train in the forward running direction: the speed curve is set to 0 only for the last section's VUSBA.*

```
axNonTrenoProssimoDirNorm:
 forall i (formato(i) ->
  (~trenoProssimo(i) &
   (lung(i) > 2 | lung(i) = 2 & ~liberazione(i)) &
   direzioneMarcia(i,normale) ->
    (curva(penultima(i)).VISBA = velMax(penultima(i)) &
     curva(penultima(i)).VUSBA = velMax(ultima(i)) &
     curva(ultima(i)).VISBA = velMax(ultima(i)) &
     curva(ultima(i)).VUSBA = 0)));
```

*There is not a close train in the backward running direction: the speed curve is set to 0 only for the last section's VISBA (it exchanges its role with VUSBA because of the direction).*

```
axNonTrenoProssimoDirInv:
 forall i (formato(i) ->
  (~trenoProssimo(i) &
   (lung(i) > 2 | lung(i) = 2 & ~liberazione(i)) &
   direzioneMarcia(i,inversa) ->
    (curva(penultima(i)).VUSBA =  velMax(penultima(i)) &
     curva(penultima(i)).VISBA = velMax(ultima(i)) &
     curva(ultima(i)).VUSBA = velMax(ultima(i)) &
     curva(ultima(i)).VISBA = 0)));
```

*There is a close train in the forward running direction: the speed curve is set to 0 from the last-but-one section's VUSBA to the end of the route.*

```
axTrenoProssimoDirNorm:
 forall i (formato(i) ->
  (trenoProssimo(i) &
   (lung(i) > 2 | lung(i) = 2 & ~liberazione(i)) &
   direzioneMarcia(i,normale) ->
    (curva(penultima(i)).VISBA = velMax(penultima(i)) &
     curva(penultima(i)).VUSBA = 0 &
     curva(ultima(i)).VISBA = 0 &
     curva(ultima(i)).VUSBA = 0)));
```

*There is a close train in the backward running direction: the speed curve is set to 0 from the last-but-one section's VISBA to the end of the route. (As before, it exchanges its role with VUSBA because of the direction)*

```
axTrenoProssimoDirInv:
 forall i (formato(i) ->
  (trenoProssimo(i) &
   (lung(i) > 2 | lung(i) = 2 & ~liberazione(i)) &
   direzioneMarcia(i,inversa) ->
    (curva(penultima(i)).VUSBA = velMax(penultima(i)) &
     curva(penultima(i)).VISBA = 0 &
     curva(ultima(i)).VUSBA = 0 &
     curva(ultima(i)).VISBA = 0) ));
```

*Routes with unitary length: with a close train, the speed curve is set identically to 0; without, only the exit speed is set to 0.*

```
axLung1:
 forall i (formato(i) ->
  (~liberazione(i) & lung(i) = 1 ->
   (trenoProssimo(i) ->
    (curva(prima(i)).VISBA = 0 &
     curva(prima(i)).VUSBA = 0)
   ) &
   (~trenoProssimo(i) &
    direzioneMarcia(i,normale) ->
    (curva(prima(i)).VISBA = velMax(prima(i)) &
     curva(prima(i)).VUSBA = 0)
   ) &
   (~trenoProssimo(i) &
    direzioneMarcia(i,inversa) ->
    (curva(prima(i)).VUSBA = velMax(prima(i)) &
     curva(prima(i)).VISBA = 0))));
```

### Axioms for new and unchanged routes

*VISBA and VUSBA are computed in this way:*
*with a forward running direction, VISBA is set to the speed limit (velMax)*
*of the same section, while VUSBA is set to the limit of the next sba.*
*As usual VISBA exchanges its role with VUSBA with a backward running direction.*
*Note: the last two sections are managed by the trenoProssimo axioms.*
*- forward running direction*

```
axStazionario-NuovoDirNorm:
 forall i (formato(i) ->
  (~liberazione(i) &
   direzioneMarcia(i,normale) ->
   forall k
    (1 <= k < lung(i)-1 ->
     (curva(sba(k,i)).VISBA = velMax(sba(k,i)) &
      curva(sba(k,i)).VUSBA = velMax(sba(k+1,i))))));
```

*- backward running direction*

```
axStazionario-NuovoDirInv:
 forall i (formato(i) ->
  (~liberazione(i) &
   direzioneMarcia(i,inversa) ->
   forall k
    (1 <= k < lung(i)-1 ->
     (curva(sba(k,i)).VUSBA = velMax(sba(k,i)) &
      curva(sba(k,i)).VISBA = velMax(sba(k+1,i))))));
```

*General case (route length greater than 2):*
*the speed curve for the first sba is set to 0, while the following - with the excepiton of the last two sections - maintain their configuration.*

```
axLiberatoGenerale:
 forall i (formato(i) ->
  (liberazione(i) & (lung(i) <> 2) ->
   (curva(prima(i)).VISBA = 0 &
    curva(prima(i)).VUSBA = 0 &
     forall k
      (1 < k < lung(i)-1 ->
       past(curva(sba(k,i)).VISBA,1) = curva(sba(k,i)).VISBA &
       past(curva(sba(k,i)).VUSBA,1) = curva(sba(k,i)).VUSBA
     ))));
```

### Axioms for reduced routes

*These axioms manage the case of reduced routes:*
*the once-first sba is disposed and marked as free for other routes.*
*Its VISBA and VUSBA curve items are set to 0.*
*The 2-length case is partially managed by the trenoProssimo axioms.*
*Route length equal to 2: the speed curve is set to 0 in the first sba.*
*The speed curve of the last sba depends on trenoProssimo.*

```
axLiberatoLung2:
 forall i (formato(i) ->
  (liberazione(i) & lung(i) = 2 ->
   (curva(prima(i)).VISBA = 0 &
    curva(prima(i)).VUSBA = 0) &
   (trenoProssimo(i) ->
    (curva(ultima(i)).VISBA = 0 &
     curva(ultima(i)).VUSBA = 0)
   ) &
   (~trenoProssimo(i) &
    direzioneMarcia(i,normale) ->
    (curva(ultima(i)).VISBA = velMax(ultima(i)) &
     curva(ultima(i)).VUSBA = 0)
   ) &
   (~trenoProssimo(i) &
    direzioneMarcia(i,inversa) ->
    (curva(ultima(i)).VUSBA = velMax(ultima(i)) &
     curva(ultima(i)).VISBA = 0)
   )
  ));
end calcoloCurva
```

## 4. TEST CASE GENERATION FOR THE SPECIFICATION OF THE SAFETY KERNEL

The test case generation activity was concentrated on a small set of *scenarios*, which are situations deemed worth a verification activity. Although considering other scenarios could be useful, the selected ones were quite significant and the most important to verify for the system.

The test case generation activity in TRIO is based on the generation, supported by suitable tools, of histories for the specified system. Such histories (henceforth called *test cases*) describe possible execution sequences of the system, including both input data to be provided to the system and the expected corresponding output data. The supporting tools allow generation of test cases in an optimized way avoiding as much as possible the production of redundant information, characterizing input vs. output events, and effective handling of nondeterministic behaviors.

In a first phase of the testing activity, test cases can be used to check the specification itself, verifying whether what we specified is consistent with the knowledge of experts about the system. For instance, it is possible to propose a history of the system breaking some safety requirements and automatically verifying whether such behavior is allowed by the specification. This kind of analysis increases confidence in the correctness of the specification, much in the same way as testing a program may increase the confidence in its reliability: in fact, although testing cannot prove the absence of errors, it is especially valuable as a means to validate functional requirements.

In a second phase, when the specification is considered correct, the test cases can be collected and used to test the implementation of the system. Testing is simplified because the correct output data are also available.

In this project, the testing activity has allowed us to find a small, but subtle, error in the specification, due to an erroneous interpretation of the informal requirements of the system. The error was indeed fixed very easily. However, this experience is far from uncommon: without such a validation activity, there are errors in the specification that can easily go undetected and be included in the implementation of the system.

### 4.1 The scenarios

The study has considered six sba's (called sba_1, .., sba_6) and ten routes (it_1, .., it_10). Every route corresponds to the travel from one sba to another one, as described in Table 4.1. Many other routes are possible, but the study was restricted to the routes of the proposed scenarios.

| route | Starting sba | arrival sba |
|---|---|---|
| **it_1** | Sba_1 | sba_6 |
| **it_2** | Sba_2 | sba_6 |
| **it_3** | Sba_3 | sba_6 |
| **it_4** | Sba_4 | sba_6 |
| **it_5** | Sba_5 | sba_6 |
| **it_6** | Sba_6 | sba_6 |
| **it_7** | Sba_1 | sba_4 |
| **it_8** | sba_2 | sba_4 |
| **it_9** | sba_3 | sba_4 |
| **it_10** | sba_4 | sba_4 |

**Table 4.1**: The routes considered during the testing activity

The test case generation activity has covered the following situations:

1. a train goes from beginning to the end in forward direction (hence, from sba_1 to sba_6);
2. two trains, one following the other, proceed in the forward direction: the first train proceeds from sba_2 to sba_6, while the second enters only later, going from sba_1 to sba_4;
3. two trains proceed as in the previous case, but are too close

one to each other, violating safety requirements;

4. two trains, one following the other, proceed in the backward direction (this scenario is symmetrical to case 1).

The results show that the specification correctly allows the generation of test cases corresponding to scenarios 1, 2 and 4, while scenario 3 is, again correctly, rejected by the tools because it violates the formal specification.

For reasons of efficiency of execution, the topology of the Naples subway (which is not reported here) has been included in the form of constraints for every history rather than as axioms. Such constraints describe the values, for such particular subway, of the following time independent items:

the relation *pred* between pairs of contiguous sba's;

the function *rest*, which maps every route to the corresponding route deprived of the first sba;

the function *sba*, which gives the i-th sba of a given route;

the function *velMax*, denoting the maximum speed for each sba;

the function *lung*, which gives the number (length) of sba of every route;

the *puntoFinale* function, which gives back the indication of the final point of every route.

### 4.2 Experimental results

### Scenario 1: one train

This is the most basic scenario: there is only a train running in the subway. This train must cover, with a forward running direction, every section of the subway. There are no delays, faults or other abnormal behaviors.

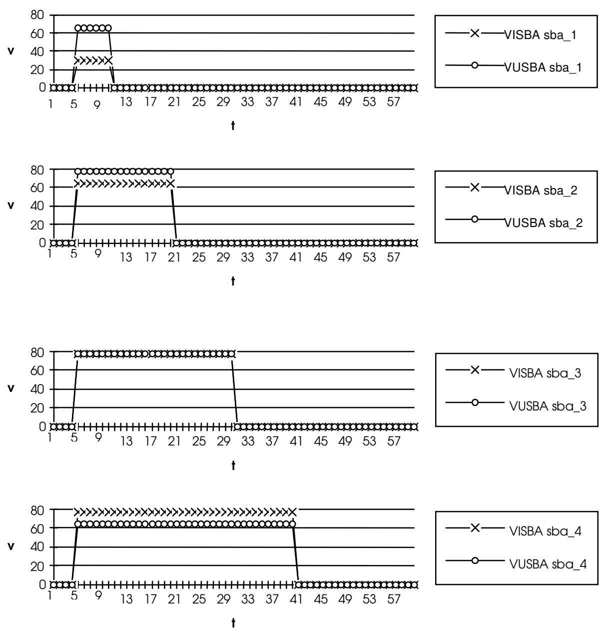
The requested route is it_1 at instant 5. This causes the train, initially in the sba_1 section, to begin its march, headed towards sba_6.

As we can see in the following figure, sba_1's VISBA and VUSBA are set to their maximum values, during the time interval 5.10 - i.e. when the train is on sba_1.

At instant 11 the train passes sba_1: it_1 route is freed and the current route becomes it_2 (i.e. from sba_2 to sba_6).

Similarly, every ten instants the train passes an sba, which is released: the train reaches sba_3 at 21; sba_4 at 31 etc.

At instant 51 the trains stops: the destination (sba_6) is reached.



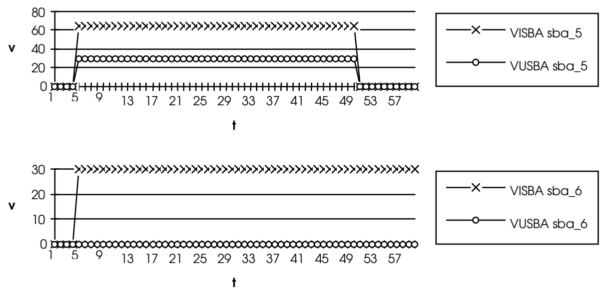




**Figure 4.1.** The time progress of the speed curve - the VISBA and VUSBA items

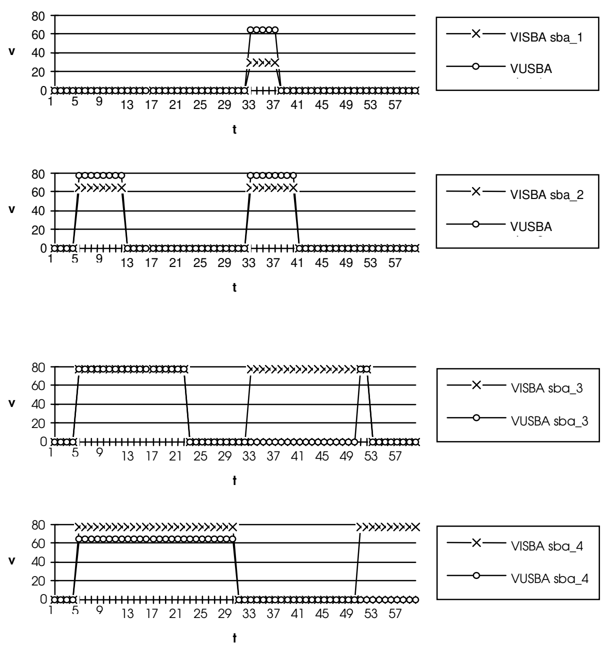
### Scenario 2: two trains

The second scenario is more complex: there are two trains (train 1 and 2) placed in sba_2 and sba_1, respectively. The first train must cover sba_2 to sba_6 with a forward running direction. The second train starts from sba_1 and must reach sba_4, clearly with the same running direction. The two routes are it_2 and it_7, respectively.

The requested route is it_7 at instant 5: train 1 starts up. As depicted in the next figure, train 1 remains on sba_2 until instant 13. Then it covers sba_3, then sba_4 at instant 23, and sba_5 at 31. From 31 to 50 it is blocked in sba_5.

At instant 33 the route it_7 becomes active - the train 2 is set in motion to reach sba_4. It takes 5 instants to get through sba_1. It then reaches sba_2 at instant 38, then sba_3 at 41. Here the second train stops, because train 1 blocks sba_5. Actually, the specification states that there must be at least one completely free sba between two trains-namely, sba_4.

Sba_4 is available only since instant 51, because train 1 passes to sba_6. So train 2 can go through sba_3 to reach its destination: sba_4. This is the end of the scenario: the two trains attended their duties.



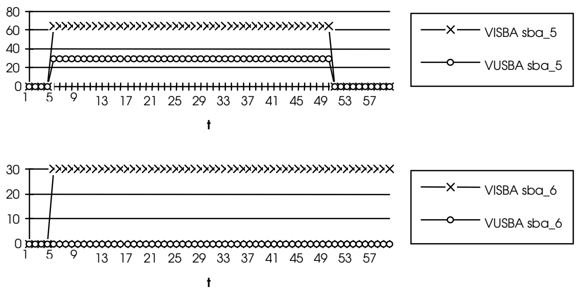



**Figure 4.2.** The time progress of the speed curve - the VISBA and VUSBA items for scenario 2.

**Scenario 3: two trains in a potentially dangerous situation**

As an interesting critical situation, we forced a speed curve in which the two trains - described in scenario 2 - can collide. Particularly, we consider the case of a non-zero speed curve in sba_3 and sba_4 when the two trains are in sba_5 and sba_3, respectively (see figure 4.3).

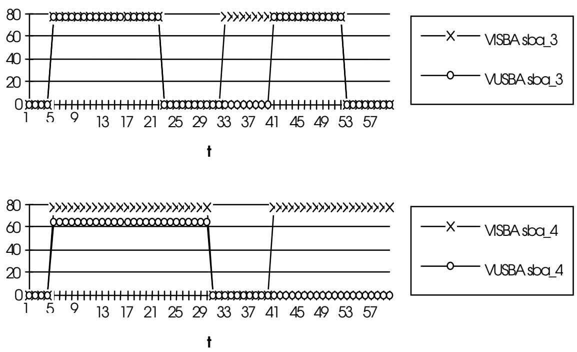



**Figure 4.3.** The modified dangerous speed curve of scenario 3.

This is a very dangerous situation, because train 2 could reach train 1 and then crash. However, as we can see in figure 4.4, the TRIO semantics tools immediately reject this pattern: it is not compatible with the specification of the speed curve-computing module.

In fact the execution trace show that the TRIO tools reject the history at instant 41: the axiom *axNonTrenoProssimoDirNormale* does not hold.

```
Formula axNonFormati is True at 40
Formula axDirezMarcia is True at 40
Formula axTrenoProssimo is True at 40
Formula axNonTrenoProssimoDirNorm is True at 40
Formula axNonTrenoProssimoDirInv is True at 40
Formula axTrenoProssimoDirNorm is True at 40
Formula axTrenoProssimoDirInv is True at 40
Formula axLung1 is True at 40
Formula axStazionarioNuovoDirNorm is True at 40
Formula axStazionarioNuovoDirInv is True at 40
Formula axLiberatoGenerale is True at 40
Formula axLiberatoLung2 is True at 40
Formula axNonFormati is True at 41
Formula axDirezMarcia is True at 41
Formula axTrenoProssimo is True at 41
Formula axNonTrenoProssimoDirNorm is False at 41
```

**Figure 4.4:** Execution trace of the TRIO testing tools for the dangerous scenario 3.

**Scenario 4: two trains moving backward**

This last scenario is practically identical to the second one: the only difference is the running direction, which is backward for the two trains.

Actually, the topological configuration is the following:

```
/*** inverted Topological Configuration ***/
 pred(sba_2,sba_1) : [1..60]
 pred(sba_3,sba_2) : [1..60]
 pred(sba_4,sba_3) : [1..60]
 pred(sba_5,sba_4) : [1..60]
 pred(sba_6,sba_5) : [1..60]
```

We used the scenario to test the axioms for the backward running direction. The results turned out to be, as expected, wholly symmetrical to those of the forward running direction scenario and are omitted here.

## 5. CONCLUSIONS

The main goal of this study was to investigate and assess the potential applicability of the TRIO technique in the field of railway signaling systems. The experiment was certainly successful. We derived from the informal documentation of the Safety Kernel of the Naples Subway a formal specification of the requirements, written in the TRIO language. Then a validation phase, based on testing techniques was performed on the specification, leading to the discovery of a subtle error in the specification. The validation activity also produced a small set of scenarios that could be used as functional test cases. As a modeling language TRIO certainly proved to be adequate to describe the data and timing requirements of the Safety Kernel. We believe that the adoption of the same formal method would be very beneficial in the development of other very critical components of the signaling system of the subway such as the ACEI and the Peripheral Post. In general, we estimate that the adoption of the TRIO language and tool environment would provide better specifications and improve the quality of the test plans, in terms of: a explicit correspondence between test cases and properties they are meant to verify, a precise evaluation of the obtained coverage, and a greater confidence of test case correctness (test cases would be obtained systematically by means of semiautomatic tools from the specification).

## 6. REFERENCES

[1] "METRO NAPOLI – Interfaccia Topografica – Descrizione Funzionale", Ansaldo Trasporti, Gennaio 1994.

[2] E.Ciapessoni, A.Coen-Porisini, E.Crivelli, D.Mandrioli, P.Mirandola, A.Morzenti, "From formal models to formally-based methods: an industrial experience", ACM TOSEM - Transactions On Software Engineering and Methodologies, vol. 8, No 1, January 1999, pages 80-115.

[3] A. Morzenti, P. San Pietro, "Object-Oriented Logic Specifications of Time Critical Systems", ACM TOSEM - Transactions on Software Engineering and Methodologies, vol.3, n.1, January 1994, pp. 56-98.

[4] "Impianti di segnalamento e automazione. Relazione di integrazione al progeto esecutivo", Metropolitana Milanese, Progettazione e direzione lavori, Novembre 1993.