Swiss Federal Institute of Technology Lausanne
Department of Computer Science

Diploma Work Report

A Formal Specification for a Real-Time Train Controller

Simon Kramer*

February 19, 2001

*is at the Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland and has carried out the present work during a stage at the Politecnico di Milano, Italy. The work has been supervised by Prof. Dino Mandrioli on the side of the Politecnico di Milano and by Prof. Martin Odersky on the side of the EPFL.

Abstract

We give a formal specification for a real-time controller for trains that operate on the Italian railway network. The Controller will control train movement and is part of a larger system destined to guarantee safety with respect to dangers originating from train traffic in the railway network.

Based on an informal specification document from the Italian railway company, we construct a simple state-based model and formalise it in terms of the property-based specification language TRIO. The obtained specification being formal, we are able to perform certain verifications on it, such as checking its satisfiability and verifying correctness of refinement steps.

Contents

List of Tables

List of Figures

List of Specification Schemata

List of Proofs

About this Document

The present document is structured at three different levels. At the top level of the sectioning hierarchy, we make a *logical* distinction as well as a distinction related to the different *project phases* our work has gone through.

The project phases map in order to the three parts 'Prologue', 'The Specification' and 'Epilogue'. In the first part, we introduce the reader to the problem domain, explain how we intend to solve the problem, and, within the problem domain, carve out the sub-domain that we intend to address in this document. We give a solution to our problem in the second part, and finally consider the solution in its context in the third part.

The logical distinction separates the parts that contain information *about* the specification ('Prologue' and 'Epilogue') from the part that contains the proper specification *itself.*

At the next deeper level of our section hierarchy, we develop the *process-related* issues of our work, and by doing so, gradually produce the desired *results.* So, results are situated at the bottom of the sectioning hierarchy.

The reason for the chosen document structure is that by subordinating process-related issues to the distinction of meta and object level in the specification problem, we have separated *problem-* or *case-specific* information (the specification) from *general* and *project-related* information. This is useful if, for example, we want to present our specification as a case study among others that use the same notations and conventions.

Part I

Prologue

Chapter 1

Introduction

1.1 A Real-World Case

We are addressing the problem of specifying formally a real-time controller for trains that operate on the Italian railway network. The controller will control train movement and is part of a larger system destined to guarantee safety with respect to dangers originating from train traffic in the railway network[1].

Our work is part of a joint project undertaken by the Italian national railway company (Ferrovie dello Stato, FS), the Politecnico di Milano, and the three industrial partners ADtranz, Alsthom and Ansaldo, which are charged with the implementation of the complete system.

In order to get an overall understanding of the system, we first consider it from a *model-based* or *extensional* point of view by defining an abstract model (see Figure 1.1) for it. For this purpose, we define the term *Environment* to refer to the train and its physical context such as tracks, light signals and other trains. The Environment can generate an *external event*, which is intended to be captured by the train controller (*Controller*). The Controller may react to the external event by executing a specific *action* in the domain of the Environment.

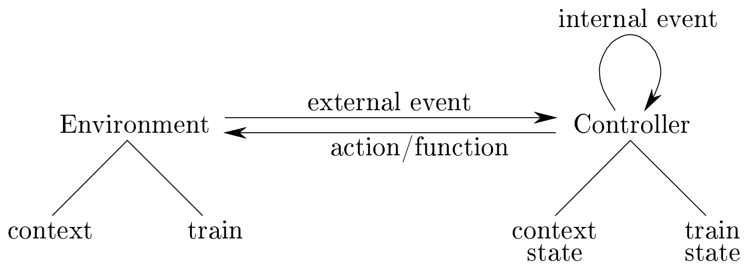


Figure 1.1: Abstract System Model

An external event conveys *data* to the Controller, which uses the data for updating two internal states (*context state* and *train state*). Updating is performed

[1]In Chapter 7, we will see that the Controller is not only physically a part of the larger system, but also logically — with respect to system properties such as safety

by generating *internal events*. The same kind of events is generated for carrying out calculations that may precede the issuing of actions by the Controller.

We refine our model by giving an abstract implementation of the external behaviour of the Controller in terms of its internal states. We also use internal states for defining a *reaction policy* for the Controller. The Controller is supposed to act accordingly when considering reaction to external events:

Definition 1 (Event Reaction Policy) The Controller reacts to an external event if and only if the train state is *inconsistent* with the context state.

The notion of inconsistency can be expressed by defining a *predicate*. In fact, the definition of this predicate corresponds exactly to the specification of the *functional requirements* for our train controller (see Requirement 2 as an example of a *non-functional requirement.*). Further, the inconsistency predicate enables us to define the concept of a *scenario*, which is an instance of a so called *controller history*:

Definition 2 (Controller History) A controller history is defined to be a sequence of pairs $(cs_0, ts_0), \ldots, (cs_n, ts_n)$ such that:

- cs_0 denotes the first context state and ts_0 denotes the first train state before the occurrence of some internal state update, and
- cs_n denotes the first context state and ts_n denotes the first train state after some later occurrence of an internal state update

Thus, a controller history is compatible with a totally-ordered linear *time structure*[2] that may have a beginning and an end. Moreover, the time structure is discrete and can be considered to be induced by the update activity of the Controller.

Definition 3 (Scenario) A scenario is defined to be a controller history for which the inconsistency predicate is false for the first and last pair, and true for all other pairs in the history.

Let us now consider the system from a *property-based* or *intensional* point of view by focusing on its purpose, i.e., guaranteeing safety in the railway network. As far as the Environment is concerned, we are obliged to consider it as a black-box and express its properties as *assumptions*. This is because we only control the train but not the context. As a consequence, we must presume some intuitive understanding of terms that actually belong to the domain of the Environment but which we are going to use in our considerations. Such terms are enclosed in quotation marks (‘’). Finally, once the assumptions have been expressed, we will be able to derive from them some first (informal) requirements for the Controller.

[2] not to be confused with the time *domain* of the formalism we are going to use (see Section 2.2)

Assumption 1 (Event Emission) The Environment generates sufficiently many and meaningful events for the Controller so that the latter is able to reproduce an 'adequate' representation of the Environment in its internal states. Moreover, the Environment generates all external events in such a way that the Controller is able to capture them at the 'right' time and at the 'right' place.

Assumption 2 (Event Transmission) No external events, nor actions, are lost or altered in the transmission medium.

Requirement 1 (Receptivity) The Controller must be receptive for external events at any time.

Requirement 2 (Reactivity) The Controller must react, i.e., take actions within some 'strict' time bounds. This means that for each Controller activity (updating of internal states, calculations) we must define such a time bound, which may also depend on the actual internal states of the Controller.

Requirement 3 (Functionality) The Controller must comply exactly with the above stated event reaction policy, i.e., the Controller must correctly implement the inconsistency predicate (to be defined).

Let us now adopt an *information-based* point of view of our system and look what different types of information values circulate in it, and see if we can classify them.

We can derive the principal types of information values from the fact that the Controller controls train movement. Train movement is defined by the direction of movement (forward or backward with respect to the orientation of the train), train speed, acceleration and deceleration. We call them the *Four Controlled Physical Quantities, 4CPQ*.

From the fact that events convey data, i.e., information values, and the fact that the Controller must be operational for different *types of trains*, we derive a classification of information values: there is (external) data originating from two spatially different sources (train and context), and at two different times (configuration and operation). Thus, events induce a two dimensional data space (see Figure 1.2).

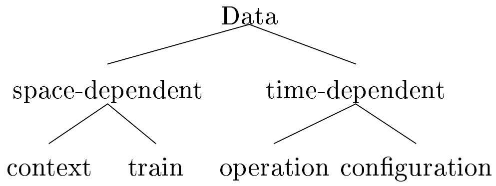


Figure 1.2: Data Space

1.2 Problem, Goal, Task

Our problem is specifying a real-time train controller. The goal obviously is to finally have this specification, and the verifications we can perform on it, in hand. Our task is to find out how to accomplish the goal, and to document the answers found. For this reason, this document contains *process-oriented* as well as *result-oriented* parts.

Having accomplished our task, we will be able to look a bit beyond the scope of our goal, and say a word about the relation between the Controller and the context as far as safety in the railway network, the ultimate concern of the joint project, is concerned.

Chapter 2

Methodology

2.1 Approach

Our work is situated at the very beginning of a large industrial project and thus involves naturally some preliminary exploring of the problem domain (see Section 1.1). The exploration scrutinises a large amount of informal information we have received from the Italian railway company (see Appendix A for an overview of that information, which is contained in [6], [7] and [8]).

The approach consists in first reading and understanding as well as possible this information. From this information, we then extract what we consider relevant for accomplishing our task. We will judge it necessary to develop a new framework for the extracted information, which will make necessary the restructuring of the original information and the introduction of some new concepts. We call this new framework *system model*. Since we want to obtain a formal specification, we proceed by formalising the system model and performing relevant verifications on the specification. Finally, we will be able to group individual specifications into modules, which is what will accomplish our task.

2.2 Formalism

We use the declarative TRIO formalism as well as its extension TRIO$^+$ as the specification language for expressing the properties of our Controller[1]. The following short introduction to the formalism focuses on TRIO and addresses TRIO$^+$ only very briefly, assuming that the reader is already familiar with object-oriented concepts.

The presentation of the formalism is informal and mainly syntactic. It is based on [4]. For a formal presentation of TRIO, in particular of its semantic foundations, we refer the reader to [3].

[1] Due to the declarative nature of TRIO, resulting specifications are pure data. Declarative formalisms are therefore considered less suitable for expressing *control* on data than operational formalisms, which provide such control by construction (see [1] for an axiomatisation of the operational formalism of Petri nets in terms of TRIO).

2.2.1 Description

2.2.1.1 TRIO

TRIO is a first-order temporal logic intended to be used for the specification of real-time systems. It provides support to a variety of validation techniques like specification testing, simulation, and property proof. TRIO$^+$ adds the ability to construct specifications of complex systems in a systematic and modular way using constructs for hierarchical system decomposition and object-oriented concepts like inheritance and genericity.

The ingredients of a TRIO formula are variable, function, and predicate names; the propositional connectors $\neg$ and $\rightarrow$, as well as the derived ones $\wedge$, $\vee$, $\leftrightarrow$, etc; and the quantifiers $\exists$ and $\forall$.

Variables, functions and predicates may be time-dependent (TD) or time-independent (TI). Moreover, TRIO is a typed language defining suitable domains for variables (domain of legal values), functions (domain-range pair) and predicates (a domain of legal values for each argument). For variables, there is the distinguished *temporal* domain, which may be a domain of integers, rationals or reals.

A TRIO formula is constructed in the usual inductive way. A *term* is either a variable name or a function name followed by a possibly empty list of terms of the correct type. An *atomic formula* is a predicate name followed by a possibly empty list of terms of the correct type. A *formula* is either a term or several atomic formulae linked by logical connectors.

There are two predefined primitive temporal operators Past and Futr. The value of $\mathsf{Futr}(f,\ \Delta)$, resp. $\mathsf{Past}(f,\ \Delta)$, is the value of the formula f at a distance of Δ time units in the future, resp. past, with respect to the current time instant (left implicit). In our specifications, we will use two more temporal operators, which are defined in terms of Futr, resp. Past:

$$\begin{aligned} \mathsf{WithinF}(f,\ \Delta) &\stackrel{def}{=} \exists(\Delta' \leq \Delta)(\mathsf{Futr}(f,\ \Delta')) \\ \mathsf{WithinP}(f,\ \Delta) &\stackrel{def}{=} \exists(\Delta' \leq \Delta)(\mathsf{Past}(f,\ \Delta')) \end{aligned}$$

As in classical first-order logic, one can define the concepts of satisfiability and validity of a TRIO formula, with respect to suitable interpretations. Based on an interpretation S, an evaluation function $S_i(f)$ is defined that assigns a truth value to any formula f of the language at any time instant i of the temporal domain T.

A TRIO formula f is said to be *temporally satisfiable* in an interpretation S if $S_i(f) = \mathsf{true}$ for some $i \in T$. In such a case, we say that the interpretation constitutes a *model* for the formula. A formula is said to be *temporally valid* if it is true in every time instant of the temporal domain. Finally, a TRIO formula is said to be *time invariant* if it either is temporally valid, or cannot be satisfied in any interpretation.

A TRIO formula is *classically closed* if all of its (time-independent) variables are quantified; it is *temporally closed* if it does not contain time-dependent variables or predicates, if it has either Som ('at some instant in the past or the future') or Alw ('at every instant in the past and the future') as the outermost operator, or, finally, if it results from the propositional composition or classical closure of temporally closed formulae. It can be proven that any temporally closed formula is time invariant; this can be understood intuitively by considering that the operators Som and Alw provide a way to quantify existentially resp. universally the current time. For this reasons, we define a *specification* of a system as a TRIO formula that is closed, both classically and temporally.

2.2.1.2 TRIO$^+$

A TRIO$^+$ specification is built by defining suitable classes. A class is a set of TRIO axioms describing the system to be specified. Classes may be simple or structured, may be generic, and may be organised in inheritance hierarchies.

A simple class is a group (body) of TRIO axioms, preceded by the declaration (header) of all occurring predicates, variables, and functions. Structured classes are classes that have components, called *modules*. A class may not be used to declare its own modules, neither directly nor indirectly, so recursive class definitions are ruled out. The temporal domain must be the same for a class and for its modules.

An instance of a class is a model for the axioms of the class, i.e., an interpretation for all entities that have been declared in the header, such that all axioms are true. A class declaration is thus the intensional representation of all its models, and an object of the class is identified with the entire history of its evolution or, in other terms, by the series of values of its state variables. Modules may not be used directly in axioms because they are not logical symbols such as variable, function or predicate names.

2.2.2 Use

We are going to use TRIO formulae for expressing different kinds of properties in our specification process, i.e., requirements, assumptions, and designs.

Requirements

A requirement is according to [11] a "property expected of the system". And it "is expressed as a constraint over the system behaviours, i.e., the states of the system over time".

It is reasonable to look at the specification of our Controller as a set of requirements, at least in the first place. The set should be checked for satisfiability because an implementation (a program) for a set of logical formulae exists if and only if the set is satisfiable.

There are different kinds of requirements. For example, we can have functional and non-functional requirements as well as safety and utility requirements. Functional requirements specify operations of the system — non-functional requirements don't, they describe other properties of the system such as timing constraints. Safety requirements express properties that the system must have in order to guarantee safety in some well-defined sense. Utility requirements improve system performance, they do not add any new operations to the system specification.

Assumptions

In Section 1.1 we have already made some (informal) assumptions. Assumptions about some subsystem must be made whenever the subsystem escapes our control. This is to guarantee a safe and valid requirements formalisation. Also, [11]:

> For a design to implement a requirement it is necessary to make assumptions about both the environment in which the system will operate and the physical properties of the implementation.

Designs

With requirements and assumptions we do only state *what* properties we want the system to have, resp. *what* properties we assume the environment of the system to have. We can refine our system specification in a subsequent phase of designs. According to [11] this "involves making choices and taking decisions about *how* requirements are to be met".

The last phase in a specification process is the implementation of the system specification. However, this is beyond the scope of this work. In such a phase, we would have to talk about control plans and scheduling issues such as priority policies etc.

Verification and Validation

TRIO has a proof system, which is described in [1]. It allows verification of the correctness of a refinement step made in the process of specification refinement. A refinement step consists in the derivation of a design from a requirement, and possibly of some assumptions. Correctness of a refinement step is defined as follows (see [11]):

Definition 4 (Refinement Step Correctness) Let $D_i, \ldots, D_j$ denote designs, $A_i, \ldots, A_j$ assumptions, and R_i a requirement. Then correctness is defined to be a ternary relation between the set of all possible designs, the set of all possible assumptions and the set of all possible requirements such that $(D_i \wedge \ldots \wedge D_j \wedge \ldots \wedge A_i \wedge \ldots \wedge A_j) \rightarrow R_i$

Thus, proving correctness of a refinement step is the same as proving a logical implication. However, it is not sure that all such implications may be proven with the proof system: since it provides a metric on time, the proof system is necessarily incomplete.

Validity of a specification can be examined using history checking, which is described in [2]. Specification validation must, of course, be made together with the client of the specified system, since only the client can judge if the specification actually meets his intention of the desired system.

2.3 Formalisation Guidelines

We formalise our abstract system model in two steps: first, we formalise its macro behaviour, sketched out informally in Section 1.1 and call the result abstract model skeleton. This is done in Section 4.1. Second, we formalise the micro behaviour and explicit the inconsistency predicate. This is done in Section 4.2.

The formalisation basically consists in finding a mapping between model-related terms and specification-related terms. In [2] we finde some indications on how this is accomplished in the general case (see Table 2.1):

Model Term	**Specification Term**
physical components and immutable relations among them	individual constants, TI predicates
temporary relations, events	TD predicates
values, measures of physical quantities subject to change	TD variables
predefined fixed operations of the system	functions
properties of the specified system	TI variables

Table 2.1: From Model Terms to Specification Terms

In our case, this means that events and actions will be expressed as time-dependent predicates, and that each internal state will be formed by a set of variables, whose values may be generated by functions.

2.4 Conventions

2.4.1 Lexical

We give the regular expressions describing the lexema that express in the forthcoming specifications the concepts that are specific to our model. The name descriptions are typeset in the same font as will be the corresponding concept names. Meta symbols are '⟨', '⟩', '::=' and '+'.

$$
\begin{aligned}
\langle\text{Name}\rangle &::= \langle\text{AlphabeticCharacter}\rangle^{+} \\
\langle\text{Predicate Name}\rangle &::= \langle\mathsf{Name}\rangle \\
\langle\text{Event Name}\rangle &::= \langle\text{Predicate Name}\rangle\uparrow \\
\langle\text{Clearing Event Name}\rangle &::= \overline{\langle\text{Event Name}\rangle} \\
\langle\text{Action Name}\rangle &::= \langle\text{Predicate Name}\rangle\downarrow \\
\langle\text{Variable Name}\rangle &::= \langle\mathit{Name}\rangle \\
\langle\text{Function Name}\rangle &::= \langle\text{Name}\rangle \\
\langle\text{Type Name}\rangle &::= \langle\mathbf{Name}\rangle \\
\langle\text{String Value}\rangle &::= \langle\mathtt{Name}\rangle \\
\langle\text{Set Name}\rangle &::= \langle\mathbb{NAME}\rangle
\end{aligned}
$$

Terms of the inconsistency predicate will gradually appear as we write down the specifications for our Controller. A term t of the inconsistency predicate will be surrounded by a frame, just like: $\boxed{t}$.

We adopt the following naming conventions for name descriptions, i.e., full names: event name descriptions will be such that a nominal part precedes a past particle, and action name descriptions will be such that a verb in imperative mode precedes a nominal part.

2.4.2 Syntactical

During specification development, we will use so-called *specification schemata* and so-called *requirement-design tables*. A specification schema is a tabular presentation of some generic specification. The specification is generic in the sense that it has a list of formal parameters associated to it, which can be assigned to a list of actual parameters to generate a concrete (parameterless) specification.

Figure 2.1 shows the layout of a specification schema. There is a field for the specification schema name, ⟨Name⟩, used as the symbol refering to the field ⟨Specification⟩, which will contain a set of TRIO formulae. The field ⟨Full Name⟩ gives a name description for the field ⟨Name⟩, the field ⟨Abstract⟩ will contain a short description of the concern of the specification schema, the field ⟨Specification Alphabet⟩ will give the definitions of all the symbols (event, action, variable and function names) that are used in the formulae of the field ⟨Specification⟩, and the field ⟨Informal Description⟩ will contain an explanation of the intension created by the formulae of the field ⟨Specification⟩. We draw **!**
the reader's attention to the fact that every such formula has to be thought of as being embraced by the operator Alw.

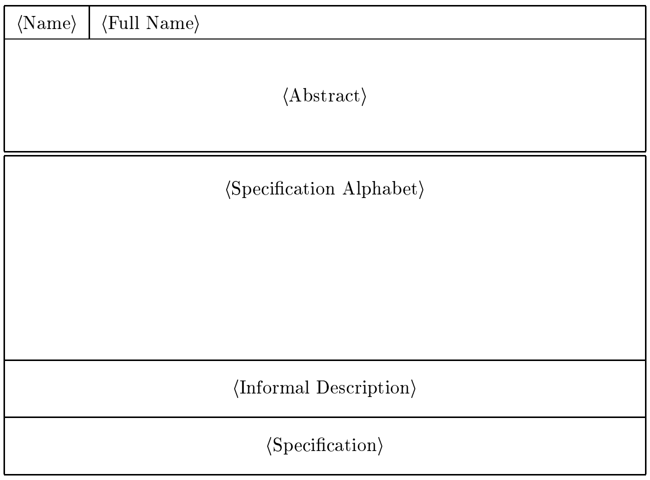


Figure 2.1: Specification Schema Layout

When instantiating specification schemata, which are actually *sets* of predicates, although we write them as if they were predicates, we will sometimes also use so-called *conditional axiom definitions*, i.e., axioms that are only 'there' when some condition holds.

Definition 5 (Conditional Axiom Definition) Let c be some condition, and $\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$ a set of predicates, for example an instance of a specification schema. Then the conditional axiom definition operator $\Rightarrow$ is defined as follows:

$$\begin{aligned} \mathsf{c} \Rightarrow \{\mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{n}\} &\stackrel{def}{=} \mathsf{c} \rightarrow \bigwedge\{\mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{n}\} \\ &\stackrel{def}{=} \mathsf{c} \rightarrow \mathsf{p}_1 \wedge \ldots \wedge \mathsf{p}_\mathsf{n} \end{aligned}$$

Figure 2.2 shows the layout of a requirement-design table, which is meant to visualise the relation between some requirement and the designs that are derived from it. Since specification development is a step-wise process, designs can in their turn become requirements, where from some other design(s) can again be derived.

$\langle$Requirement$\rangle$
$\langle$Design$_1\rangle$
$\vdots$
$\langle$Design$_n\rangle$

Figure 2.2: Requirement-Design Pair(s) Table Layout

2.4.3 Semantic

Definition 6 (Event or Action Predicate) If an event or action predicate is true at a certain moment t, then it has been false just before t and will be false just after t. Thus, events and actions are always time-dependent (TD) predicates.

Events are supposed to originate from *sensors* and indicate a significant change of value in some *time-dependent* variable. Such a variable contains *actual* values, i.e., values that are measured by some sensor.

Actions are destined to be carried out by *actors*, which may read some *prescribed* value from some *time-independent* variable.

Updates to a variable are supposed to have happened when the associated external event predicate becomes true. This means that we make abstraction **!** of internal events. Moreover, updates to variables that refer to some mode are *silent*, i.e., they do not cause the creation of an external event.

Chapter 3

Extent

Our specification extends in two dimensions, i.e., in breadth and in depth. We call the specification breadth *scope* and the specification depth *abstraction hierarchy*. Scope answers the question “How much does the specification cover?” whereas with an abstraction hierarchy we answer the question “How detailed is the specification?”.

3.1 Scope

The specification covers the *normal behaviour* of the implemented Controller. The latter is supposed to consist of a control logic (hardware loaded with the control program) and a peripheral system (sensors, actors and the subsystem connecting them with the control logic), which also consists of hardware. The Controller behaves normally if and only if there is neither a hardware defect in, nor an external source of disturbance (electro-magnetic radiation, for example) acting on, neither the control logic nor its peripheral system.

Abnormal behaviour is not covered in our specification. According to [11], abnormal behaviour can be divided into *exceptional* and *catastrophic behaviour*. Normal behaviour together with exceptional behaviour defines *acceptable behaviour* (see Figure 3.1), which is the behaviour that does not violate safety requirements. On the other hand, catastrophic behaviour does violate safety requirements, which is, of course, unacceptable.

We see that abnormal behaviour is actually due to the existence of *hardware*, which is also the reason why our specification does not cover it because the hardware to be used is not known to us and, also, because the covering of abnormal behaviour lies beyond the scope of this work.

Some of the abnormal behaviour, i.e., exceptional behaviour, we are able to manage, i.e., ensure the safety requirements in spite of its existence by making appropriate *failure hypotheses* (see [11]). In case of catastrophic behaviour, however, the best we may be able do is to signal its occurrence. Consequently, we must admit that not all safety hazards may be excluded for sure, and neither may some of the unacceptable behaviour!

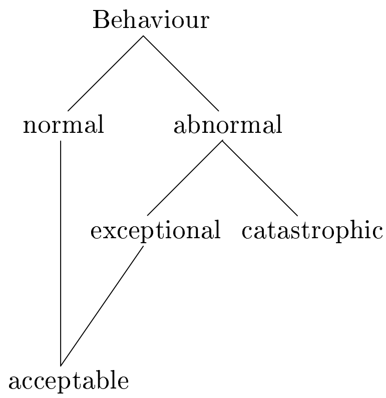


Figure 3.1: Behaviour Classes

3.2 Abstraction Hierarchy

A specification may *cover* a certain class of behaviours, it is, however, not sufficient to *ensure* them. With requirement and design specifications, expressed in a certain specification language, we are still in the *declarative* realm and have just accomplished the first step in the development of the desired system.

In order to get a working system, we must move into the *operational* realm, i.e., implement the set of design specifications using an (imperative) programming language. It is only by ensuring that during this move the stated design specifications are preserved that we ensure that the desired system eventually exhibits the specified behaviours.

Following the definition of designs in Section 2.2.2, we can say that designs contain more information than requirements because the first ones are more specific than the second ones. An implementation has to be even more specific, and therefore contains even more information because we must give an algorithm that effectively does what the designs state that should be done.

Thus, we can say that requirements are more abstract than their corresponding designs, which are in turn more abstract than any possible corresponding implementation. This is our abstraction hierarchy[1].

[1] The higher degree of abstraction that distinguishes a requirement from its corresponding set of designs is revealed in the more frequent use (in first-order formulae explicit and in temporal formulae implicit) of the quantifier $\exists$ in requirement formulae. Thus, the existential quantifier can be seen as a sort of *abstraction operator* (see [9])

Part II

The Specification

Chapter 4

Formalisation

4.1 Abstract Model Skeleton

In this section, we give the 'skeleton' of the formal semantics of our abstract system model (presented informally in Section 1.1). The given semantics is a skeleton in the sense that it defines some global but incomplete structure. In our case, the structure is global with respect to the Environment and the Controller. It is incomplete because it does not address the definition of the inconsistency predicate (addressed in Section 4.2). The abstract model skeleton can be expressed by the following, single formula:

Definition 7 (Abstract Model Skeleton) Let $\mathsf{ee}\!\uparrow$ denote an external event, $\mathsf{a}\!\downarrow$ the action corresponding to $\mathsf{ee}\!\uparrow$, Δ_{01} the time bound within which update of internal Controller states must have occurred, Δ_0 the time bound within which $\mathsf{a}\!\downarrow$ must have been issued by the Controller such that $\Delta_{01} < \Delta_0$, and IP the inconsistency predicate. Then the axiom defining formally the semantics of the abstract model skeleton is:

$$\begin{aligned}\mathsf{Alw}(((\mathsf{ee}\!\uparrow \wedge \mathsf{WithinF}(\mathsf{IP},\ \Delta_{01})) \rightarrow \mathsf{WithinF}(\mathsf{a}\!\downarrow,\ \Delta_0)) \wedge \\ (\mathsf{a}\!\downarrow \rightarrow \mathsf{WithinP}(\mathsf{ee}\!\uparrow \wedge \mathsf{WithinF}(\mathsf{IP},\ \Delta_{01}),\ \Delta_0)))\end{aligned}$$

It happens that we have actually also expressed formally the intended meaning of Requirements 1 and 2 (stated informally in Section 1.1): in the context of the above definition the operator Alw captures the idea of receptivity whereas the operators $\mathsf{WithinF}$ and $\mathsf{WithinP}$ capture the idea of reactivity.

4.2 Inconsistency Predicate

In Section 4.2.1 we state *what* information we are going to use in the development of the inconsistency predicate, then, *how* we are going to process this information, and finally, *what shape* the resulting output will have.

Section 4.2.2 contains some judgements about the original specification document, which are meant to justify the choice of our own specification layout (laid down in Section 4.2.3).

We proceed in our specification process by defining some specification schemata (see Section 4.2.4), which we shall instantiate with different parameters in Section 4.2.5 to generate most of the necessary specifications.

At that point, we will be able to write out the inconsistency predicate, which is what we do in Section 4.2.6.

4.2.1 Specification Process

We draw the necessary input information mainly from two parts of the original specification document, i.e., the part called 'Specifica dei requisiti funzionali del sistema controllo marcia treno' and Chapter 6, called 'Funzioni', of the part 'Specifica dei requisiti del sistema controllo marcia treno, Volume I' (see Appendix A). The latter contains an informal description of the 'functions' the Controller is supposed to implement.

In order to define a valid inconsistency predicate we must understand as clearly as possible the intentional content of the original specification. Therefore, we will closely follow its presentation, which is *functional*, in the first place (in this chapter), but will move towards a *state-based* presentation — conforming to our abstract model — later on (in Chapter 6).

The shift towards a state-based presentation suggests structuring the specification according to *data*, i.e., state variables, by the means of *classes* (in the sense of object-orientation). However, already before 'shifting', some structuring according to *operations*, i.e., actions, can be done by the means of mathematical *sets* and based on the 4CPQ.

The output of our specification process, i.e., the inconsistency predicate, will take the form of a disjunction of certain left parts of implications or equivalences of design formulae. Such a left part will have the form of a conjunction of an event predicate and a predicate on some internal state(s) of the Controller.

The inconsistency predicate will be true whenever there occurs an external event causing the corresponding predicate on internal states to become true, which is how we want the inconsistency predicate to behave.

4.2.2 Specification Sources

In our view, Chapter 6 of the part 'Specifica dei requisiti del sistema controllo marcia treno, Volume I' contains some confusion with respect to at least three concerns: first, the concern what a function actually is, second, the concern what logical level the function belongs to, and third, the concern of separation of phases in specification development.

As for the first concern, we claim that not all of the 'functions' are actually functions but rather are data *for* functions {'velocità di rilascio', 'grado di frenatura', 'pendenza della linea', 'peso assiale', 'metro corrente'}, event sources that *generate* that data for functions {'segnali fissi'}, or *modes* of functioning {'linee con BACC', 'Supero Rosso', 'Diagnostica'}.

With respect to the second concern, we claim that some of the remaining 'real' functions actually belong to different logical levels, i.e., object and meta level. The object level being the logical domain of functions that are related to the treatment of external events (our central concern) {'Indebito superamento di un segnale a via impedita', 'Prosecuzione itinerario', 'Ingresso su binario di ricevimento ingombro o corto', 'Protezione di parauti', 'Itinerari deviati', 'Superamento della velocità massima della linea', 'Marcia su binario illegale'}, and the meta level being the logical domain of functions that are related to *managing* the *set* of functions (the functionality) of the Controller {'Gestione della uscita dal sistema controllo marcia treno', 'Controllo della corretta operatività del P.d.M rispetto alla inserzione disinserzione della RSC'}. We think that these meta functions are very much implementation and use oriented, which is why we do not consider them in the phases we are addressing in the specification process.

Finally, by the third concern we mean more precisely that the above mentioned Chapter 6 mixes the phase of specification of requirements and the phase of specification of designs within the process of specification development, and, moreover, it does it without actually distinguishing explicitly between the two phases. Our specification obviously will distinguish explicitly between specification elements that represent requirements and specification elements that represent designs by labelling respective formulae as such.

4.2.3 Specification Layout

We structure our specification according to an operation-based scheme. The operations being — from the point of view of the Environment — all external events, and — from the point of view of the Controller — all actions. We develop the scheme step-by-step and based on the set of external events:

From Chapter 6 of [6] we extract the principal external events — they are the ones that cause or may cause the inconsistency predicate to evaluate to true:

Definition 8 (Inconsistency Causes) The set of inconsistency causes, $\mathbb{IC}$, is the set of external events that cause or may (depending on the content of the internal states of the Controller) cause the inconsistency predicate to evaluate to true right after the update of internal states:

$\mathsf{die}\!\uparrow$ deviated itinerary entered!

$\mathsf{ite}\!\uparrow$ illegal track entered!

$\mathsf{rlv}\!\uparrow$ red light violated!

$\mathsf{saa}\!\uparrow$ shock absorbers ahead!

$\mathsf{mascr}\!\uparrow$ maximum speed constraint received!

$\mathsf{miscr}\!\uparrow$ minimum speed constraint received!

$\mathsf{ste}\!\uparrow$ short track entered!

$\mathsf{toe}\!\uparrow$ track with obstacles entered!

$\mathsf{mtfrr}\!\uparrow$ movement to 'forward' request received!

$\mathsf{mtbrr}\!\uparrow$ movement to 'backward' request received!

so that $\mathbb{IC} \stackrel{def}{=} \{\mathsf{die}\!\uparrow, \mathsf{ite}\!\uparrow, \mathsf{rlv}\!\uparrow, \mathsf{saa}\!\uparrow, \mathsf{mascr}\!\uparrow, \mathsf{miscr}\!\uparrow, \mathsf{ste}\!\uparrow, \mathsf{toe}\!\uparrow, \mathsf{mtfrr}\!\uparrow, \mathsf{mtbrr}\!\uparrow\}$.

Looking at these inconsistency causes, we observe that most of them must actually have some counterpart that will announce the ceasing of the situation that made the Environment issuing the external event in the first place. We call such a counterpart *clearance event* and define the set of events that cause them as follows:

Definition 9 (Clearable Events) The set of clearable events, $\mathbb{CE}$, is the set of external events with a corresponding clearance event:

$$\mathbb{CE} \stackrel{def}{=} \{\mathsf{die}\uparrow, \mathsf{ite}\uparrow, \mathsf{mascr}\uparrow, \mathsf{miscr}\uparrow, \mathsf{ste}\uparrow, \mathsf{toe}\uparrow\}$$

Since not all external events have a corresponding clearance event, we also define:

Definition 10 (Unclearable Events) The set of unclearable events, $\mathbb{UE}$, is the set of external events that do not have a corresponding clearance event:

$$\mathbb{UE} \stackrel{def}{=} \{\mathsf{rlv}\uparrow, \mathsf{saa}\uparrow, \mathsf{mtfrr}\uparrow, \mathsf{mtbrr}\uparrow\}$$

According to our model, external events that cause an inconsistency between the internal states of the Controller make the latter issue an action in the domain of the Environment. Actions directed to the train are supposed to control the train, which may become faster or slower, or may even change direction. At the highest level of abstraction, no other state changes are of interest.

In fact, all of these state changes make necessary a modification of the quantities of acceleration or deceleration, which is the reason why we call the quantities 'speed' and 'direction' *primary*, and the quantities 'acceleration' and 'deceleration' *secondary*.

Looking at our model from the point of view of cause and effect, we can say that inconsistency causes (obviously) *cause* Controller actions, which in turn *effect* a change in the primary controlled quantities. Due to this causality chain, which relates primary controlled quantities to inconsistency causes, we have considered it reasonable to partition further inconsistency causes into four subsets according to the change their corresponding actions incur in the primary controlled quantities (see Figure 4.1 for the final structure of the class of external events):

$$\mathbb{SD} \stackrel{def}{=} \{\mathsf{die}\uparrow, \mathsf{ite}\uparrow, \mathsf{rlv}\uparrow, \mathsf{saa}\uparrow, \mathsf{mascr}\uparrow, \mathsf{ste}\uparrow, \mathsf{toe}\uparrow\} \qquad \text{(speed decrease)}$$

$$\mathbb{SI} \stackrel{def}{=} \{\mathsf{miscr}\uparrow\} \qquad \text{(speed increase)}$$

$$\mathbb{DB} \stackrel{def}{=} \{\mathsf{mtbrr}\uparrow\} \qquad \text{(direction to backward)}$$

$$\mathbb{DF} \stackrel{def}{=} \{\mathsf{mtfrr}\uparrow\} \qquad \text{(direction to forward)}$$

4.2.4 Specification Schemata

4.2.4.1 Composite Types and Special Operators

We present some preliminary definitions of concepts that will be used in the following specification schemata:

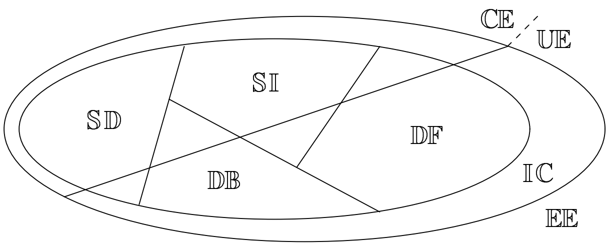


Figure 4.1: Event Classes

Definition 11 (Train Type) The train type is defined as the cross-product of the type of its breaks (**bkt**), its length (**tnl**), the maximum speed it may assume (**mtns**) and its weight on axis (**tnw**):

$$\mathbf{tnt} \stackrel{def}{=} \mathbf{bkt} \times \mathbf{tnl} \times \mathbf{mtns} \times \mathbf{tnw}$$

Definition 12 (Track Type) The track type is defined as the cross-product of its break ratio (**tkbr**) and its inclination (**tki**):

$$\mathbf{tkt} \stackrel{def}{=} \mathbf{tkbr} \times \mathbf{tki}$$

Definition 13 (Thresh-Hold Operators) Let a and b denote variables of one and the same type $\mathbf{T}$, and $0 < th < 1$ denote a real number that is fixed according to $\mathbf{T}$. Then the thresh-hold operators are defined as follows:

$$\begin{aligned} a <_{th} b &\stackrel{def}{=} a < (1 - th)b \\ a =_{th} b &\stackrel{def}{=} a < (1 + th)b \wedge a > (1 - th)b \\ a >_{th} b &\stackrel{def}{=} a > (1 + th)b \end{aligned}$$

The thresh-hold operators allow us to conveniently model *margins of tolerance*. Operand values that lie within these margins will cause the corresponding operator predicate to evaluate to false, which in turn will inhibit the Controller to react to the external event(s) that caused the change in the operand values in the first place. We deliberately leave the definition of these thresh holds to the people who are competent to do this.

4.2.4.2 Schemata

In the following schemata, the abbreviations 'e-t' and 'e-c' stand for 'environment-train' and 'environment-context'. Further, we draw the reader's attention to the fact that in the following, we have not specified the functions cbf and ctc for the same reason as we did not specify thresh holds previously.

$\mathsf{GAPSR}(\mathsf{sc}\!\uparrow, ps, \mathrm{cbf}, \mathsf{cs}\!\uparrow)$	General Action Pattern for Speed Reduction

Specifies a general (re)action pattern for the controller — and thus a behaviour pattern for the train — upon reception of some implicit or explicit maximum speed constraint (ps) from the context.

Predicate Name	**Event**	
	Source	**Description**
$\mathsf{sc}\!\uparrow$	e-c	speed constraint received!
$\mathsf{td}\!\uparrow$	e-t	traction deactivated!
$\mathsf{ba}\!\uparrow$	e-t	breaks activated!
$\mathsf{bd}\!\uparrow$	e-t	breaks deactivated!
$\mathsf{cs}\!\uparrow$	e-c	clearance for speed constraint received!

Predicate Name	**Action**
$\mathsf{dt}\!\downarrow$	deactivate traction!
$\mathsf{ab}\!\downarrow$	activate breaks!
$\mathsf{db}\!\downarrow$	deactivate breaks!

Variable Name	**State**		
	Type		**Description**
as	e-t	TD	actual speed (km/h)
ps	e-c	TI	prescribed speed (km/h)
tnt	e-t	TI	train type
tkt	e-c	TD	track type
abf	e-t	TD	actual break force (kN)
pbf	e-t	TI	prescribed break force (kN)

Function Name	**Value**
cbf	calculated break force (kN) in function of as, ps, tnt and tkt

$$
\begin{array}{rcl}
\mathsf{aux}(\Delta) & \stackrel{def}{=} & \mathsf{Futr}(\mathsf{ab}\!\downarrow \wedge\, pbf = \mathrm{cbf}(as, ps, tnt, tkt),\ \Delta_1 + \Delta) \\
\boxed{\mathsf{sc}\!\uparrow \wedge\, as >_{th} ps} & \leftrightarrow & \mathsf{aux}(0) \\
\mathsf{ab}\!\downarrow & \rightarrow & \mathsf{Futr}(\mathsf{dt}\!\downarrow,\ \Delta_{211}) \wedge \mathsf{Futr}(\mathsf{td}\!\uparrow,\ \Delta_{21}) \wedge \Delta_{211} < \Delta_{21} \\
\mathsf{ab}\!\downarrow \wedge\, as >_{th} ps & \leftrightarrow & \mathsf{Futr}(\mathsf{ba}\!\uparrow \wedge\, abf =_{th} pbf,\ \Delta_2) \wedge \Delta_2 > \Delta_{21} \wedge \mathsf{aux}(\Delta_2) \\
\boxed{\mathsf{cs}\!\uparrow \wedge\, as <_{th} ps} & \leftrightarrow & pbf = 0 \wedge \mathsf{Futr}(\mathsf{db}\!\downarrow,\ \Delta_3) \\
\mathsf{db}\!\downarrow & \rightarrow & \mathsf{Futr}(\mathsf{bd}\!\uparrow \wedge\, abf = 0,\ \Delta_4)
\end{array}
$$

Specification Schema 1: General Action Pattern for Speed Reduction

$\mathsf{GAPSI}(\mathsf{sc}\uparrow, ps, \mathrm{ctc}, \mathsf{cs}\uparrow)$	General Action Pattern for Speed Increase

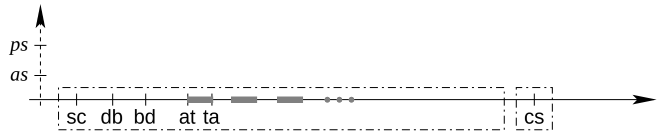
Specifies a general (re)action pattern for the controller — and thus a behaviour pattern for the train — upon reception of some implicit or explicit minimum speed constraint (ps) from the context.

Predicate Name	**Event**	
	Source	**Description**
$\mathsf{sc}\uparrow$	e-c	speed constraint received!
$\mathsf{bd}\uparrow$	e-t	breaks deactivated!
$\mathsf{ta}\uparrow$	e-t	traction activated!
$\mathsf{cs}\uparrow$	e-c	clearance for speed constraint received!

Predicate Name	**Action**
$\mathsf{db}\downarrow$	deactivate breaks!
$\mathsf{at}\downarrow$	activate traction!

Variable Name	**State**		
	Type		**Description**
as	e-t	TD	actual speed (km/h)
ps	e-c	TI	prescribed speed (km/h)
tnt	e-t	TI	train type
tkt	e-c	TD	track type
atc	e-t	TD	actual traction current (A)
ptc	e-t	TI	prescribed traction current (A)

Function Name	**Value**
ctc	calculated traction current (A) in function of as, ps, tnt and tkt



$$\boxed{\mathsf{sc}\uparrow \wedge as <_{th} ps} \leftrightarrow \mathsf{Futr}(\mathsf{db}\downarrow,\ \Delta'_{100}) \wedge \mathsf{Futr}(\mathsf{bd}\uparrow,\ \Delta'_{10}) \wedge \Delta'_{100} < \Delta'_{10} \wedge \mathsf{Futr}(\mathsf{at}\downarrow \wedge ptc = \mathrm{ctc}(as, ps, tnt, tkt),\ \Delta'_1) \wedge \Delta'_{10} < \Delta'_1$$

$$\boxed{\mathsf{at}\downarrow \wedge as <_{th} ps \wedge \neg\mathsf{cs}\uparrow} \leftrightarrow \mathsf{Futr}(\mathsf{ta}\uparrow \wedge atc =_{th} ptc,\ \Delta'_2) \wedge \mathsf{Futr}(\mathsf{at}\downarrow \wedge ptc = \mathrm{ctc}(as, ps, tnt, tkt),\ \Delta'_3)$$

Specification Schema 2: General Action Pattern for Speed Increase

4.2.5 Specifications

In the following, we present the specifications for actions that the Controller takes in response to external events. The specifications are grouped according to the set each external event belongs to, i.e., $\mathbb{SD}$, $\mathbb{SI}$, $\mathbb{DB}$ or $\mathbb{DF}$.

We have extensively used instantiation of specification schemata by varying their respective set of actual parameters. For example, it has been possible to use the same specification schema for clearable as well as for unclearable events by introducing a clearance event with the constant value `true` for unclearable events (`true` being the neutral element of conjunction).

Another case of specification schema instantiation is related to specifications about explicit speed constraints. In such instantiations, an explicit value for the parameter 'prescribed speed', ps, does not appear in the parameter list since it is assumed that the variable has already this value when the event predicate becomes true (see Section 2.4.3). So, no explicit value is needed nor desirable at the time of specification instantiation.

A final remark as far as the presentation of requirement-design pairs is concerned: it is actually inverse to the order by which we have obtained them. In fact, the intention of each design maps more or less to the intention created by some function description in the original specification document (the inverse is not true). This is a typical case of *reverse engineering* by which we get from something more concrete to something more abstract.

4.2.5.1 Speed Decrease

Table 4.1: Requirement-Design Pairs for Speed Decrease

Requirement 4

$$\forall(\mathsf{ee}\uparrow \in \mathbb{SD})\exists\Delta((\mathsf{ee}\uparrow \wedge as >_{th} ps) \rightarrow \mathsf{WithinF}(as =_{th} ps,\ \Delta))$$

Design 1 (Circulation along Deviated Itinerary) Let lt (line type) denote the string variable whose value is equal to the current line code. Then the specification for circulation along deviated itineraries is as follows:

$$\begin{aligned}
lt = \mathtt{C30} \quad &\Rightarrow \quad \mathsf{GAPSR}(\mathsf{sc}\uparrow \stackrel{def}{=} \mathsf{die}\uparrow,\ ps \stackrel{def}{=} 30,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{die}},\ \mathsf{cs}\uparrow \stackrel{def}{=} \overline{\mathsf{die}\uparrow})\\
lt = \mathtt{C60} \quad &\Rightarrow \quad \mathsf{GAPSR}(\mathsf{sc}\uparrow \stackrel{def}{=} \mathsf{die}\uparrow,\ ps \stackrel{def}{=} 60,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{die}},\ \mathsf{cs}\uparrow \stackrel{def}{=} \overline{\mathsf{die}\uparrow})\\
lt = \mathtt{C100} \quad &\Rightarrow \quad \mathsf{GAPSR}(\mathsf{sc}\uparrow \stackrel{def}{=} \mathsf{die}\uparrow,\ ps \stackrel{def}{=} 100,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{die}},\ \mathsf{cs}\uparrow \stackrel{def}{=} \overline{\mathsf{die}\uparrow})
\end{aligned}$$

Design 2 (Circulation on Illegal Track)

$$\mathsf{GAPSR}(\mathsf{sc}\uparrow \stackrel{def}{=} \mathsf{ite}\uparrow,\ ps \stackrel{def}{=} 30,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{ite}},\ \mathsf{cs}\uparrow \stackrel{def}{=} \overline{\mathsf{ite}\uparrow})$$

Table 4.1: Requirement-Design Pairs for Speed Decrease

Design 3 (Red Light Violation) Let sr denote the boolean variable whose value is true when the special operation mode 'Supero Rosso' has been activated and false otherwise. Then the specification for red light violation is as follows:

$$\begin{array}{rcl} \mathsf{TrainTrip} & \stackrel{def}{=} & \mathsf{GAPSR}(\mathsf{sc}\!\uparrow \stackrel{def}{=} \mathsf{rlv}\!\uparrow,\ ps \stackrel{def}{=} 5,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{rlt}},\ \mathsf{cs}\!\uparrow \stackrel{def}{=} \mathsf{true}) \\ sr = \mathsf{false} & \Rightarrow & \mathsf{GAPSR}(\mathsf{sc}\!\uparrow \stackrel{def}{=} \mathsf{rlv}\!\uparrow,\ ps \stackrel{def}{=} 0,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{rlt}},\ \mathsf{cs}\!\uparrow \stackrel{def}{=} \mathsf{true}) \\ sr = \mathsf{true} & \Rightarrow & \mathsf{TrainTrip} \end{array}$$

Design 4 (Approach of Shock Absorbers)

$$\mathsf{GAPSR}(\mathsf{sc}\!\uparrow \stackrel{def}{=} \mathsf{saa}\!\uparrow,\ ps \stackrel{def}{=} 5,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{saa}},\ \mathsf{cs}\!\uparrow \stackrel{def}{=} \mathsf{true})$$

Design 5 (Reception of Maximum Speed Constraint)

$$\mathsf{GAPSR}(\mathsf{sc}\!\uparrow \stackrel{def}{=} \mathsf{mascr}\!\uparrow,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{mascr}},\ \mathsf{cs}\!\uparrow \stackrel{def}{=} \overline{\mathsf{mascr}\!\uparrow})$$

Design 6 (Circulation on Short Track)

$$\mathsf{GAPSR}(\mathsf{sc}\!\uparrow \stackrel{def}{=} \mathsf{ste}\!\uparrow,\ ps \stackrel{def}{=} 10,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{ste}},\ \mathsf{cs}\!\uparrow \stackrel{def}{=} \overline{\mathsf{ste}\!\uparrow})$$

Design 7 (Circulation on Track with Obstacles)

$$\mathsf{GAPSR}(\mathsf{sc}\!\uparrow \stackrel{def}{=} \mathsf{toe}\!\uparrow,\ ps \stackrel{def}{=} 30,\ \mathrm{cbf} \stackrel{def}{=} \mathrm{cbf}_{\mathrm{toe}},\ \mathsf{cs}\!\uparrow \stackrel{def}{=} \overline{\mathsf{toe}\!\uparrow})$$

4.2.5.2 Speed Increase

Table 4.2: Requirement-Design Pair for Speed Increase

Requirement 5

$$\forall(\mathsf{ee}\!\uparrow \in \mathbb{SI})\exists\Delta((\mathsf{ee}\!\uparrow \wedge\ as <_{th} ps) \rightarrow \mathsf{WithinF}(as =_{th} ps,\ \Delta))$$

Design 8 (Reception of Minimum Speed Constraint)

$$\mathsf{GAPSI}(\mathsf{sc}\!\uparrow \stackrel{def}{=} \mathsf{miscr}\!\uparrow,\ \mathrm{ctc} \stackrel{def}{=} \mathrm{ctc}_{\mathrm{miscr}},\ \mathsf{cs}\!\uparrow \stackrel{def}{=} \overline{\mathsf{miscr}\!\uparrow})$$

4.2.5.3 Forward Direction

Requirement 6 (Movement to 'Forward') Let $\mathsf{mtfrr}\!\uparrow$ denote the event indicating that a request to move the train in forward direction has been made, and *dir* denote the string variable indicating the actual direction of train movement. Then the specification for movement to 'forward' is as follows:

$$\boxed{\mathsf{mtfrr}\!\uparrow \wedge\, as = 0} \rightarrow dir = \texttt{forward}$$

4.2.5.4 Backward Direction

Requirement 7 (Movement to 'Backward') Let $\mathsf{mtbrr}\!\uparrow$ denote the event indicating that a request to move the train in backward direction has been made, and *mr* denote the boolean variable that takes the value true when the train is in manoeuvring mode and false otherwise. Then the specification for movement to 'backward' is as follows:

$$\boxed{\mathsf{mtbrr}\!\uparrow \wedge\, mr = \mathsf{true} \wedge as = 0} \rightarrow dir = \texttt{backward}$$

4.2.6 Explicitly It

We write out the inconsistency predicate, which says *when* the Controller reacts to external events. It does however not say *how* it reacts to them. If we want to know also that, then we must look at the whole specification, which is situated at a deeper level of abstraction.

$$
\begin{array}{lllrl}
\mathsf{IP} & \stackrel{def}{=} & ((\mathsf{die}\!\uparrow \wedge\, as >_{th} ps) \quad \vee & (\overline{\mathsf{die}\!\uparrow} \wedge as <_{th} ps)) & \vee \\
& & ((\mathsf{ite}\!\uparrow \wedge\, as >_{th} ps) \quad \vee & (\overline{\mathsf{ite}\!\uparrow} \wedge as <_{th} ps)) & \vee \\
& & ((\mathsf{rlv}\!\uparrow \wedge\, as >_{th} ps) \quad \vee & (\overline{\mathsf{rlv}\!\uparrow} \wedge as <_{th} ps)) & \vee \\
& & ((\mathsf{saa}\!\uparrow \wedge\, as >_{th} ps) \quad \vee & (\overline{\mathsf{saa}\!\uparrow} \wedge as <_{th} ps)) & \vee \\
& & ((\mathsf{mascr}\!\uparrow \wedge\, as >_{th} ps) \quad \vee & (\overline{\mathsf{mascr}\!\uparrow} \wedge as <_{th} ps)) & \vee \\
& & ((\mathsf{ste}\!\uparrow \wedge\, as >_{th} ps) \quad \vee & (\overline{\mathsf{ste}\!\uparrow} \wedge as <_{th} ps)) & \vee \\
& & ((\mathsf{toe}\!\uparrow \wedge\, as >_{th} ps) \quad \vee & (\overline{\mathsf{toe}\!\uparrow} \wedge as <_{th} ps)) & \vee \\
& & ((\mathsf{miscr}\!\uparrow \wedge\, as <_{th} ps) \quad \vee & (\mathsf{at}\!\downarrow \wedge\, as <_{th} ps \wedge \neg\overline{\mathsf{miscr}\!\uparrow})) & \vee \\
& & (\mathsf{mtfrr}\!\uparrow \wedge\, as = 0) & & \vee \\
& & (\mathsf{mtbrr}\!\uparrow \wedge\, mr = \mathsf{true} \wedge as = 0) & &
\end{array}
$$

Chapter 5

Verifications

5.1 About the Specification

The most important verification to make about a specification is to check whether or not it is *satisfiable*. If the set of logical formulae constituting the specification is not satisfiable, then no implementation, i.e., a model, be it abstract (our system model) or concrete (some control program), can exist for it[1]. As for our specification, we make the following claim:

Claim 1 (Satisfiability) The specification consisting of the set of formulae given by Design 1–Design 8, Requirement 6 and Requirement 7 is satisfiable.

Satisfiability of a set of time-dependent logical formulae is decidable in special cases such as if the corresponding time domain is finite. In that case, satisfiability can be checked algorithmically by using some appropriate tool or, also, by hand. Such a tool exists for TRIO, yet it is not fully automatic and some experience of using it is needed. On the other hand, performing the satisfiability check by hand is arduous and time-consuming. In fact, it is easier to give some informal arguments for Claim 1.

Our argument is constructive and based on so-called *causal implications*. With $\mathsf{SomF}(f) \stackrel{def}{=} \exists\Delta(\Delta > 0 \wedge \mathsf{Futr}(f,\ \Delta))$ we define:

Definition 14 (Causal implication) Let c and e denote some time-dependent predicates. Then a causal implication, or a *relation of cause and effect*, denoted $\rightsquigarrow$, between the cause c and the effect e, is defined as follows:

$$\mathsf{c} \rightsquigarrow \mathsf{e} \stackrel{def}{=} \mathsf{Alw}(\mathsf{c} \rightarrow \mathsf{SomF}(\mathsf{e}))$$

A causal implication is a special case of a temporal implication, and has the evident but interesting property of being always satisfiable. Further, we use causal implications to define the concept of so-called *causal chains*.

[1]In fact, the situation we face has arisen due to the *intensional* character of our 'definition' in Section 4.1 (formal semantics). An intensional definition is a definition that defines some mathematical object(s) (models in our case) by an enumeration of the properties (TRIO formulae in our case) the object(s) is, resp. are, supposed to have. If there is a contradiction in these properties, then nothing has been defined with the 'definition'.

Definition 15 (Causal Chain) A causal chain is a chain of causal implications: $\mathsf{c}_1 \rightsquigarrow \mathsf{c}_2 \rightsquigarrow \cdots \rightsquigarrow \mathsf{c}_\mathsf{n} \rightsquigarrow \mathsf{e}_\mathsf{n} \stackrel{def}{=} \mathsf{c}_1 \rightsquigarrow \mathsf{e}_1 \wedge \mathsf{c}_2 \rightsquigarrow \mathsf{e}_2 \wedge \cdots \wedge \mathsf{c}_\mathsf{n} \rightsquigarrow \mathsf{e}_\mathsf{n}$, where for all indices i we have $c_i \stackrel{def}{=} e_{i-1}$.

By construction, and due to transitivity of causal implications, causal chains are always satisfiable.

It happens that each of our specification schemata basically specifies two such *generic* causal chains[2] (see Specification Schemata 1 and 2, where each causal chain is represented by a dash-dotted rectangle in the field ⟨Informal Description⟩). So, our specification schemata are somehow already 'half-way' satisfiable. We say 'half-way' because the causal chains each specification schema specifies may actually overlap. Thus, we might have accidentally specified some implicit contradiction. More precisely, for some indices i and j, and for some time instant t, we might have e_i from one causal chain and e_j from the other causal chain such that $\not\models_t \mathsf{e}_\mathsf{i} \wedge \mathsf{e}_\mathsf{j}$.

For Specification Schema 1 this means that $pbf = 0$ might contradict $pbf =$ cbf, and that $abf =_{th} pbf$ might contradict $abf = 0$ at some time instant. However, this is impossible because then, $as >_{th} ps$ and $as <_{th} ps$ would have to be true at the same time, which is physically impossible.

For Specification Schema 2, it is evident that there cannot exist any contradiction since the causal chain consists of a single cause without any corresponding explicit effect.

We may conclude that both our specification schemata generate satisfiable sets of formulae, i.e., designs. The remaining question is now whether or not the conjunction of designs is still satisfiable. The answer is 'yes' because the particular causal chain pairs are disjoint with respect to the external events that induce them. So, no contradiction can be introduced by joining individual designs.

The remaining formulae in our specification are Requirement 6 and Requirement 7. It is easy to see that their conjunction is also satisfiable, and so is their conjunction with the set of design formulae, which means that the entire specification is satisfiable.

5.2 Within the Specification

5.2.1 Proof System

The TRIO proof system consists of a set of *general axioms*, a set of *temporal axioms*, and a single *derivation rule*, i.e., modus ponens (see [1]). General axioms are those axioms that are shared with any first-order theory with equality. We will not restate them here, assuming that the reader is already familiar with them. Temporal axioms are those axioms that are specific to TRIO:

TA1 $\models \mathsf{Dist}(f,\ 0) \leftrightarrow f$

TA2 $\models \mathsf{Dist}(f,\ \Delta_1 + \Delta_2) \leftrightarrow \mathsf{Dist}(\mathsf{Dist}(f,\ \Delta_1),\ \Delta_2)$

[2] We note that the set of all those causes that come first in the specified causal chains is identical to the set of inconsistency predicate terms. We may therefore say that every specified causal chain is *induced* by exactly one inconsistency term.

TA3 $\models \mathsf{Dist}(f_1 \to f_2,\ \Delta) \leftrightarrow (\mathsf{Dist}(f_1,\ \Delta) \to \mathsf{Dist}(f_2,\ \Delta))$

TA4 $\models \mathsf{Dist}(\neg f,\ \Delta) \leftrightarrow \neg\mathsf{Dist}(f,\ \Delta)$

TA5 $\models f \to \mathsf{Alw}(f)$ if f is time-independent

where $\mathsf{Dist}(f,\ \Delta)$ intuitively means that f holds at an instant laying Δ time units in the future (if $\Delta > 0$) or in the past (if $\Delta < 0$) with respect to the current time value, which, as we already know, is implicit in the formula. Assuming Dist primitive, we obtain:

$$\begin{aligned}
\mathsf{Futr}(f,\ \Delta) &\stackrel{def}{=} \Delta \geq 0 \wedge \mathsf{Dist}(f,\ \Delta) \\
\mathsf{Past}(f,\ \Delta) &\stackrel{def}{=} \Delta \geq 0 \wedge \mathsf{Dist}(f,\ -\Delta)
\end{aligned}$$

In addition, we will need the following theorem (proven in [1]), **Ti**, and lemma:

Ti $\vdash \mathsf{Alw}(f) \to \mathsf{Dist}(f,\ \Delta)$

Lemma 1 (Dist/$\wedge$-Distributivity) Given the formulae f_1 and f_2, and the time constant Δ, we have:

$$\vdash \mathsf{Dist}(f_1 \wedge f_2,\ \Delta) \leftrightarrow \mathsf{Dist}(f_1,\ \Delta) \wedge \mathsf{Dist}(f_2,\ \Delta)$$

Proof :

$$\begin{aligned}
\mathsf{Dist}(f_1 \wedge f_2,\ \Delta) &\leftrightarrow \neg\neg\mathsf{Dist}(f_1 \wedge f_2,\ \Delta) && \\
&\leftrightarrow \neg\mathsf{Dist}(\neg(f_1 \wedge f_2),\ \Delta) && \textbf{TA4} \\
&\leftrightarrow \neg\mathsf{Dist}(\neg f_1 \vee \neg f_2,\ \Delta) && \\
&\leftrightarrow \neg\mathsf{Dist}(f_1 \to \neg f_2,\ \Delta) && \\
&\leftrightarrow \neg(\mathsf{Dist}(f_1,\ \Delta) \to \mathsf{Dist}(\neg f_2,\ \Delta)) && \textbf{TA3} \\
&\leftrightarrow \neg(\mathsf{Dist}(f_1,\ \Delta) \to \neg\mathsf{Dist}(f_2,\ \Delta)) && \textbf{TA4} \\
&\leftrightarrow \neg(\neg\mathsf{Dist}(f_1,\ \Delta) \vee \neg\mathsf{Dist}(f_2,\ \Delta)) && \\
&\leftrightarrow \neg\neg(\mathsf{Dist}(f_1,\ \Delta) \wedge \mathsf{Dist}(f_2,\ \Delta)) && \\
&\leftrightarrow \mathsf{Dist}(f_1,\ \Delta) \wedge \mathsf{Dist}(f_2,\ \Delta) && \Box
\end{aligned}$$

5.2.2 Proofs

We have to prove correctness of two series of refinement steps. The first occurs between Requirement 4 and Design 1–Design 7 and the second occurs between Requirement 5 and Design 8.

Since all designs have been generated from *generic specifications* (specification schemata), we are actually also in a position to give *generic proofs*. In our case, a generic proof is a proof that validates a whole *series* of refinement steps rather than only a single refinement step. A series of refinement steps is such that all steps within the series originate from the same requirement but end in different designs (created from the same specification schema).

In order to prove correctness of our refinement steps, we must make some appropriate assumptions about the *effectiveness* of certain Controller actions in the domain of the Environment. In fact, the Controller alone cannot ensure that breaking effectively produces a decrease in train speed, or that an increase in the traction current effectively produces an increase in train speed because these issues are beyond its influence. We assume:

Assumption 3 (Breaking Is Effective)

$$\mathsf{A}_3 \stackrel{def}{=} \mathsf{Alw}(\mathsf{ab}{\downarrow} \rightarrow \mathsf{SomF}(\neg(as >_{th} ps)))$$

Assumption 4 (Activating Traction Is Effective)

$$\mathsf{A}_4 \stackrel{def}{=} \mathsf{Alw}(\mathsf{at}{\downarrow} \rightarrow \mathsf{SomF}(\neg(as <_{th} ps)))$$

Comparing the above assumption formulae with the stated requirement formulae, we observe that, just like requirement formulae, assumption formulae basically consist of some causal implication. Thus, at first sight, there is no difference between assumption and requirement formulae. However, a closer examination (see Table 5.2.2) of both kinds of formulae reveals that there actually *is* some difference.

Occurrence	**Requirements**	**Assumptions**
External event predicate names	only in the cause	no
Action predicate names	no	only in the cause
Names of variables with prescribed or actual values	in cause and/or effect	only in the effect

Table 5.1: Requirement and Assumption Formulae: Syntactical Differences

In fact, assumption and requirement formulae can be distinguished according to the location (cause resp. effect) certain symbols occupy within the implication. Quite naturally external event names only occur in the cause of a requirement, whereas action predicate names only occur in the cause of an assumption. We say 'naturally' because the essential role of a requirement is to describe what property has to be met *after* some external event has been registered by the Controller, whereas the essential role of an assumption is to describe what impact the Controller must be able to measure *after* the issuing of the corresponding (re)action in the domain of the Environment.[3]

Having made the necessary assumptions, we are able to prove correctness of the aforementioned refinement steps (see Proof 1 and Proof 2). In the proofs[4], we adopt the following conventions: first, we use axioms, our lemma, and our

[3] To be complete, we restate the essential role of designs: it is to describe what actions the Controller takes in order to meet the corresponding requirement.

[4] have been typeset using TEX macros from Prof. Jacques Zahnd (EPFL).

theorem directly in formulae containing Futr and not Dist. Second, justifications for application of first-order derivation rules are either omitted, or reduced to an indication of the lines at which the corresponding 'input' judgements are located.

1°	$\bigwedge \mathsf{GAPSR} \land \mathsf{A}_3$	hyp
2°	$\mathsf{sc}\!\uparrow\, \in \mathbb{SD}$	hyp
3°	$\mathsf{sc}\!\uparrow \land\ as >_{th} ps$	hyp
4°	$\mathsf{Alw}((\mathsf{sc}\!\uparrow \land\ as >_{th} ps) \leftrightarrow \mathsf{Futr}(\mathsf{ab}\!\downarrow \land\ pbf = \mathrm{cbf},\ \Delta_1))$	1°, a fortiori
5°	$\mathsf{Futr}((\mathsf{sc}\!\uparrow \land\ as >_{th} ps) \leftrightarrow \mathsf{Futr}(\mathsf{ab}\!\downarrow \land\ pbf = \mathrm{cbf},\ \Delta_1),\ 0)$	4°, **Ti**
6°	$(\mathsf{sc}\!\uparrow \land\ as >_{th} ps) \leftrightarrow \mathsf{Futr}(\mathsf{ab}\!\downarrow \land\ pbf = \mathrm{cbf},\ \Delta_1)$	5°, **TA1**
7°	$\mathsf{Futr}(\mathsf{ab}\!\downarrow \land\ pbf = \mathrm{cbf},\ \Delta_1)$	3°, 6°, mod pons
8°	$\mathsf{Futr}(\mathsf{ab}\!\downarrow,\ \Delta_1) \land \mathsf{Futr}(pbf = \mathrm{cbf},\ \Delta_1)$	7°, Lemma 1
9°	$\mathsf{Alw}(\mathsf{ab}\!\downarrow\ \to \mathsf{SomF}(\neg(as >_{th} ps)))$	1°, a fortiori
10°	$\mathsf{Futr}(\mathsf{ab}\!\downarrow\ \to \mathsf{SomF}(\neg(as >_{th} ps)),\ \Delta_1)$	9°, **Ti**
11°	$\mathsf{Futr}(\mathsf{ab}\!\downarrow,\ \Delta_1) \to \mathsf{Futr}(\mathsf{SomF}(\neg(as >_{th} ps)),\ \Delta_1)$	10°, **TA1**
12°	$\mathsf{Futr}(\mathsf{SomF}(\neg(as >_{th} ps)),\ \Delta_1)$	8°, 11°, mod pons
13°	$\mathsf{Futr}(\exists\Delta(\Delta > 0 \land \mathsf{Futr}(\neg(as >_{th} ps),\ \Delta)),\ \Delta_1)$	12°, def
14°	$\exists\Delta(\mathsf{Futr}(\Delta > 0 \land \mathsf{Futr}(\neg(as >_{th} ps),\ \Delta),\ \Delta_1))$	13°
15°	$\exists\Delta(\mathsf{Futr}(\Delta > 0,\ \Delta_1) \land \mathsf{Futr}(\mathsf{Futr}(\neg(as >_{th} ps),\ \Delta),\ \Delta_1))$	14°, Lemma 1
16°	$\exists\Delta(\mathsf{Futr}(\Delta > 0,\ \Delta_1) \land \mathsf{Futr}(\neg(as >_{th} ps),\ \Delta + \Delta_1))$	15°, **TA2**
17°	$\exists\Delta(\mathsf{Futr}(\Delta > 0,\ \Delta_1)) \land \exists\Delta(\mathsf{Futr}(\neg(as >_{th} ps),\ \Delta + \Delta_1))$	16°
18°	$\exists\Delta(\mathsf{Futr}(\neg(as >_{th} ps),\ \Delta + \Delta_1))$	17°
19°	$\mathsf{Futr}(\neg(as >_{th} ps),\ \Delta + \Delta_1)$	hyp
20°	$\exists\Delta'(\mathsf{Futr}(\neg(as >_{th} ps),\ \Delta'))$	19°
21°	$\exists\Delta'(\Delta' \leq \Delta' \land \mathsf{Futr}(\neg(as >_{th} ps),\ \Delta'))$	20°
22°	$\mathsf{WithinF}(\neg(as >_{th} ps),\ \Delta')$	21°, def
23°	$\mathsf{WithinF}(\neg(as >_{th} ps),\ \Delta')$	18°, 19°, 22°
24°	$(\mathsf{sc}\!\uparrow \land\ as >_{th} ps) \to \mathsf{WithinF}(\neg(as >_{th} ps),\ \Delta')$	3°, 23°
25°	$\exists\Delta((\mathsf{sc}\!\uparrow \land\ as >_{th} ps) \to \mathsf{WithinF}(\neg(as >_{th} ps),\ \Delta))$	24°
26°	$\mathsf{sc}\!\uparrow\, \in \mathbb{SD} \to \exists\Delta((\mathsf{sc}\!\uparrow \land\ as >_{th} ps) \to \mathsf{WithinF}(\neg(as >_{th} ps),\ \Delta))$	2°, 25°
27°	$\forall(\mathsf{sc}\!\uparrow\, \in \mathbb{SD})(\exists\Delta((\mathsf{sc}\!\uparrow \land\ as >_{th} ps) \to \mathsf{WithinF}(\neg(as >_{th} ps),\ \Delta)))$	26°
28°	$\forall(\mathsf{ee}\!\uparrow\, \in \mathbb{SD})(\exists\Delta((\mathsf{sc}\!\uparrow \land\ as >_{th} ps) \to \mathsf{WithinF}(\neg(as >_{th} ps),\ \Delta)))$	27°
29°	$(\bigwedge \mathsf{GAPSR} \land \mathsf{A}_3) \to$ $\forall(\mathsf{ee}\!\uparrow\, \in \mathbb{SD})(\exists\Delta((\mathsf{sc}\!\uparrow \land\ as >_{th} ps) \to \mathsf{WithinF}(\neg(as >_{th} ps),\ \Delta)))$	1°, 28°

Proof 1: Refinement Correctness for Speed Decrease Designs

1°	$\bigwedge \mathsf{GAPSI} \wedge \mathsf{A}_4$	hyp
2°	$\mathsf{sc}\uparrow \in \mathbb{SI}$	hyp
3°	$\mathsf{sc}\uparrow \wedge as <_{th} ps$	hyp
4°	$\mathsf{Alw}((\mathsf{sc}\uparrow \wedge as <_{th} ps) \leftrightarrow (\mathsf{Futr}(\mathsf{db}\downarrow,\ \Delta'_{100}) \wedge \mathsf{Futr}(\mathsf{bd}\uparrow,\ \Delta'_{10}) \wedge$ $\Delta'_{100} < \Delta'_{10} \wedge \mathsf{Futr}(\mathsf{at}\downarrow \wedge ptc = \mathrm{ctc},\ \Delta'_1) \wedge \Delta'_{10} < \Delta'_1))$	1°, a fortiori
5°	$\mathsf{Futr}((\mathsf{sc}\uparrow \wedge as <_{th} ps) \leftrightarrow (\mathsf{Futr}(\mathsf{db}\downarrow,\ \Delta'_{100}) \wedge \mathsf{Futr}(\mathsf{bd}\uparrow,\ \Delta'_{10}) \wedge$ $\Delta'_{100} < \Delta'_{10} \wedge \mathsf{Futr}(\mathsf{at}\downarrow \wedge ptc = \mathrm{ctc},\ \Delta'_1) \wedge \Delta'_{10} < \Delta'_1),\ 0)$	4°, **Ti**
6°	$(\mathsf{sc}\uparrow \wedge as <_{th} ps) \leftrightarrow (\mathsf{Futr}(\mathsf{db}\downarrow,\ \Delta'_{100}) \wedge \mathsf{Futr}(\mathsf{bd}\uparrow,\ \Delta'_{10}) \wedge$ $\Delta'_{100} < \Delta'_{10} \wedge \mathsf{Futr}(\mathsf{at}\downarrow \wedge ptc = \mathrm{ctc},\ \Delta'_1) \wedge \Delta'_{10} < \Delta'_1)$	5°, **TA1**
7°	$\mathsf{Futr}(\mathsf{at}\downarrow \wedge ptc = \mathrm{ctc},\ \Delta'_1)$	3°, 6°, mod pons
8°	$\mathsf{Futr}(\mathsf{at}\downarrow,\ \Delta'_1) \wedge \mathsf{Futr}(ptc = \mathrm{ctc},\ \Delta'_1)$	7°, Lemma 1
9°	$\mathsf{Alw}(\mathsf{at}\downarrow \rightarrow \mathsf{SomF}(\neg(as <_{th} ps)))$	1°, a fortiori
10°	$\mathsf{Futr}(\mathsf{at}\downarrow \rightarrow \mathsf{SomF}(\neg(as <_{th} ps)),\ \Delta'_1)$	9°, **Ti**
11°	$\mathsf{Futr}(\mathsf{at}\downarrow,\ \Delta'_1) \rightarrow \mathsf{Futr}(\mathsf{SomF}(\neg(as <_{th} ps)),\ \Delta'_1)$	10°, **TA1**
12°	$\mathsf{Futr}(\mathsf{SomF}(\neg(as <_{th} ps)),\ \Delta'_1)$	8°, 11°, mod pons
13°	$\mathsf{Futr}(\exists\Delta(\Delta > 0 \wedge \mathsf{Futr}(\neg(as <_{th} ps),\ \Delta)),\ \Delta'_1)$	12°, def
14°	$\exists\Delta(\mathsf{Futr}(\Delta > 0 \wedge \mathsf{Futr}(\neg(as <_{th} ps),\ \Delta),\ \Delta'_1))$	13°
15°	$\exists\Delta(\mathsf{Futr}(\Delta > 0,\ \Delta'_1) \wedge \mathsf{Futr}(\mathsf{Futr}(\neg(as <_{th} ps),\ \Delta),\ \Delta'_1))$	14°, Lemma 1
16°	$\exists\Delta(\mathsf{Futr}(\Delta > 0,\ \Delta'_1) \wedge \mathsf{Futr}(\neg(as <_{th} ps),\ \Delta + \Delta'_1))$	15°, **TA2**
17°	$\exists\Delta(\mathsf{Futr}(\Delta > 0,\ \Delta'_1)) \wedge \exists\Delta(\mathsf{Futr}(\neg(as <_{th} ps),\ \Delta + \Delta'_1))$	16°
18°	$\exists\Delta(\mathsf{Futr}(\neg(as <_{th} ps),\ \Delta + \Delta'_1))$	17°
19°	$\mathsf{Futr}(\neg(as <_{th} ps),\ \Delta + \Delta'_1)$	hyp
20°	$\exists\Delta''(\mathsf{Futr}(\neg(as <_{th} ps),\ \Delta''))$	19°
21°	$\exists\Delta''(\Delta'' \leq \Delta'' \wedge \mathsf{Futr}(\neg(as <_{th} ps),\ \Delta''))$	20°
22°	$\mathsf{WithinF}(\neg(as <_{th} ps),\ \Delta'')$	21°, def
23°	$\mathsf{WithinF}(\neg(as <_{th} ps),\ \Delta'')$	18°, 19°, 22°
24°	$(\mathsf{sc}\uparrow \wedge as <_{th} ps) \rightarrow \mathsf{WithinF}(\neg(as <_{th} ps),\ \Delta'')$	3°, 23°
25°	$\exists\Delta((\mathsf{sc}\uparrow \wedge as <_{th} ps) \rightarrow \mathsf{WithinF}(\neg(as <_{th} ps),\ \Delta))$	24°
26°	$\mathsf{sc}\uparrow \in \mathbb{SI} \rightarrow \exists\Delta((\mathsf{sc}\uparrow \wedge as <_{th} ps) \rightarrow \mathsf{WithinF}(\neg(as <_{th} ps),\ \Delta))$	2°, 25°
27°	$\forall(\mathsf{sc}\uparrow \in \mathbb{SI})(\exists\Delta((\mathsf{sc}\uparrow \wedge as <_{th} ps) \rightarrow \mathsf{WithinF}(\neg(as <_{th} ps),\ \Delta)))$	26°
28°	$\forall(\mathsf{ee}\uparrow \in \mathbb{SI})(\exists\Delta((\mathsf{sc}\uparrow \wedge as <_{th} ps) \rightarrow \mathsf{WithinF}(\neg(as <_{th} ps),\ \Delta)))$	27°
29°	$(\bigwedge \mathsf{GAPSI} \wedge \mathsf{A}_4) \rightarrow$ $\forall(\mathsf{ee}\uparrow \in \mathbb{SI})(\exists\Delta((\mathsf{sc}\uparrow \wedge as <_{th} ps) \rightarrow \mathsf{WithinF}(\neg(as <_{th} ps),\ \Delta)))$	1°, 28°

Proof 2: Refinement Correctness for Speed Increase Designs

Chapter 6

Modularisation

As announced in Section 4.2.1, the specification can be structured, first, further, and second, according to data. For this purpose, we refine the data space introduced in Section 1.1 and relate the different components of the space among each other.

As a result, we obtain a *state-based* (in conformance with our abstract system model) and *layered* data model for our Controller. Finally, we show how the introduced concepts map into TRIO$^+$ concepts thus providing indirectly a modular version of the Controller specification. We say 'indirectly' because we do not write out explicitly the modular specification, but merely describe how the modular version can be obtained from the non-modular version.

The data space of Section 4.2.1 can be refined by sub-typing the type 'operation data' with the type 'mode-related data' and with the type 'temporary data'. Modes of operation are the modes due to the various line types (lt), the manoeuvring mode (mr) and the mode 'Supero Rosso' (sr). Temporary data are all variables containing actual and prescribed values ($as, ps, abf, pbf, atc, ptc$) as well as the variable indicating the track type (tkt) and the variable indicating the current direction of train movement (dir). We end up with four time-dependent data types, i.e., configuration, operation, mode-related and temporary data.

Since our data consists of state variables, we can observe that, put in the above ordering, the time-dependent data types are actually ordered according to two criteria: first according to the frequency updates in state variables would typically occur during operation, and second, according to visibility of data. By the first, we mean that configuration data is, for obvious reasons, least often updated and temporary data is, of course, most often updated during operation. By the latter, we mean that the scope of configuration data must be global with respect to the rest of data for obvious reasons, but not the other way round. The analogous is true for operation and mode-related data.

Our abstract system model is state-based grouping information from the context and the train in two separate internal states. Space-dependent data thus occupies the first dimension in our two-dimensional data space. Time-dependent data consequently is subordinated to it occupying the second dimension. This means that each internal state will be structured further according to the above introduced layers for time-dependent data. Figure 6 shows this further structuring of data.

Each data layer can be implemented with a TRIO$^+$ class with the inheritance hierarchy creating the necessary visibility ordering of variables. An instance of the class 'ContextState' and an instance of the class 'TrainState' model the internal states of the Controller. Generic classes can be used to model the specification schemata of Section 4.2.4, which can be instantiated to generate Design 1–Design 8. Finally, two separate axioms can be used to express Requirement 6 and Requirement 7.

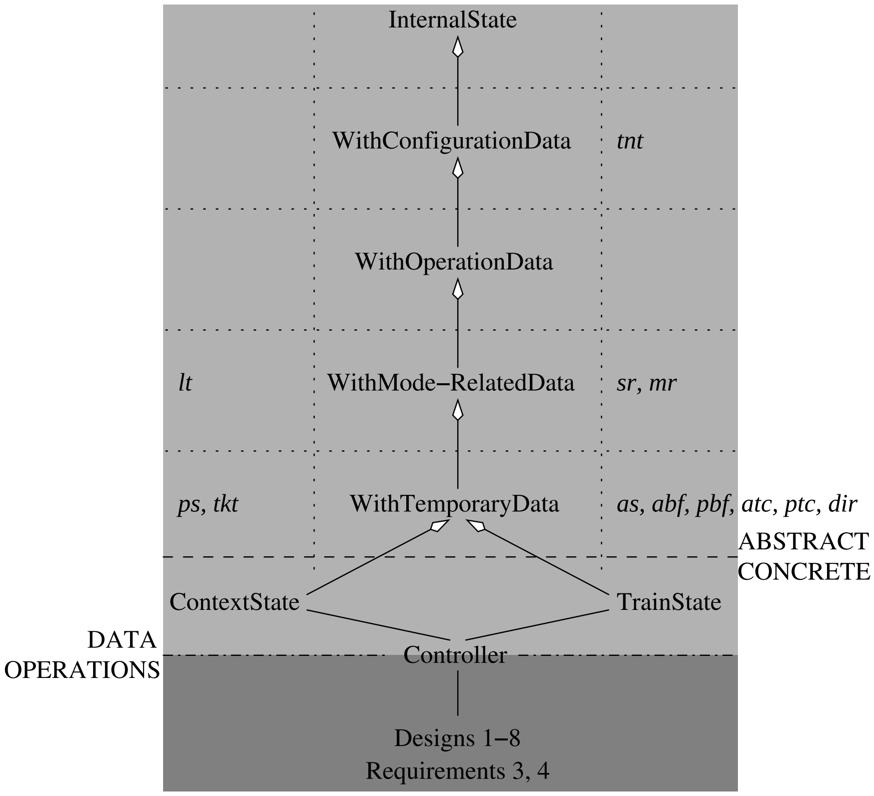


Figure 6.1: A Layered Data Model for Internal States

Part III

Epilogue

Chapter 7

Beyond the Specification

So far, we have considered the Controller for itself. In this chapter, we briefly investigate the relation between the Controller and the rest of the system, the *ground system*, as far as safety in the railway network is concerned.

The most important property of a safe railway network is that there is never any single train crash. A train crash can only occur when there are at least two trains circulating on the same piece of track. We define a piece of track to be a single continuous pair of rails of a certain length delimited on both sides by a semaphore.

If we want to exclude train crashes, then we must disallow that two trains circulate on the same piece of track. This is normally done by switching the semaphores at both of its ends to green whenever there is no train circulating on the track, and by switching the same semaphores to red whenever the track has already been entered by a train. If the whole railway network is systematically constructed using only pieces of tracks protected by semaphores, then it can be guaranteed that crashes cannot occur.

However, by introducing semaphores, we have introduced the risk of *dead-lock*[1]. What about a train that wishes to leave a protected piece of track at a certain end e when at the same time another train is waiting at e for entering the same piece of track? A possible solution for dead-lock is to provide a one-way-only piece of track at e that allows the first train to go round the second train without using the same piece of track. If we systematically provide this kind of 'round-about' then no dead-lock can occur.

Of course, each train has to respect a red semaphore and wait until it becomes green. This is the most important responsibility of the Controller. It guarantees local safety with respect to semaphore *use*. On the other hand, the ground system must effectively provide these semaphores and manage them in a correct way. This is the most important responsibility of the ground system, which guarantees local safety with respect to semaphore *management*.

[1]Absence of dead-lock being a utility requirement (see [10] for more information), we see that **the satisfaction of *safety* requirements may collide with the satisfaction of *utility* requirements**. In other words, design decisions taken in order to satisfy some safety requirement may not generally be taken independently from those design decisions that are taken in order to satisfy utility requirements, and vice versa.

For crash safety, both correct semaphore use *and* management are necessary and only taken *together* are they sufficient to guarantee it. We may conclude that **in a railway network employing semaphores, crash safety is a *global* and *systemic* property**.

Chapter 8

Conclusion

Our work is a modest case of scientific consulting to a large industrial project. In an exploratory effort, it presents an *abstract model*, and based on it, a *formal specification* for the desired real-time train controller specified informally in [5], [6], [7] and [8].

8.1 Discussion

For the conception of the abstract model, some pre-processing of the informal information was found to be necessary. This involved *restructuring*, and making *abstraction* of certain input detail considered to be irrelevant for the level of abstraction at which we judged reasonable to tackle the problem.

The overall approach has been *constructive* or *bottom-up* in the sense that we have used pieces of already existent information to construct a new information representation, our abstract system model. It is *top-down* with respect to the abstraction ordering that exists between requirements and designs.

The restructuring has been made first according to *operations* and then according to *data*. Further, it has entailed a *presentation shift* from the functional view of the controller behaviour, such as it is described in the original specification document, to the state- and event-based view of the controller behaviour, put forward in the present document.

Information which of we have made abstraction has been 'banned' into sharply-delimited *confinements* of some well-defined, *generic concepts*, such as special operators, types, and a few specific functions. Their particular definitions have been left for later phases that will be devoted to the more concrete aspects of specification development.

The specification has been obtained as a result of a *formalisation* process employing the specification language TRIO. With the result, we have been able to perform various *verifications*, thanks to the formal character of the specification. We have given the specification its final, *modular* shape using the object-oriented extension of TRIO, TRIO^{+}. As a plus, we have outlined the role of the controller specification with respect to the rest of the larger system.

During formalisation, we followed some clear *guidelines* specifying controller behaviour at two different levels of abstraction (macro- and micro-behaviour), which we have described separately in two different sections.

Before formalising controller behaviour, we introduced various *conventions*. Lexical conventions served us to name problem-specific concepts (specification alphabet), whereas syntactical conventions served us to relate these concepts among each other in the textual scopes of our specification. Finally, we introduced some semantic conventions in order to be able to relate our abstract model to some supposed environment.

We said to which *extent* we intended to formalise controller behaviour by distinguishing two dimensions within the specification process. We called the first dimension 'scope' (specification breath) and the second dimension 'abstraction hierarchy' (specification depth).

As the most notable feature of the formalisation itself, we cite the **intensive use of *generic treatment* of input information**. In fact, **nearly the whole variety — and apparent complexity — of input information could be formalised using only two generic specifications**.

Generic treatment has also shown considerable benefit for specification verification. It has actually allowed *generic proofs* (carried out formally) of refinement step correctness, and simplified the check of specification satisfiability (justified informally).

In the specification modularisation, we have refined controller states by introducing *visibility layers* for the state variables of the internal controller states. Both, internal states and their layers, can be implemented using TRIO$^+$ *classes*. Controller behaviour formulae can be grouped in an *array* of instances of generic classes (designs) and normal classes (requirements).

8.2 Further Work

Of course, our specification must be refined in breadth as well as in depth. The issue of *abnormal behaviour*, resp. *abstracted information* has to be addressed. Also, the specification in terms of TRIO$^+$ must be written out by using the recipe given in Chapter 6.

Abstracted information is confined in composite types, thresh-holds, generic functions (cbf, ctc), and time bounds, all of which must be defined. Even deeper in the abstraction hierarchy, code generation, i.e., the generation of some control program, has to be undertaken.

Code can, at least theoretically, be obtained from a specification using a *refinement calculus*, which does not yet exist for TRIO at the moment. More traditionally, code can be generated directly by having the intention of the formal specification in mind and checking the validity of the code a posteriori using a *model checker*.

Appendix A

Original Specification Document Table of Contents

DIVISIONE INFRASTRUTTURA	SPECIFICA DEI REQUISITI DI SISTEMA PRELIMINARE	
	Codifica: DI TC SR IS 13 XXX A	FOGLIO 1 di 1

SPECIFICA DEI REQUISITI DI SISTEMA CMT	
Volume	**Titolo**
0	INDICE GENERALE

(*)

Stampa del 02/02/01 8.09

Rev.	Data	Descrizione	Redazione	Verifica Tecnica	Autorizzazione
A	18/04/00	Prima Emissione	Mauro Michelacci	(*)	Michele Mario Elia

INDICE GENERALE

VOLUME 1^ - SRS

SEZIONE A – GENERALITA'

SEZIONE B - SISTEMA

DIVISIONE INFRASTRUTTURA	SPECIFICA DEI REQUISITI DI SISTEMA PRELIMINARE	
SCMT – INDICE GENERALE	Codifica: DI TC SR IS 13 XXX A	FOGLIO 4 di 4

DIVISIONE INFRASTRUTTURA	SPECIFICA DEI REQUISITI DI SISTEMA PRELIMINARE	
SCMT – INDICE GENERALE	Codifica: DI TC SR IS 13 XXX A	FOGLIO 5 di 5

SEZIONE C - INTERFACCIA TRA SST E SSB

SEZIONE D - FORMATO DATI PER LA COMUNICAZIONE FRA SOTTOSISTEMA DI TERRA E DI BORDO

SEZIONE E - SPERIMENTAZIONE

SEZIONE F – OMOLOGAZIONE

SEZIONE G - REALIZZAZIONE

ALLEGATI AL VOLUME 1

- *ANALISI RAMS PRELIMINARE*
  - ALLEGATO I - Preliminary Hazard Analysis
  - ALLEGATO II - Sistema CMT FS, Analisi RAM preliminare
  - ALLEGATO III - Sistema CMT FS, Allocazione dei requisiti quantitativi di sicurezza

VOLUME 2^ - SRS SOTTOSISTEMA DI TERRA

SEZIONE A - SST

VOLUME 3^ - SOTTOSISTEMA DI BORDO

DIVISIONE INFRASTRUTTURA	SPECIFICA DEI REQUISITI DI SISTEMA PRELIMINARE	
SCMT – INDICE GENERALE	Codifica: DI TC SR IS 13 XXX A	FOGLIO 18 di 18

2.6.2 Calcolo del profilo dinamico
2.6.3 Controllo di velocità

2.7 Frenatura di emergenza e modalità' riarmo freno

2.8 Comunicazioni terra-treno
2.8.1 Coding Strategy e Air Gap
2.8.2 Identificazione senso di marcia
2.8.3 Informazioni trasmesse a bordo: messaggi e variabili

2.9 Gestione errori

3 STATI E MODI OPERATIVI

3.1 Definizioni degli stati e modi operativi

3.2 Transizioni fra stati e modi operativi
3.2.1 Transizioni sul modo operativo "manovra"
3.2.2 Transizioni sul modo operativo "vigilante"
3.2.3 Transizioni sul modo operativo "**SCMT**"
3.2.4 Transizioni sul modo operativo "**RSC**"
3.2.5 Transizioni sul modo operativo "**RSC + SCMT**"

4 MANOVRA

4.1 Funzioni in condizioni di normalità

4.2 Funzioni in condizioni di degrado

5 VIGILANTE

5.1 Definizione di treno fermo

5.2 Operatività del vigilante

5.3 Intervento di emergenza **VIG**

5.4 Annullamento dell'intervento di emergenza **VIG**

5.5 Condizioni di degrado

6 SCMT

6.1 Funzioni in condizioni di normalità
6.1.1 Interpretazioni dati
6.1.2 Modalità operative

6.2 Funzioni in condizioni di degrado

DIVISIONE INFRASTRUTTURA	SPECIFICA DEI REQUISITI DI SISTEMA PRELIMINARE	
SCMT – INDICE GENERALE	Codifica: DI TC SR IS 13 XXX A	FOGLIO 19 di 19

7 RSC

7.1 Funzioni in condizioni di normalità
7.1.1 Comunicazione dati
7.1.2 Interpretazione dati
7.1.3 Modalità operative

7.2 Funzioni in condizioni di degrado

8 SCMT e RSC

8.1 Funzioni in condizioni di normalità

8.2 Funzioni in condizioni di degrado

9 REGISTRAZIONE EVENTI E DIAGNOSTICA DI BORDO

9.1 Registrazione eventi su sistema informativo di condotta "**DIS**"
9.1.1 Dati da registrare
9.1.2 Salvataggio delle informazioni
9.1.3 Gestione dei dati memorizzati
9.1.4 Dati trasferiti dal sistema **DIS** al Sottosistema di Bordo (SSB)

9.2 Registrazione su punte tachigrafiche

9.3 Disposizione di bordo
9.3.1 Registrazione eventi sul sistema diagnostico di bordo

9.4 Diagnostica di bordo

10 DIAGNOSTICA DEL SSB

10.1 Diagnostica residente

10.2 Diagnostica **CMT**
10.2.1 Registrazione eventi sulla diagnostica **CMT**
10.2.2 Manutenzione del SSB

10.3 Diagnostica da tool esterno
10.3.1 Diagnostica dinamica
10.3.2 Diagnostica statica

11 PRESTAZIONI

Bibliography

[1] M. Felder, D. Mandrioli, A. Morzenti. *Proving Properties of Real-Time Systems Through Logical Specifications and Petri Net Models*, Report no. 91-072, Politecnico di Milano, 1991.

[2] M. Felder, A. Morzenti. *Validating Real-Time Systems by History-Checking TRIO Specifications*, ACM Transactions on Software Engineering and Methodology, Vol. 3, No. 4, 1994.

[3] A. Morzenti, D. Mandrioli, C. Ghezzi. *A Model Parametric Real-Time Logic*, ACM Transactions on Programming Languages and Systems, Vol. 14, No. 4, 1992.

[4] A. Morzenti, P. San Pietro. *Object-Oriented Logical Specification of Time-Critical Systems*, ACM Transactions on Software Engineering and Methodology, Vol. 3, No. 1, 1994.

[5] M. Michelacci, *Specifica dei requisiti funzionali del sistema controllo marcia treno*, Ferrovie dello Stato, Divisione Infrastruttura, Italy, 1999.

[6] M. Michelacci, *Specifica dei requisiti del sistema controllo marcia treno: Volume I*, Ferrovie dello Stato, Divisione Infrastruttura, Italy, 2000.

[7] M. Michelacci, *Specifica dei requisiti del sistema controllo marcia treno: Volume II*, Ferrovie dello Stato, Divisione Infrastruttura, Italy, 2000.

[8] M. Michelacci, *Specifica dei requisiti del sistema controllo marcia treno: Volume II*, Ferrovie dello Stato, Divisione Infrastruttura, Italy, 2000.

[9] C.A.R. Hoare, J. He. *Unifying Theories of Programming*, Prentice Hall, 1998.

[10] C. Heitmeyer, D. Mandrioli (editors). *Formal Methods for Real-Time Computing*, John Wiley & Sons, 1996.

[11] M. Joseph (editor). *Real-Time Systems. Specification, Verification, Analysis*, Prentice Hall, 1996.