Generic simulation framework applied to Metro Systems

Peter Matlock*

IBM Research, Singapore

2019-01-10

Abstract

City transit systems often include some type of vehicle network which is by design separated from general city traffic; this type of public transit is referred to in this work as a metro system. Such a system typically includes uniform behaviour of vehicles with simple dynamics, and a decoupling from weather, street traffic, and other interference. Not all properties of such a system can be easily observed; obtaining information about these hidden degrees of freedom, and also answering questions about how such a system might behave in given scenarios requires the use of a simulator.

A simulation system with generic properties is described, applicable to systems comprised of objects which move on well-defined trajectories, and have certain containment properties. Interaction degrees of freedom of the objects in such a system include capacities, speeds, origin-destination logic and schedules. The philosophy is primarily that complex aggregate behaviour can be a consequence of the dynamics of objects with simple interaction rules. Supplementing this, the framework includes the ability to interface with external servers in order to model behaviour which cannot be replicated with the fundamental objects available.

Use of this framework is illustrated by modelling, using publicly-available data, the vehicles and passengers of the metro system of a particular world city, providing a so-called digital-twin capability. After calibration of such a model, properties of the metro system such as platform and train occupancies can be investigated, and the effect of incidents such as blocked tracks or system breakdowns can be predicted. The system and associated tools are released as open-source software.

*pmatlock@sg.ibm.com until 2019-01-11; subsequently pwm@induulge.net

Contents

Chapter 1

Introduction

City transit systems often include some type of vehicle network which is by design separated from general city traffic; in the present work, such a network (or possibly a combination of such transit modes) will be generically referred to as a 'metro system'. This term is used somewhat loosely, and the considerations discussed will be variously applicable to systems termed as Subway, LRT, Tram or Regional Rail systems. The characteristics specifically required here are a lack of coupling to street traffic, and (of course related) uniform behaviour of vehicles within the system on a somewhat simpler network than that of city streets. In fact, it is possible to address simulation of a system with these characteristics without specifically speaking of a metro system.

Given identification of these properties, it can be expected that the microscopic modelling of such a system and its users might be generalised in terms of some set of objects which interact via degrees of freedum such as capacity, speed, origin-destination, and schedule. In Chapter 2 such a simulation system is described, first by defining in an abstract way the dynamical objects and behaviour, and then by reference to a concrete implementation. The discussion in Chapter 2 does not make explicit reference to trains, vehicles or passengers; this avoidance of domain specificity is effected in the hope that other systems which may have similar properties of being locally described by one-dimensional progress and a simple set of interaction rules might also be simulated. At the least, it is expected that designing simulation technology around simple abstract rules allows application to metro systems of different types and construction.

In Chapter 5 a basic configuration of objects is assembled to illustrate how a simple system can be realised and its behaviour understood.

In Chapter 6 some general details of metro-system simulation are discussed, and it is shown how the objects defined in Chapter 2 can be assembled to form such a simulation. It is indicated how generic data describing a metro system of interest may be used to construct a model of such a system, and it is discussed how details of interest like vehicle speeds, station layout, the metro network itself and user demand map to the abstract objects in the system.

In Chapter 7, publicly-available data are used to construct an approximate model of the metro system of Madrid, Spain, with the intention of demonstrating a full example system. The resultant model is used in subsequent chapters to illustrate calibration and extraction of results. In Chapter 8, a method is shown of calibrating the parameters of the passenger walking-speed distribution in order to match real-world reference data ('ground truth'). In Chapter 9, a number of results are extracted from a simulation run, demonstrating the system's use as a digital twin, to access degrees of freedom which are inaccessible (or practically inaccessible) in the real world.

The concrete implementation of the simulator is provided in terms of Java source code, examples and auxiliary tools; indeed the primary function of the present document is as comprehensive documentation of this implementation. Programs and data relevant for each chapter are generally contained in a directory beginning with the chapter number. The material is hosted at:
https://github.ibm.com/sgresearch/Zimulator

Chapter 2

Abstracted simulator

A completely general simulation framework, or even a definition of what that might mean beyond perhaps a special-purpose programming language, is well beyond the scope of the present work. In this chapter is presented an abstraction, containing objects sufficient for simulation of a 'metro system' as described in the introduction.

The following features are identified as desirable; in the present case these are far from independent.

Simplicity – Events are to occur in the simulation because the objects within behave that way, not because particular behaviour is forced using specific code. Complex behaviour results from simple well-defined interactions.

Generality – System behaviour should be implemented in terms of high-level abstractions. Behaviour of objects should not be restricted to particular real-world applications, in either name or function. No specific application will constrain the framework, though of course it shall be applicable to metro systems.

Versatility – Metro systems, as a canonical example, come in many sizes and shapes. It is desirable that the framework developed be capable of adapting to these discrepancies, and also to different levels of detail in modelling infrastructure.

The scope of the system will be simulation of container agents, transporting things around networks, in time.

It is attempted to keep specification and implementation conceptually separated; therefore the framework and the specific implementation are described in separate sections in what follows. The source code for the concrete implementation is the subject of Chapter 3.

The framework will accommodate the idea of agents-for-agents; that is, containment of objects which themselves are capable of containment. However useful this idea might prove beyond one or two levels, it is expected to restrict the design so that the system is not over-specific in this regard. This implies also the accommodation of multiple space and time scales.

2.1 Zimulator

The Zimulator is comprised of a system of interacting objects, which may represent several levels of 'containment' (i.e. agents for agents).

It is indeed a zimulator; the abbreviation 'z' is chosen[1] to preface objects, so that no conflation can occur between casual use of a word, and a precisely-defined object in the system. A putative user is free to discuss links, routes, lines, boxes, trains, aeroplanes, containers, cars and anything else; the terms here are words like 'zbox' and 'zlink' which retain precise meanings, defined in the present chapter.

2.1.1 Implementation

Some desirable properties of the implementation are:

- Real-time capability – It should be possible to start and stop simulation, to store and retrieve system states, and to subsequently evolve on various branches from a given state. Optimisation[2] or consideration of 'what-if' scenarios would utilise this feature.
- Flexible specification – A system is described using an input language. It should balance simplicity with flexibility and extensibility.

[1]The letter 'z' is not intended to correspond to any other use of this letter; it is merely chosen since few other words begin with it.

[2]i.e. involving a search of parameter space.

- Flexible outputs – Verbosity of various objects can be selected.
- Speed – Simulation should be sufficiently fast, in terms of execution speed.
- Modularity – However desirable, not all system behaviour can always be encoded in terms of the generic simple rules which will be defined in the present chapter. A method of communication should be implemented so that the core simulation can request decisions, obtain system modifications, and implement other complex behaviour from external *servers*. This mechanism is described in detail below.

'Implementation' will thus refer to a set of several things, beyond simple instantiation of the objects described in Sec. 2.2:

- Situation-handling interface (Sec. 2.3)
- Input language (Sec. 2.4)
- System integration (Sec. 2.5)
- State-storage format (Sec. 2.6)
- Core dynamics module (Sec. 2.7)
- Verbosity and Reporting; output formats (Sec. 2.8)

These are addressed in appropriate sections below.

2.2 Specification

The Zimulator amounts to defined interactions among a number of objects. Some objects are static, providing information (like ZDEMAND, or ZBOXEN describing infrastructure). Most time-dependence in the system is described by the evolution of ZBOXEN, the ways in which they are *contained* in each other and the ways they are *linked* with each other. 'Containment' always refers to direct containement $n = 1$; otherwise it shall be referred to as 'order-n containment'.

Containment is effected by having a given ZBOX contain several others as they undergo time development.

Linking is effected by connection of ZLINKS (which are explicit or implicit), and they control the flow of ZBOXEN.

In what follows, a specification of the properties of each type of object are provided; after these a specification of system interaction rules is provided.

2.2.1 zsystem

A ZSYSTEM Ψ references a number of objects; some are specified as inputs, and some are generated internally during simulation. This is the main object where system-wide parameters are specified.

A ZSYSTEM definition must be given a label. When modifying an extant ZSYSTEM, either the same label can be used, or the label can be omitted (which is unambiguous as there can be only one ZSYSTEM).

Time within the ZSYSTEM is measured in seconds, relative to some base. T_0 specifies the origin for system time. The ZSYSTEM is ignorant of real-world dates and times; all times in the system are simply described with respect to this T_0. t_0 is the simulation starting time. The state of all objects described in [initial] input streams is the state of the system at this time.

- Properties:

  T_0 – origin for simulated time. This is just a coordinate definition; other times are measured and also reported with respect to this – that is, it is not used for any dynamics. Usually this is zero.

  t_0 – start time, relative to T_0.

  t_1 – end time, relative to T_0.

  t – Simulation time, relative to T_0. The state of all objects in the system at time t can be determined.[3]

  Ξ – list(1) of tuples H, where H is the *identifier* for an available ZSERVER. The concept of a ZSERVER identifier is described in Sec. 2.5. These are referenced starting from 0, which is the default server. Specification of which of these to utilise is made in the ξ fields of ZTYPE and ZBOX

[3]As simulation occurs, every object has either a discrete state which remains valid at t, or else a temporal range which contains t.

2.2.2 ztype

Every ZBOX in the system is of a certain type A and subtype n, which refer to a ZTYPE.

- Properties:

  A, n – This ZTYPE defines this type name A and sub-type number $n > 0$.

  C – Containment: list(2) of tuples A, n, ξ which are ZBOX Types and Numbers, to be allowed containment in a ZBOX of this ZTYPE. If **C** is omitted, then nothing is allowed. n absent or 0 indicates that any sub-type may be contained. ξ is a server-consultation mask, absent or '.' if no flags are applicable.

  q – Path-resolution consideration: 0 indicates that ZBOXEN may explicitly consider containment in a ZBOX of this ZTYPE. 1 indicates that ZBOXEN may only consider implicit containment by considering a ZPATH. $q > 1$ is not [yet] supported. The default value is 0 if unspecified.

  V – Speed limit for contained ZBOXEN. [ZSU/s]

  S – Minimum Spacing between contained ZBOXEN. [ZSU]

  L – Capacity [ZSU]; used together with W to fit other ZBOXEN into this one; 'inside size'. This can be zero so as not to restrict capacity (this is only useful in *Bag* case).

  W – Capacity factor (1 has no effect). $W \neq 1$ makes sense only for *Pipe* or *Span* ZBOXEN.

  Progress and velocity within this ZBOX are scaled so that progression time is unchanged. Size and spacing of those contained are not; the capacity is increased by a factor of approximately W.

  ρ – Controls bifurcation of Presence upon entering a ZBOX of this ZTYPE. If this is absent, then no bifurcation occurs. If specified, this indicates a factor in the fraction of the Presence of a ZBOX which will be transferred to this container. $\rho > 0$.

  ρ_0 – Minimum presence: If bifurcation would result in less than this, then no bifurcation is performed. If this is unspecified, then it will default to some small number, something like 1×10^{-5}, but the exact value is not guaranteed.

  x takes values in the range $[0, L-l]$ for the simple $W = 1$ case, so that the maximum progression time is $t_{\text{prog}} = (L-l)/v$. The maximal content is $N_{\max} = (L+S)/(l+S)$.

  Generally, with capacity $L' = WL$,

$$N_{\max} := \frac{L'+S}{l+S}, \qquad t_{\text{prog}} := \frac{x_{\max}}{v_{\text{eff}}}, \qquad x_{\max} := L' - l$$

  so that in the general $W \geq 1$ case,

$$x_{\max} = WL - l, \qquad N_{\max} = \frac{WL+S}{l+S},$$

$$T_{\text{prog}} = \frac{L-l}{v}, \qquad v_{\text{eff}} = \frac{WL-l}{L-l}v.$$

  N – Numerical Capacity. If omitted, capacity is not limited by number.

  χ – a ZLINK template for an *implied* ZLINK between φ and its container (whenever it is contained). μ and ν are ignored; φ and z_φ are implied.

  m – Containment Mode: *Static, Span, Pipe, Shelf, Fifo, Bag, Sink.*

  v – natural speed; this is the speed obtained when no collision or limit is in effect. [ZSU/s]

  \$ – The base cost assigned to traversal of this ZBOX. The default is 0.

  \$$f$ – The cost factor applied when traversing another ZBOX. The default is 1.

  l – Size; used to contain this ZBOX inside another; 'outside size' [ZSU]

  R – Reporting mode: Some combination of 'S', 'C', 'P'. The default is none of these, resulting in a quiet ZBOX. This can also be specified for particular ZBOXEN, which will override this R value. When a ZBOX is given R, it will report on its progress through the system; what exactly is reported is implementation-dependent and described in Section 2.8. R also may contain a decimal integer which specifies the reporting channel; if this is omitted, then the channel defaults to 0. The flag 'd' can be added to increase greatly the detail level.

  Z – A list(2) of tuples A, n, indicating the types of container in which a ZBOX should go to sleep if it cannot exit immediately, awakening only when the container is subject to a new implicit ZLINK, or when other ZBOXEN cease blocking the way. *It is important to understand that this should not affect system behaviour, only simulation performance.*

  ξ – Server-contact specifications for a ZBOX. This can be over-ridden by the ξ specification of a particular ZBOX. See Sec. 2.3.

2.2.3 zbox

A zbox is the fundamental containment and transport unit in the system; all dynamical behaviour is modeled by the movement of zboxen. A zbox can contain other zboxen, can be linked to other zboxen, and can be contained and move within other zboxen (one at a time).

Properties are generally ‘permissive’ if omitted.

- Static properties:

  i – This is a string of external information which will be included in reporting. This information plays no rôle in simulation, but is intended to be useful in passing information to servers without (which might otherwise be tempting) polluting the label.

  A, n – Reference a ztype for this zbox.

  π – Set to 1 to indicate that this zbox is not to be a dynamical object itself, but instead a prototype.

  R – Reporting mode: Over-rides the ztype R value if present.

  S, L, W, N – If present, over-ride those belonging to the ztype (A, n).

  v – If present, over-rides the velocity belonging to the ztype (A, n).

  ρ – Presence $[0, 1]$; if this is specified, then the *zbox* is cloned when certain possibilities arise. If this is not specified, then Presence is not used.

- Dynamical variables: (All but t can be given initial values where sensible)

  z – current zbox in which this one is contained. A zbox can be contained in one zbox, or uncontained.

  x – position of containment within z, trailing edge. [ZSU]

  $\mathbf{Z}$ – list of zboxen contained within this zbox.

  P – Current zpath which φ is following.

  i_P – Current index along zpath (next desired container).

  ξ – Server-contact specifications; over-rides ztype ξ.

Position within a zbox is measured using zbox spatial units [ZSU]; these are arbitrary units, the meaning of which must coincide between intrinsic properties of a zbox, and extrinsic properties of its contained zboxen. In other words, l of a contained zbox is to be compared with L of its container, but l and L for a given single zbox have no relation to each other; l is the “outside size” and L is the “inside size”. Specification of l and L should be in terms of integers, though v and x need not be integer-valued.

A zbox φ will typically enter another zbox λ, progress through λ, and then exit λ, all based on rules, rates and conditions.

The same symbol is used to denote a zbox and also the set of its contents.

- φ contained within λ is written as $\varphi \in \lambda$ or $\lambda \ni \varphi$.
- φ enters λ is written as $\varphi \hookrightarrow \lambda$ or $\lambda \hookleftarrow \varphi$.
- φ exits λ is written as $\lambda \curvearrowright \varphi$ or $\varphi \curvearrowleft \lambda$.

A typical sequence thus consists of the stages $\varphi \notin \lambda$, $\varphi \hookrightarrow \lambda$, $\varphi \in \lambda$, $\lambda \curvearrowright \varphi$, and finally $\lambda \not\ni \varphi$.

Containment modes

Span and *Pipe* are continuous. *Shelf* and *Fifo* are discrete. *Sink* is special. Containment always respects types, controlled by c, C, A, n. *Bag* and *Static* have no notion of position. *Span* and *Pipe* can be scaled in capacity using W.

- *Static* – No movement occurs inside this zbox. It is only used to specify static structure. The containment is fixed. If m is unspecified, the containment type defaults to Static.
- *Span* – zboxen move with continuous position through this zbox without interacting with each other. They are limited in number, and also by total size.[4] The relevant variables are $\{L, W, V, N, S\}$ and $\{v, x, l\}$ for contained zboxen.
- *Pipe* – zboxen move with continuous position through this zbox while retaining their order. They are limited by number and by total size. The relevant variables are $\{L, W, V, N, S\}$ and $\{v, x, l\}$ for contained zboxen.

[4]‘Size’ in this case might be interpreted as something other than spatial extent.

- *Shelf* – ZBOXEN push onto the shelf on one end, and are pushed off of the other end. There is no continuous position, only an ordinal value, but sizes and spacing are respected. The main property is that ZBOXEN may only leave when the shelf is full. When there are no size and spacing restrictions, ‘full’ means N ZBOXEN within. In the general case, ‘full’ is defined to be a state where a ZBOX identical to the ZBOX poised to leave could not first be accommodated. The relevant variables are $\{L, N, S\}$ and $\{l\}$ for contained ZBOXEN.
- *Fifo* – This is like Shelf, but with no requirement to push ZBOXEN out; the cache need not be full. Or, this is like Pipe but with no continuous positions; only ordinal values. The relevant variables are $\{L, N, S\}$ and $\{l\}$ as for the Shelf type.
- *Bag* – This is like Span, but with no notion of position or order. Contained ZBOXEN are limited in number, and also by total size. The relevant variables are $\{N, S\}$ and $\{l\}$ for contained ZBOXEN.
- *Sink* – There is always room to contain a ZBOX, at which time it will disappear. None of the containment variables $\{V, L, N, S\}$ is relevant.
- *Filo* – [A pile of ZBOXEN *yet unimplemented*]

2.2.4 zlink

A ZLINK χ is a directed connection between two ZBOXEN φ and λ; when such a link exists with respect to a ZTYPE τ it is written $\varphi \longrightarrow_\tau \lambda$. When the direction of the ZLINK is unimportant, it is written as $\varphi \relbar\joinrel\relbar_\tau \lambda$.

- Properties: μ_χ, ν_χ – The ZLINK is $\mu \longrightarrow \nu$.
  $\mathbf{A}_\chi$ – Allowance list; a set of triples of the form (A, n, Δ).

Here A is a ZBOX Type, n is a sub-type number, and Δ is a direction indicator. If A is absent, there is no restriction on type or sub-type. If n is absent (zero), the restriction is only by type A. Δ indicates forward, backward, or bidirectional flow. [5] During simulation the configuration of ZLINKS is expected to be time-dependent.

The simplest ZLINK behaviour is the case of two ZBOXEN φ and λ, each of which can contain ZTYPE τ. When $\varphi \rightarrow_\tau \lambda$, ZBOXEN of ZTYPE τ may pass between φ and λ, supposing that entry, exit and containment are allowed, and all ZLINKS are in the correct orientation.

A *zlinked sequence* for a given ZTYPE τ connects two ZBOXEN ϕ and λ by means of 0 or more intermediate ZBOXEN $\{\xi_j\}$. $\phi \relbar\joinrel\relbar_\tau \xi_1 \relbar\joinrel\relbar_\tau \ \ldots \xi_N \relbar\joinrel\relbar_\tau \lambda$. Both ϕ and λ must be able to contain τ, but the intermediate ZBOXEN ξ_j must be unable to contain τ. The notation $\phi \lhook\joinrel\relbar\joinrel\rightharpoonup_\tau \lambda$ will be used both as a statement and as the set of ZBOXEN within the zlinked sequence.

A *zlinked path* is a zlinked sequence with the additional requirement that the ZLINKS be positively oriented. $\phi \longrightarrow_\tau \xi_1 \longrightarrow_\tau \ \ldots \xi_N \longrightarrow_\tau \lambda$. The notation $\phi \lhook\joinrel\longrightarrow_\tau \lambda$ will be used both as a statement and as the set of ZBOXEN within the zlinked path.

When $\varphi \lhook\joinrel\longrightarrow_\tau \lambda$, a ZBOX μ of ZTYPE τ may pass directly (instantaneously) between φ and λ.

Neighbourhoods

This sub-section describes details pertaining to the sets of ZBOXEN reachable from a given base ZBOX. The details here are essential only for a detailed understanding of the underlying code; these details are likely unimportant for a higher-level user of the Zimulator.

Several sets of ZBOXEN are convenient to define, based on a given ZBOX ϕ and ZTYPE τ:

$$\begin{aligned}
\Lambda_\tau(\phi) &:= \{\lambda \mid \phi \lhook\joinrel\relbar\joinrel\rightharpoonup_\tau \lambda\} \cup \{\phi\} \\
\Lambda'_\tau(\phi) &:= \{\lambda \mid \phi \lhook\joinrel\longrightarrow_\tau \lambda\} \\
\tilde{\Lambda}_\tau(\phi) &:= \bigcup_\lambda \phi \lhook\joinrel\relbar\joinrel\rightharpoonup_\tau \lambda \\
\Lambda^{(1)}_\tau(\phi) &:= \{\lambda \mid \phi \relbar\joinrel\relbar_\tau \lambda\}
\end{aligned}$$

Clearly, $\Lambda'_\tau(\phi) \subseteq \Lambda_\tau(\phi) \subseteq \tilde{\Lambda}_\tau(\phi)$. Note that generally $\Lambda^{(1)} \nsubseteq \tilde{\Lambda}$ since $\phi \relbar\joinrel\relbar_\tau \gamma$ does not imply that there exists λ such that $\gamma \ \in \ \phi \lhook\joinrel\relbar\joinrel\rightharpoonup_\tau \lambda$. It might be useful to write naturally for $j > 1$

$$\Lambda^{(j)}_\tau(\phi) {:=} \{\lambda \ \mid \ \phi \relbar\joinrel\relbar_\tau \xi_1 \relbar\joinrel\relbar_\tau \xi_2 \cdots \xi_{j-1} \relbar\joinrel\relbar_\tau \lambda\}$$

and then, of course, there exists sufficiently large j such that $\tilde{\Lambda}_\tau(\phi) \subseteq \Lambda^{(j)}_\tau(\phi)$.

These Λ sets are used within the implementation in order to manage shifting of ZBOXEN. They are maintained and cached for performance. In the source code they are referred to as Λ *neighbourhoods*.

[5] The most usual case is expected to be a single-element $\mathbf{A}$ for a particular desired ZLINK behaviour.

2.2.5 zpath

A ZPATH K specifies a sequence of ZBOXEN to be visited by a ZBOX φ, but they are not always specified directly.

A ZPATH can express two different things, *Plan* and *Intent*.

1. *Plan* – A ZPATH is used to specify a *well-defined* trajectory. ZSTOPS within the sequence describe every visit to be made.
2. *Intent* – A ZPATH can instead be used to indicate a desired list of destinations. ZBOXEN selected by ZSTOPS within the sequence each need to be visited, but some notion of route choice must typically be made between these locations. In the simplest case of an Origin-Destination pair, an Intent ZPATH will contain only two ZSTOPS, both of which will be of the explicit type.

   Given the contect of the system, and a given type (A, n) of ZBOX, an *Intent* ZPATH can be resolved to a *Plan* ZPATH.[6]

A ZBOX can only be sent on one ZPATH at a time. A given ZPATH can be shared among more than one ZBOX.

- Properties:

  A, n – ZTYPE which can use this ZPATH.

  Λ – Ordered list of ZSTOPS.

  m – Mode: Open (default) or Closed. Open indicates a single path with start and end points. Closed indicates a cyclic path, with no endpoints. (If cyclic, the zpath never 'completes' and would [probably] not usually be used in a ZSCHEDULE.)

  t – Type: Plan (default) or Intent.

Each stop on the path determines a ZBOX λ_j. φ is to enter, traverse, and exit λ_j, for $j = 0 \ldots N_K - 1$ in order.

The last stop on a path could of course be a ZBOX of Containment Type *Sink*.

2.2.6 zstop

A ZSTOP is used at a stage of a ZPATH to determine a ZBOX to be visited.

- Properties:
  - Either:

    φ – An explicit ZBOX.

    σ_i, σ_f – Fractional entry and exit positions[7] in the referenced ZBOX φ. These are optional and default to $\sigma_i = 0$, $\sigma_f = 1$. They are ignored if the container type has no notion of position. (*Span* and *Pipe* sport such a notion.)

    In the usual case, a containing ZBOX γ is entered by a moving ZBOX φ when an appropriate ZLINK exists, progression through γ occurs, and then when progression is complete, any ZLINK connected to γ can be used by φ to shift to the next desired container.

    When σ values are supplied, the behaviour is potentially different:

    * Arrival: When arriving in a container ZBOX φ, via a ZLINK χ, the starting position within φ is not $x = 0$, but instead $x = \sigma_i x_{\max} = \sigma_i (LW - l)$.
    * Departure: When arriving at the position $x = \sigma_f x_{\max}$ the contained ZBOX is deemed to have completed its traversal of φ, and may exit according to the usual rules. It cannot travel further within φ.

    The usual ZLINK shifting rules always apply.
  - Or:

    K, i, j – if not an explicit ZBOX, then this specifies any ZBOX travelling on path K, from container ZBOX μ to ZBOX ν. The ZBOX described by the ZSTOP is the one which is travelling on the ZPATH and contained temporarily in μ or ν. μ and ν must be in order on the ZPATH but need not be subsequent. (Actually, since a given ZPATH K may visit μ and ν more than once, indices i and j into the list Λ of K are used.)

[6] This can be considered *low-level* path resolution and always corresponds to finding a shortest path. *High-level* path resolution (for example, consideration of other paths or paths with constraints) can be implemented via the server mechanism, described below.

[7] These are not fully supported in the *Pipe*-containment case; this is documented in the source code.

2.2.7 zschedule

A ZSCHEDULE H is a way of repeating a given path a number of times. At each specified time, a ZBOX is chosen to be assigned to the path. The ZPATH is assigned if a ZBOX is available. The chosen ZBOX must be able to join the path with suitable ZLINKS extant; otherwise the ZPATH will still be assigned, but the ZBOX may become stuck.

- Properties:

  R – Reporting mode, used as in ZDEMAND.

  T_0 – Base time, relative to ZSYSTEM T_0.

  $\mathbf{T}$ – List(1) of deployment times, relative to base time.

  P – A ZPATH to assign at each of the specified times. If P is an Intent path, it will be resolved to a Plan path when appropriate (currently, at deployment).

  If a ZBOX cannot be procured at deployment time, the particular deployment will be cancelled. If the ZBOX selected is busy on a ZPATH, that deployment will be cancelled; the next time in $\mathbf{T}$ will be tried.

  S – a ZSOURCE to determine how to procure ZBOXEN.

  E_T, E_C, E_N, E_L, E_K – Economy Coefficients (exactly as in ZDEMAND)

2.2.8 zdemand

A ZDEMAND is a short-hand way of describing a large number of Origin-Destination pairs, together with times at which to deploy ZBOXEN

- Properties:

  R – Reporting mode. 'S' will report when ZBOXEN are deployed. 'P' will detail the path choice.

  S – a ZSOURCE to determine how to procure ZBOXEN.[8]

  $\mathbf{L}$ – list of zboxen.

  T_0 – base time, relative to system base time.

  $\mathbf{D}$ – A list(4) of tuples (o, d, T, N). The *o*-*d* pair will be converted to an intent ZPATH. The tuples can also be (o, d, T, N, v) or $(o, d, T, N, \mathbf{v})$, with ZBOX velocity specified. T is measured with respect to the base time. [9] $\mathbf{D}$ will be sorted in time before being used; resulting labels in the system will reflect this.

  E_T, E_C, E_N, E_L, E_K – Economy Coefficients – Path Economy E is dependent linearly on time, cost, number of ZBOXEN visited, number of ZLINKS traversed, and number of container ZBOXEN on ZPATHS utilised. These are the coefficients. If omitted, default values of 0.05, 1.0, 1.0, 0.0, 12.0.

Here, o and d are integer references $(0\dots)$ into $\mathbf{L}$ to specify the origin and destination ZBOXEN. T is a time to initialise a ZBOX with this travel intent at o. As many as N ZBOXEN are initialised from o at time T (relative to T_0), depending on availability.

2.2.9 zsource

A ZSOURCE is used in objects like ZDEMAND or ZSCHEDULE to choose a ZBOX for deployment.

- Properties: φ – A ZBOX.

  m – Mode: One, From.

  - One: φ is a *particular* ZBOX to send on the ZPATH.
  - From: φ is a particular ZBOX *from* which to procure ZBOXEN of the proper type (A, n) for use on the ZPATH P.

  o – Origin: Container, Teleport

  v_μ, v_σ – Optional specification of velocity distribution. Whenever this ZSOURCE is used to procure a ZBOX, a velocity is chosen from a log-normal distribution with mean v_μ and standard deviation v_σ. That is, Lognormal(μ, σ^2) with

$$\mu = \log v_\mu - \frac{1}{2} \log\big(\frac{v_\sigma^2}{v_\mu^2} + 1\big), \quad \sigma^2 = \log\big(\frac{v_\sigma^2}{v_\mu^2} + 1\big).$$

[8]Do not supply a single-ZBOX ZSOURCE unless it is a prototype.

[9]The form with $\mathbf{v}$ (which is expressed as v, v, v, v, v and expresses natural velocity in subsequent zpath containers) is likely useful only for testing, and should not be depended upon.

This is *overridden* by explicit specification of velocities within a ZDEMAND or ZSCHEDULE.

When either of the above modes selects a prototype ZBOX, a *copy* is procured. If the origin type is Container, the copy is fabricated inside the prototype's container, so there should be room for it. If the origin type is instead Teleport, the copy will be placed inside the first container on the relevant ZPATH.

Note that a prototype ZBOX sbould be empty; possibly non-empty prototype ZBOXEN can be supported later.

2.3 Server consultation

The core simulation involves a small number of types of basic objects which interact according to a few basic rules. To a large extent, the main philosophy here is that simple properties can produce complex behaviour.

Nevertheless, not everything can be encoded purely with ZBOXEN, so consultation with external modules is implemented. Sophisticated models for route choice or market dynamics need not be part of the core.

Core functions include:

- Basic ZBOX movement and rules using containment and ZLINKS
- Basic ZBOX deployment through ZDEMAND and ZSCHEDULE
- Shortest-path resolution at microscopic level (i.e. every ZBOX)

Communication involves reference to various ZOBJECTS; this is always by label.

2.3.1 Event specification

Within ZBOX and ZTYPE there can appear a server-consultation specification ξ. Consultation is always 'with respect to' a ZBOX φ. A server is consulted when: (and)

- The ξ specification of φ includes some event indicator F
- The ξ mask in the containment-allowance C of the container of φ includes F (C does not mask modifier flags, as described below, only events).
- The condition F becomes true for φ.

Each field within ξ can contain one of the following event indicators for a ZBOX φ of ZTYPE χ:

- C – 'containment': $\varphi \hookrightarrow \lambda$, just *after* entry into λ.
- E – 'exit': $\lambda \curvearrowright \varphi$ just *before* exit from λ.

Modifier flags may also be present:

- f – 'first to arrive': Adds to C the condition that λ is empty before $\varphi \hookrightarrow \lambda$.
- l – 'last to leave': Adds to E the condition that λ is empty after $\lambda \curvearrowright \varphi$.
- v – 'no vacancy': Adds to C the condition that another ZBOX similar to φ would be unable subsequently to enter λ, due to capacity alone (i.e. not specific positional configuration). In other words, the condition is that φ 'fills' the container.
- p – 'path': Adds the condition that the container must correspond to the last ZSTOP on φ's current ZPATH.

A server number may also be present. This is an index into the Ξ list of ZSYSTEM, and defaults to 0 if omitted. As an example, 'Cf12,Ep2' would indicate consultation with server #12 on first entry, and consultation with server #2 on exit at end of `zpath`.

2.3.2 Request

- φ – the ZBOX involved (by label).
- ... – various information about φ: φ's container, whom φ contains, how many contained together, φ's ZPATH, etc.

A server should always be designed so that if information is missing, the request is still accepted and the response is something valid.

2.3.3 Response

The response to *any* server request is some amount of ZSYNTAX language as described in Sec. 2.4, with the following embellishment. The ZOBJECTS specified (always by label) each have an associated insertion mode:

- *Fresh* – The ZOBJECT is inserted into the system.
- *Modification* – The specified ZOBJECT is modified.
- *Fresh-or-Modification* – If the label is in use, the ZOBJECT is modified; otherwise it is fresh. (Reference to the ZSYSTEM is always of this type.)
- **Deletion* – The specified ZOBJECT is removed from the system.
- **Copy-and-Modify* – The specified ZOBJECT is duplicated and then modified.

*These two modes are not yet implemented in the code.

Not all request types are applicable to all ZOBJECTS. The specific ZSYNTAX for specification of these insertion modes is documented below in Sec. 2.4.2

Finally, despite the connotation of remoteness and slowness associated with a 'server', there is no necessity for either of these undesirable properties. Servers may be implemented as local code by a user of the library (and implementor of the `zio` interface). These considerations are discussed in Chapter 3 in Sec. 3.2.

2.4 Input language

The input language *zsyntax* is intended to be universal, but of course other input languages could be fashioned which either convert to this format or can be compiled separately to object-file format. In this section is described the zsyntax *source* format, which is also the format for server response as described in Sec. 2.3.

Inputs describe the dynamical system Ψ to be simulated. There are also inputs (but fewer, since these are separate from the system) which describe *how* to simulate Ψ, how to output information about Ψ and elements therein, and how to manage simulation (starting, stopping, etc.).

All input is via utf-8 text files.[10] If a `#` appears anywhere on a line, the remainder of the line is treated as a comment.

ZSYNTAX input can be split and concatenated among files arbitrarily, provided this is done between top-level ZOBJECTS.

ZSYNTAX input files could end with a `.zim` suffix. When editing by hand in emacs, `awk-mode` seems to work well for the syntax; maybe `cc-mode` is also usable.[11]

Properties which are denoted by a Greek letter can be spelled using Roman ones, but the converse is not supported.

Time specifications are always one of `h:m:s`, `h:m` or `s`. Each of `h`, `m` and `s` is a positive decimal number with no bound. For example, 'noon' is any of `43200` $\equiv$ `12:00` $\equiv$ `12:00:00` $\equiv$ `00:720` $\equiv$ `06:360:0`.

The input language here is probably slightly more general that needed, so that later expansion or hierarchical structure are not awkward.

Generally, an object definition is expressed inside `[…]`, a referenced object inside `<…>`, and a list of objects by `{…}`. `[…]` and `<…>` are interchangeable anywhere an object is required. In any of these constructions, entries are whitespace-separated. The reserved characters `{ } [ ] < > =` and whitespace are solely for defining structure, and may not be used in labels or values. All non-alphanumeric symbols with unicode $<$ \$100 (Punctuation) are reserved for later application, save `_` which can be used, and `- .` which can be used provided they are not the initial character.[12]

Whitespace-separated entries are placed between the `[…]`. The first such entry is always the object type, such as `zbox`. An optional unique[13] label can appear as the next field, with no delimiters.

Labels serve two purposes. Firstly, they can be references within the input using `<…>`. Secondly, they can be used in output streams to identify objects. For debugging purposes, it is advisable always to provide a label. In labels, a `/` is added to denote generated (as opposed to specified) objects, as noted below.

[10] `http://utf8everywhere.org`
[11] For this reason, some of the `.zim` files included in examples begin with `#!awk` as a text-editor hint.
[12] The user is free to use cat emoticons, ice-cream cones and smiley faces from unicode pages 499-502, for example.
[13] That is, the label is in a global name space.

Following are a number of assignments which are all of the form `property=value` to set parameters as described in Section 2.2.[14] The property can be a symbol, like `z` or φ (without decoration, since it is a plain text file), or can be the name of a property, like `zbox`.[15]

An argument which is not an object or reference (for example, values in a list, or something after '=' which is just a number or text) does not sport any delimiting notation. To omit an element in an ordered list, the placeholder '`.`' is used.

Some examples of what this notation looks like are:

```
[zbox Stn_AMK A=Station C={SubStation 1} ]
[zbox SubStn_AMK_NB A=SubStation C={Train 1 Platform 1} m=Fifo L=1.0 z=<Stn_AMK> ]
```

`{…}` contains a list, possibly of objects, possibly of simpler things. A list of four ZBOXEN might be expressed as `{ [zbox tr1 …] [zbox tr2 …] <tr3> <tr4> …}`
which defines the objects `tr1` and `tr2` while referencing `tr3` and `tr4`. A ZLINK could be expressed
`[zlink TrainDoor1` μ`=<tr1>` ν`=<tr2> A={Pax . b}]`.

Lists generally have no element delimiters. Without such separators, there is special syntax for structured 'pairs' or tuples, as used in e.g. ZPATHS. The list `{…}` simply contains the tuple items in order. A list of three pairs is expressed like `{`a_1 b_1 a_2 b_2 a_3 b_3 `}` where a_j and b_j are the objects (or references). Thus, the list notation `{…}` is required even for a single tuple.

Lists which are *not* lists of objects (i.e. lists which contain only simple fields like numbers or flags) have a default element length, which is supplied in paranthesis in the list description, but can in some cases be given optional parameters. The default element length is used if no delimiters are present (e. g. the ZTYPE C list defaults to (A, n) pairs). Optionally, a list can be expressed so that various lengths of tuples can be expressed easily. This is implemented as the *sed*-inspired form `{/ a b c / a b c d / a b c / a b c d e /}` for a list of four elements which happen to be a 3-, 4-, 3-, and 5-tuple. Lists like ZTYPE C and ZDEMAND D require this feature.

2.4.1 Specific input notation

- ZSYSTEM
  `[zsystem label T_0=... t_0=... t_1=... ]`
  The objects in the system are not specified within this; all the objects simply appear elsewhere in the input streams.
  It is usual to specify T_0 as 0 or 00:00 and then, for example, specify `t_0=08:00` and `t_1=18:30`.
- ZTYPE
  `[ztype A=... n=... C=... S=... m=... L=... W=...` σ`=... V=... l=... c=... v=...` π`=... R=...` ρ`=...` ρ`_0=...` ξ`=... n_`ξ`=... ]`
  Synonyms: `[ sigma=... pi=... rho=... rho_0=... xi=... n_xi=... ]`
  R is a combination of S, C and P for 'Self', 'Container' and 'Path'.
  The server-contact specifications ξ are written as a comma-separated list. An example would be ξ`=Cf1,Ep2` .
- ZBOX
  `[zbox label i=... R=... A=... n=... s=... z=... x=... Z=... P=... i_P=... S=... L=... W=... N=... l=...` π`=... v=... ]`
  S, L, W, N and l are implemented as integers, while V, x and v are floating-point. The use of integers is so that used space inside a ZBOX can be accumulated without error or recalculation.
  z or Z are specified; when $z_\varphi = \lambda$, $\varphi \in Z_\lambda$ is implied, and vice versa. It is invalid to specify that a given ZBOX is contained (directly) in more than one other.
  π should be 0 or 1 and is taken as 0 if omitted.
  i_P only makes sense if P is specified; it defaults to zero.
  Synonyms: `[zb sigma=... pi=... ]`
- ZLINK
  `[zlink label` μ`=...` ν`=... A=... ]`
  Synonyms: `[zl mu=... nu=... ]`
- ZPATH
  `[zpath label A=... n=...` Λ`=... m=... t=... ]`
  Synonyms: `[zp Lambda=... ]`

[14] Maybe also support `property:value`? Not compatible with insertion-mode labels, though.
[15] These will become well-defined later.

- ZSTOP
  `[zstop label` φ`=... K=...` σ`_i=...` σ`_f=... i=... j=... ]`
  Synonyms: `[zs phi=... sigma_i=... sigma_f=... ]`
- ZSOURCE
  `[zsource label` ϕ`=... m=... o=... v_`μ`=... v_`σ`=... ]`
  Synonyms: `[ phi=... v_mu=... v_sigma=... ]`
- ZSCHEDULE
  `[zschedule label R=... T_0=... T=... P=... S=... E_T=... E_C=... E_N=... E_L=... E_K=... P_r=... ]`
  Synonyms: `[zsch ]`
- ZDEMAND
  `[zdemand label R=... R_n=... S=... L=... T_0=... D=... Reference=... Cache=... E_T=... E_C=... E_N=... E_L=... E_K=... P_r=... ]`
  If a prototype has a label, the deployed ZBOXEN will receive the same label with `/` and a natural number appended.
  Here `Reference` is a string containing `%s`, which will be replaced with the entry in `L` to get the actual ZBOX Label. Thus, `Reference=%s` will just use the `L` labels directly and is the default.
  `Cache` is either `true` or `false` and indicates whether resolved ZPATHS are to be cached.

2.4.2 Server consultation

The event mask ξ is specified as a combination of the event letters (uppercase) `C E` . . . . When appearing in a list, without assignment, the value of ξ is again just a combination of those letters, with `.` as a placeholder if no flags are specified.

In server responses, the insertion mode is specified via a label prefix, as described in the table.

Function	Label syntax	Description
Fresh		Creates an unlabeled zobject.
Fresh-or-Modification	`Label`	If label exists, like `M:` otherwise like `F:`.
Fresh	`F:Label`	If label exists, request is ignored.
Modification	`M:Label`	If label does not exist, request is ignored.
*Deletion	`D:Label`	If label does not exist, request is ignored.
*Copy-and-Modify	`C:Label:NewLabel`	If label does not exist, ignored.

* These ones are not yet implemented. Also, 'ignoring' as described is not implemented; an error will be flagged. The user should see the source code for details (Chapter 3).

2.5 System Integration

System integration is implementation-dependent. In the present implementation, two Interfaces are defined:

- `zim` Interface. The Zimulator implements this, so that it may be called for simulation.
- `zio` Interface. The Zimulator requires an external implementation of this to communicate with the wider system, and will use it for all IO operations. `zio` contains functions for Verbosity, Reporting, ZSERVER communication, and state storage and retrieval.

As mentioned in Sec. 2.2.1, within the ZSYSTEM specification is a list of ZSERVERS, each with an *identifier*. This identifier is simply a string used to identify the ZSERVER; it is passed to the server-consultation method in the `zio` interface. The identifier could be a name like `server5`, URI like `http://zserver.ibm.com/somthing/else`, or anything at all. The meaning is *not defined* within the Zimulator specification.

2.5.1 Command-line invocation

The Zimulator core supports only the calls defined in the `zim` interface. A command-line interface is supplied as a separate tool and is described in Chapter 4.

2.6 Object files

The object-file format is written by and read by only the core simulation; object files are not expected to be exposed to servers or other tools.

Typical object code stores in compact (i.e. byte-level, not text) format the state of the whole system, so that:

- Simulation from t_A until t_C results in system state S_C.
- Simulation from t_A until t_B, storing the state S_B in an object file, and then subsequently loading the state S_B from the file, and simulating until time t_C results in the same system state S_C.

Minor deviation from this exact 'associativity' is allowed (at least on the order of 'double-precision' calculation).

As it stands, the precise object-file format is *implementation-dependent*; documentation should be available in the source-code for that implementation. It is not intended to store object files such that they are then used elsewhere.

2.7 Core dynamics

Zboxen interact via simple rules; 'time' is always present, so the state of the system evolves in time.

2.7.1 Time evolution of system

In some simulations time is simply discretised and the system is stepped by Δt repeatedly, noting various interactions among elements at each time step.

The implementation could operate in one of the following ways:

- Time steps This is the trivial $t \to t + \delta t$ at each time step, with δt chosen so that the problem at hand is sufficiently approximated.
- Time intervals

  This amounts to choosing Δt to be the interval until the next 'transition' in the system, and extrapolating all behaviour within this interval. Then, $t \to t + \Delta t$ where Δt is not necessarily small. Only objects in the system which are involved in the transition are processed.
- Discrete – Changes of state (which depend on existence of ZLINKS and configurations of ZBOXEN.) These changes are instantaneous.
- Continuous – Movement within a container. This is not parametrised as discrete steps, but as linear Ranges $x_1, t_1 \to x_2, t_2$.

The Reference Implementation operates in this way. As a detail, it is noted that at any given instant in time,

- More than one container may be anticipated to satisfy an entry condition.
- More than one ZBOX may be competing for entry into a given container.
- In general, n ZBOXEN may be competing for entry into m container ZBOXEN.

2.8 Reporting outputs

Reporting is specific and precise, whereas *verbose* output is for human use in debugging and tuning a system. Reporting must be unambiguous, expandable, and easy to parse. Therefore every report line has the following structure:

`R: objtype var=val var=val ...`

Some variables will always be present, so the minimum output line is of the form:

`R: objtype ztype=... t=...`

where t is the system time. `objtype` is one of `zbox`, `zpath`, `zsystem`, etc.).

Various objects within the system can report on their progress and state. 'S' indicates generic output and valid whereved R can be specified. The most important example is a ZBOX.

To the `ztype` variable will be assigned `A,A',n` where A is the ZTYPE name, and A' is a unique positive integer assigned to that name at runtime. Other variables reported include:

'S' mode: `x0 t0 x1 t1 l z.L o state label z.label`

'C' mode: `z.n Z.n`

'P' mode: `K`

Here o is the containment position (i.e. how many zboxen are in front), and n is the total containment number (i.e. how many are contained in total). *state* is the ZBOX state, which will be M, D or S. The S state does not indicate movement, but is a discrete element of the state machine. In the the M or D states, `x0 t0 x1 t1` will be present, describing the current free movement between (x_0, t_0) and (x_1, t_1). M indicates a general movement within a container; D indicates the last such movement before shifting out of the present container is allowed. This is enough information for interpretation of these report variables in the context of model simulation; for internal details of the ZBOX state machine the use is of course referred to the source code, outlined in Chapter 3.

To streamline parsing of reported information, any object may issue its reporting lines on a particular *channel*, by including a decimal number within the R specification (this is mentioned above in describing the ZBOX object). If the channel number is omitted, it defaults to 0. The reporting channel is specified when calling the user's implementation of the `sendReportLine()` function within the `zio` interface.

The flag 'd' can be added to increase greatly the detail level; this seature is primarily useful for debugging the specification of a system.

As an example, suppose a ZBOX is specified with `R=SC10d` . Then, this particular ZBOX will issue very detailed report lines on reporting channel 10, containing information about its own state and also its container. Suppose a ZTYPE includes the specification `R=S8`; then all ZBOXEN of this type which do not override R will report their basic movements and state information on channel 8.

Example output lines will be shown in Chapter 7 in the context of the Madrid Metro model.

Chapter 3

Zimulator code

Section 3.1 is a high-level description of the source code for the Zimulator. This section will be of little interest to the reader only wishing to make use of the simulator without modification or improvement.

Interfacing with the code (that is, making use of the Zimulator as a library) is documented below in Sec. 3.2. A complete example of interfacing with the Zimulator library is given in Sec. 4 in the form of the command-line interface which is used to run all the provided simulation examples.

3.1 Source files and structure

Here are described in summary the Java source files which comprise the simulation core. The following list is intended as a very high-level tour of the various source-code files; of course an in-depth understanding of the source code can only be obtained by examination of the code and associated comments, beginning with the `zio` and `zim` interfaces and the `Zimulation` class which is the public entry point.

- Public interfaces: (Discussed below in Sec. 3.2)
  `zim.java` – implemented by the Zimulator
  `zio.java` – implemented by the user (e.g. CLI tool) to communicate with the Zimulator
- Object file reading and writing:
  `CompiledFileIO.java` – routines for reading and writing the system state
  `CompiledFileRW.java` – interfaces for low-level IO of objects and reference resolution
  `PrimitiveDataIO.java` – low-level routines for reading and writing basic types
- Communication with external servers:
  `ExternalServer.java` – wrappers for communicating with servers
  `ServerFlags.java` – routines for manipulation of the ξ server flags specified
  `ServerRequest.java` – serialization of server requests
- Source-file zsyntax loading:
  `ZsyntaxInputParser.java` – main parser for zsyntax input
  `LabelReferencing.java` – resolution of labels specified in zsyntax
  `LowLevelFileReading.java` – low-level reading of zsyntax
  `ZtypeReferencing.java` – resolution of (A, n) for specified ZTYPES
  `ConnectivityResolution.java` – consistency of ZLINKS, containment and ZPATHS `ConstantValues.java`
- System and lists:
  `Zimulation.java` – main simulation class; top-level control; `zim` interface implementation
  `zsystem.java` – holder for the system lists and other simulation parameters
  `SystemLists.java` – the four system lists ($[0][S][T][Z]$; documented in the code)
  `Containment.java` – management of containment of ZBOXEN
- Verbosity and Reporting:
  `Reporter.java` – system output Reporting. `Verbose.java` – verbose output `HTML_output.java` – output (debugging) of HTML-table representation of system ZBOX configuration
  `PS_output.java` – output (debugging) of Postscript representation of system ZBOX configuration
  `BoxCounter.java` – counting of ZBOXEN and production of live terminal output
  `DoReportFlags.java` – handling of various reporting flags
  `ReportFlags.java` – storage of reporting flags
  `TimeRoutines.java` – time syntax (e.g. h:m:d)

`CommandLine.java` – something to execute when Zimulator is invoked from command line (This only prints a message; it is *not* the CLI described in Chapter 4.)

- Object behaviour:
  `zobject.java`, `zbox.java`, `zdemand.java`, `zpath.java`, `zschedule.java`, `zsource.java`, `zstop.java`, `ztype.java`, `zlink.java` – behaviour of the various ZOBJECTS described in Chapter 2
  `Presence.java` – fractional presence of ZBOXEN; not utilised in metro-system simulation.[1]
  `Accommodation.java` – handling questions of capacity when ZBOXEN shift to new containers
- Paths and Links:
  `Economics.java` – keeping track of expense of various portions of a ZPATH
  `ZboxNetwork.java` – implementation of a Network interface from the auxiliary library, used to assign ZPATHS
  `PathResolution.java` – filling in missing ZBOXEN along ZPATHS
  `Route.java` – keeping track of logistics of ZPATHS
  `Neighbourhoods.java` – storage of various Λ neighbourhoods (described in Sec. 2.2.4)
  `NeighbourhoodCalculation.java` – calculation of the Λ neighbourhoods

3.2 `zim` and `zio` interfaces

The Zimulator implements `zim`*; the user implements* `zio`*.* This is explained in the present section.
The `zio` interface is very simple:

```
public interface zim
{
    public void adjustFlags(String[] VarValPairs);
    public void adjustParameters(String[] VarValPairs);
    public int run();
}
```

The Zimulator is intended to be used by performing the following steps. A concrete example of doing so can be found in the CLI described in Chapter 4.

- Instantiate a class which implements the `Zimulator.zio` interface (described just below).
  This class describes various input and output streams.
- Instantiate a Zimulation class; `MyZim = new Zimulation(zio MyZio)`. This will contain default settings.
- Optionally call `MyZim.adjustFlags()` and `MyZim.adjustParameters()` to control the simulation. Arguments to these methods contain `variable=value` pairs, as used with the CLI tool, described in Chapter 4.
- Call `MyZim.run()` to perform the simulation.
  This initiates actual simulation. It does not return until the simulation has finished. During the simulation, of course, servers might be consulted, depending on how the system has been prepared.
- The output from the simulation will be sent, *as it is simulated*, in accordance with that specified in `zio`.

As mentioned, all of this requires the user to implement the five functionalities (six methods) of the `zio` interface:

```
public interface zio
{
  // (1) Reading inputs:
  Iterator<Iterator<String>> getZsyntaxInputs();
  // (2) Consulting servers:
  Iterator<String> makeZserverRequest(String ServerId, Iterator<String> Request);
  // (3) State storage:
  DataOutputStream saveZobjFile();
  DataInputStream loadZobjFile();
  // (4) Reporting:
  void sendReportLine(int ReportChannel,String ReportLine);
  // (5) Verbosity:
  void sendVerboseLine(String VerboseLine);
}
```

This interface is designed in such a way as to provide complete versatility for the user. For a concrete implementation of the `zio` interface, the reader is referred to Chapter 4.

[1]Thus, not well tested.

- `getZsyntaxInputs()`
  By means of this iterator, the Zimulator reads in initial input zsyntax. The form of 'iterator over iterators of strings' is meant to make it trivial to feed the Zimulator a number of source files.
- `makeZserverRequest()`
  The `ServerIdentifier` is merely the identifier text string provided in the ZSYSTEM Ξ list. The reply from such a server should be zsyntax text.
- `saveZobjFile()` and `loadZobjFile()`
  These methods are called to store and retrieve the system state using an internal object-file format. Of course, the user might elect that this be done via files or in some other way.
- `sendReportLine()`
  All reporting output from the simulation is sent through this function; a line at a time is sent, with a channel specified. The channel number used is that specified in the R field of the object making the report.
- `sendVerboseLine()`
  All verbosity output from the simulation is sent via this function, one line at a time.

It is guaranteed that all provided iterators will be used immediately and completely. DataInputStream and DataOutputStream for object files are also accessed "all at once" and subsequently closed.

The difference between *verbosity* and *reporting* has been described in Sec. 2.8.

It is noted that a 'server' is only external from the point of view of the core Zimulator. It may indeed be implemented as a 'slow' process which replies to an http request on a distant machine, or it may be implemented as 'fast' local code. Therefore, there is little intrinsic performance penalty in delegating various non-core functionality to zservers. The only requirement is that whatever server is implemented must emit zsyntax replies for ingestion by the Zimulator core.

Chapter 4

Command-line interface

In this section is described a Command-Line Interface (CLI) for the Zimulator. This CLI will be used in later sections in order to run various example simulations, in addition to furnishing a complete example implementation of the `zio` interface.

4.1 CLI source code

The source-files are located in the `ZimCLI` directory:

- `CommandLine.java` – class with `main()` for command-line use; constructs `Zimulation` class
- `Help.java` – self-documentation for command-line
- `ComLineIO.java` – implementation of `zio` described in detail just below
- `FileOrHttpLineIterator.java` – 'translation' from a filename or http address to a string iterator

Within `ComLineIO.java` the five `zio` functionalities are implemented as follows:

- (1) Reading inputs:
  Filenames or http sources can be specified on the command-line; these are converted to the required string iterators.
- (2) Consulting servers:
  The Server Identifiers in the specified ZSYSTEM are simply interpreted as http addresses. This is merely a choice made in the CLI; the identifiers could in fact be any strings at all.
- (3) State storage:
  The streams for loading and saving of state are connected to files; the filenames are specified on the command line.
- (4) Reporting:
  A filename is specified on the command line.
- (5) Verbosity:
  This is simply sent to `stdout`, with some attributes controlled by command-line switches.

4.2 CLI usage

Typical example usage for the examples which follow in this document is

```
$ java -jar Zimulator.jar zl=45 z=30 I=ZsyntaxSourceFile.zim R=output.zo
```

This can be understood by running the CLI with no arguments, invoking the help feature and producing the terminal output:

```
--------------------------------------------
  Zimulator -- Version:4.000  Built:2018-11-26.15:02
--------------------------------------------
Function:
  Simulate the dynamical system described by inputs; see documentation.
--------------------------------------------
Java library usage:
See Source/zim.java and Source/zio.java
--------------------------------------------
Command-line Usage:
$ java -jar Z.jar [options] [parameters]

[options] should be specified separately. User options:
  -h -help       --  Print the present output and exit; no simulation.
  -ns            --  Inhibit simulation. Useful with html or dump, or with 'o' for compilation.
  -v             --  Print verbose output to terminal while simulating (for human).
  -pauses        --  Sleep for 1s on every zbox shift. ('useful' with -v)
  -razo          --  After loading, Report all zboxen once.
  -dump          --  Dump system configuration in human-readable form to stdout,
                     immediately after reading inputs.

[parameters] are always of the form 'param=value':
  t_1   -- Simulation stopping time. Overrides that of the zsystem.
  t_0   -- Simulation starting time. Overrides that of the zsystem. Care should be exercised.
  R     -- Output filename for reporting; defaults to /dev/stdout. (for Machine)
  z     -- Simulated time interval at which to issue terminal reports (for Human)
           on stdout; defaults to 0 := no reporting.
  zl    -- When z is nonzero, this is how many lines to display on terminal.
  i     -- Input Object File. This can be specified only once,
           and will be loaded before any source files.
  o     -- Output Object File. System state will be written at end.
  html  -- Output html file. System description will be written at end.
  PS    -- Output simple PS file of all zboxen on one page; written at end.
  I     -- Input Source (zsyntax). If this is specified more than once, then
           the sources are loaded in order. These can be files or URLs (e.g. http://).
  v     -- Only effective without -v switch. Activates verbosity only for zboxen in
           the provided comma-separated list; e.g. v=Label1,Label2,...
 All files are read or written sequentially and completely; can therefore be pipes or devices.

[options] for debugging and development:
  -noneighcache -- Disables the neighbourhood Λ caches.
  -nozpathcache -- Disables resolved-zpath caching, overriding input specification.
  -nozdemand    -- Disables state processing of zdemand objects.
  -nosleeping   -- Prevents any zbox from going to sleep.
--------------------------------------------
```

In the following Chapter 5 the CLI will be used to run some simple example simulations.

Chapter 5

Toy examples

5.1 Four pipes and a span

An example is constructed with five "rooms" connected by ZLINKS. Four of these room ZBOXEN are designated *Pipe* type containment, while one is *Span* type. "Balls" will be send on ZPATHS which traverse the rooms and end in a ZBOX of type *Sink*.

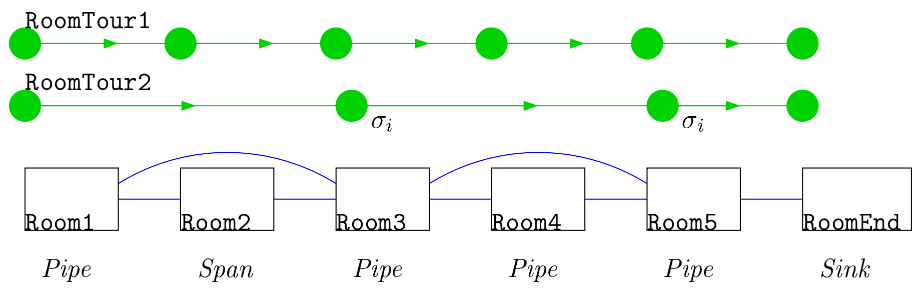

Figure 5.1: Five rooms (black), of *Pipe* and *Span* types: ZLINKS (blue) allow the two ZPATHS (green). The `RoomTour2` ZPATH enters `Room3` and `Room5` at non-zero initial position.

The full zsyntax input is listed below. First global parameters in the ZSYSTEM are defined:

```
[zsystem PipeTest T_0 = 0  t_0 = 00:00  t_1 = 12:30  Δt = 30 R=S ]
```

The types of rooms are specified:

```
[ztype A=piperoom n=1 m=Pipe C={Ball 1} L=90 ]
[ztype A=piperoomslow n=1 m=Pipe C={Ball 1} L=90 V=3.1 ]
[ztype A=spanroom n=1 m=Span C={Ball 1} L=90 V=3.1 ]
[ztype A=exitsink n=1 m=Sink C={Ball 1}]
[ztype A=Ball n=1 l=20 v=5 R=SCPd]
```

Balls will be inserted into the system; for this a prototype ZBOX is required:

```
[zbox BallProto A=Ball n=1 π=1 ]
[zsource BallSource m=One o=Teleport ϕ=<BallProto>]
```

Five rooms can be represented by ZBOXEN and connected with ZLINKS:

```
[zbox Room1 A=piperoom n=1 ]
[zlink μ=<Room1> ν=<Room2> A={Ball . 1}]
[zlink μ=<Room1> ν=<Room3> A={Ball . 1}] # zlink also from 1-3.
[zbox Room2 A=spanroom n=1 ]
[zlink μ=<Room2> ν=<Room3> A={Ball . 1}]
[zbox Room3 A=piperoom n=1 ]
[zlink μ=<Room3> ν=<Room4> A={Ball . 1}]
[zlink μ=<Room3> ν=<Room5> A={Ball . 1}] # zlink also from 3-5.
[zbox Room4 A=piperoomslow n=1 ]
[zlink μ=<Room4> ν=<Room5> A={Ball . 1}]
[zbox Room5 A=piperoom n=1 ]
[zlink μ=<Room5> ν=<RoomEnd> A={Ball . 1}]
[zbox RoomEnd A=exitsink n=1 ]
```

One ZPATH visits all rooms in order; another visits the same rooms, but shows the use of explicit entry and exit points:

```
[zpath RoomTour1 A=Ball n=1 m=Open
 Λ={ # 'Λ' is a list of zstops.
     [zstop ϕ=<Room1>]
     [zstop ϕ=<Room2>]
     [zstop ϕ=<Room3>]
     [zstop ϕ=<Room4>]
     [zstop ϕ=<Room5>]
     [zstop ϕ=<RoomEnd>]
 } ]
[zpath RoomTour1a A=Ball n=1 m=Open
 Λ={ # 'Λ' is a list of zstops.
     [zstop ϕ=<Room1>]
     [zstop ϕ=<Room2> σ_i=0.1 σ_f=0.9 ] # specific entry and exit points
     [zstop ϕ=<Room3> σ_i=0.05 σ_f=0.9 ]
     [zstop ϕ=<Room4> σ_i=0.05]
     [zstop ϕ=<Room5>]
     [zstop ϕ=<RoomEnd>]
 } ]
```

Another ZPATH skips rooms 2 and 4; this is possible due to the ZLINK configuration:

```
[zpath RoomTour2 A=Ball n=1 m=Open Λ={ [zstop ϕ=<Room1>] [zstop ϕ=<Room3> σ_i=0.2]
    [zstop ϕ=<Room5> σ_i=0.05] [zstop ϕ=<RoomEnd>] } ]
```

A ZSCHEDULE can be used to deploy some balls on these trajectories:

```
[zschedule SendBalls
 T_0=00:00:00              # A base time
 S=<BallSource>            # A zsource whence to procure zboxen
 P=<RoomTour1a>            # A path to deploy them on
 T={1 10 20 30 40 45 50 } # A list of deployment times
]
```

Three extra balls are inserted explicitly, to cause some interference:

```
[zbox Ball.Fast A=Ball n=1  z=<Room3> x=45 P=<RoomTour1> i_P=2 v=10]
[zbox Ball.Slow A=Ball n=1
 z=<Room1> x=65         # Starts in Room1 at position x.
 P=<RoomTour2> i_P=0    # Take this path; start at this index.
 v=0.9                  # Much slower than a normal Ball.
]
[zbox Ball.Stuck A=Ball n=1  z=<Room4>  P=<RoomTour1> i_P=3 x=67.5 v=0.55]
```

5.1.1 Execution

The system which has just been described can be simulated by running the CLI,
`$ java -jar ZimCLI.jar I=Testing_Pipe_Zboxen.zim R=_Testing_Pipe_Zboxen.zo`
which is part of the provided script mentioned below. This will write all reporting output to the file `_Testing_Pipe_Zboxen.zo`:

```
R: zsystem T_0=0.0 t=0.0 %=0.000
R: zbox ztype=Ball,2,1 t=0.0 R=- state=D Z.n=0 label=Ball.Fast z.label=Room3
   z.L=90 z.W=1 t0=0.0 x0=45.0 t1=2.5 x1=70.0 δt=2.5 l=20 L=0 z.n=1 K=RoomTour1
   i_P=2 D=""
R: zbox ztype=Ball,2,1 t=0.0 R=- state=D Z.n=0 label=Ball.Slow z.label=Room1
   z.L=90 z.W=1 t0=0.0 x0=65.0 t1=5.555555555555555 x1=70.0 δt=5.555555555555555
   l=20 L=0 z.n=1 K=RoomTour2 i_P=0 D=""
R: zbox ztype=Ball,2,1 t=0.0 R=- state=D Z.n=0 label=Ball.Stuck z.label=Room4
   z.L=90 z.W=1 t0=0.0 x0=67.5 t1=4.545454545454545 x1=70.0 δt=4.545454545454545
   l=20 L=0 z.n=1 K=RoomTour1 i_P=3 D=""

   . . . (about 250 lines)

R: zsystem T_0=0.0 t=215.19739328771587 %=0.478
R: zbox ztype=Ball,2,1 t=219.59739328771587 R=- state=M Z.n=0 label=BallProto/7
   z.label=RoomEnd z.L=0 z.W=1 t0=219.59739328771587 x0=0.0 t1=219.59739328771587
   x1=0.0 δt=0.0 l=20 L=0 z.n=1 K=RoomTour1a i_P=5 D =" RoomEnd"
R: zsystem T_0=0.0 t=219.59739328771587 %=0.488
R: zbox ztype=Ball,2,1 t=219.59739328771587 R=- state=M Z.n=0 label=BallProto/7
   z.label=RoomEnd z.L=0 z.W=1 t0=219.59739328771587 x0=0.0 t1=219.59739328771587
   x1=0.0 δt=0.0 l=20 L=0 z.n=1 K=RoomTour1a i_P=5 D="vanished"
R: No new transition time found. System in static state. Halting.
```

This is an example of *reporting* output as described in Sec. 2.8. Parsing of this output is straightforward; any variables which are not needed can simply be ignored, and those required can be extracted.

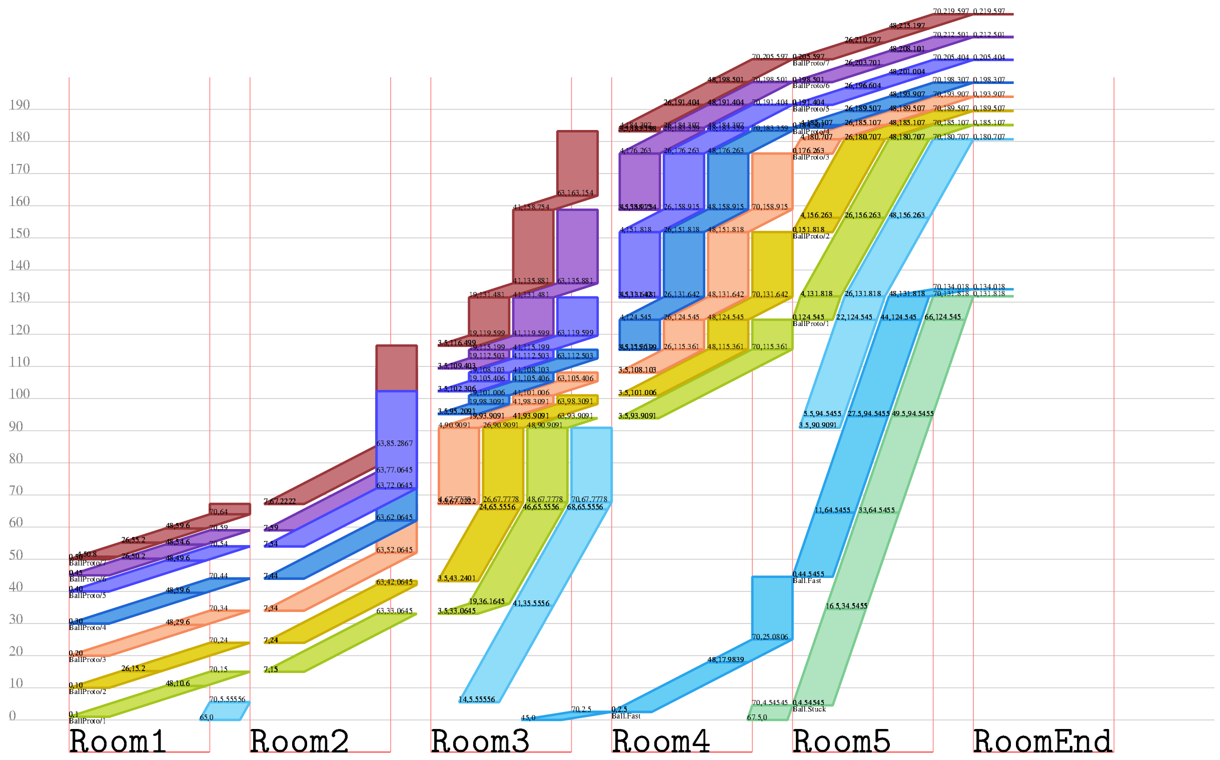

Figure 5.2: “Balls” moving through five “rooms”: The time axis is vertical, labeled in seconds on the left, progressing upwards. Each ball’s trajectory is a different colour. Their behaviour and starting points are variously described in the zsyntax input file. Black numbers along the trajectories are (x, t) values.

A Gawk program (which is unsophisticated, having some things like labels hard-coded) is provided which extracts the movement of the ZBOXEN and plots the output in a Postscript file:

```
$ cat _Testing_Pipe_Zboxen.zo | gawk -f Report_to_ps.awk > _Testing_Pipe_Zboxen.eps
```

The resulting plot is shown in Fig. 5.2.

A script is provided which contains the above commands and an additional line to convert to a PDF file. It requires Ghostscript to be installed, as it uses `ps2pdf`. To run the example and produce also the output graph:

```
$ ./01_RunAndGraphResults.sh
```

Mapping of report lines to segments of the plot is straightforward; for example the first line beginning `R: zbox ztype=Ball,2,1` in the `.zo` file shown corresponds to the first blue (🟦) trajectory segment visible inside `Room3`. ZBOX speeds and speed limits are also readily visible.

In this simple example, progress of the entire simulation can be seen in a single plot. In particular, it is to be noticed that within a *Pipe*-type ZBOX, passing is not allowed, whereas in `Room2` which is a *Span*-type ZBOX, overlapping trajectories are apparent. All ball ZBOXEN eventually make their way to the *Sink* ZBOX `RoomEnd` and quit the system.

Commenting of the line describing `Ball.Stuck`

```
# [zbox Ball.Stuck A=Ball n=1  z=<Room4>  P=<RoomTour1> i_P=3 x=67.5 v=0.55]
```

and re-running the model, the new plot shown in Fig. 5.3 is obtained. It can be seen that without the impediment that `Ball.Stuck` (🟩) represented, the system of rooms behaves differently.

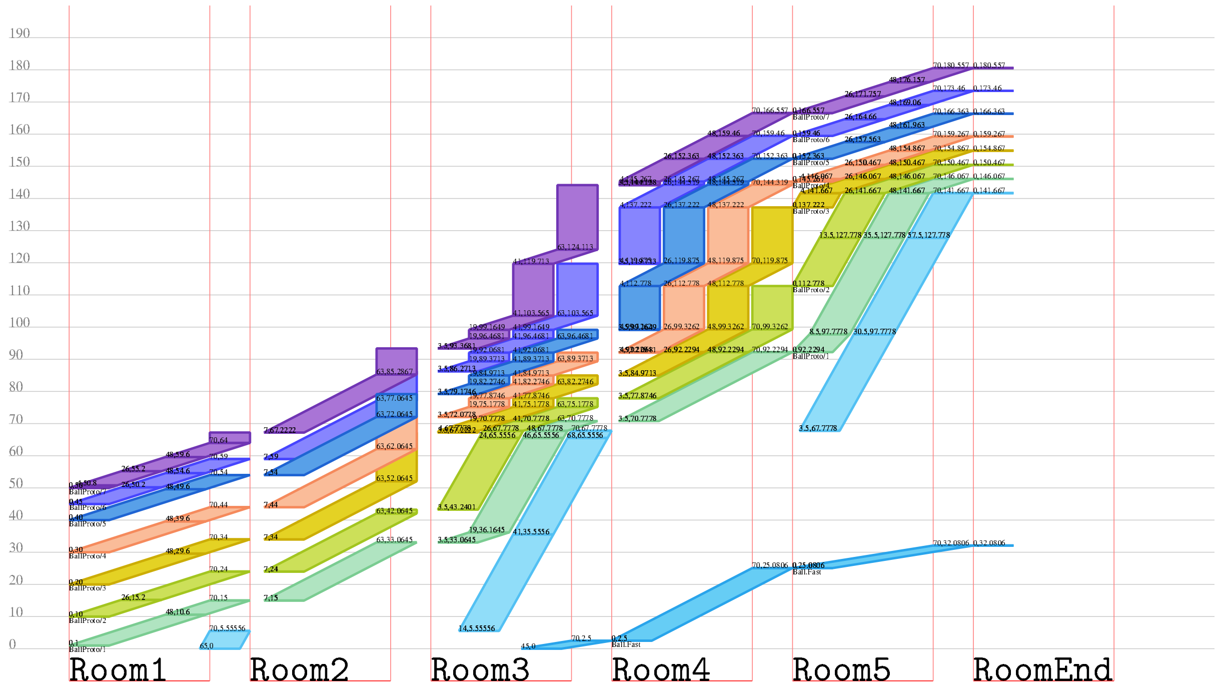

Figure 5.3: “Balls” moving through five rooms: The impediment `Ball.Stuck` has been removed from the system.

Chapter 6

Metro systems

6.1 Modelling a metro system

Reference will be made below to the various objects in the Zimulator framework; the reader is referred to Chapter 2 for the meaning of these terms (e.g. ZBOX, ZPATH, ZLINK).

The terminology adopted here is 'Metro System' and as mentioned above this term is inexact. It is important to realize that not all metro systems are identical, and source data for such systems may be expressed in various ways. Therefore there is no single 'correct' way to construct a model of such a system, and the level of detail and precision employed will vary depending on the model application and availability of data. In the present section will be discussed a reasonable way to map a generic metro system to the ingredients available in the Zimulator framework.

The system is deemed to have at least trains, stations, tracks and passengers. Beyond this minimum, there can be significant variation.

It is noted that while buses and the associated issues of more extensive traffic modelling are complex and beyond the scope of this discussion, buses travelling on dedicated or at least uncongested traffic lanes might reasonably be interpreted as 'trains' on 'tracks'. Buses in some cities are operated very much like trams or streetcars.

Stations in some systems, like tram or LRT systems, are merely platforms or sidewalks on which passengers may stand. Some systems have raised tracks and a station is thus comprised of a concourse with platforms accessible via escalators or stairwells. In subway systems, a typical station is comprised of an underground concourse with platforms at various extremedies, which concourse is accessible via a set of tunnels leading from entrances, which might be independent, or might connect to other structures like shopping malls or office centres. In some systems there are fare gates, through which passengers walk on entry to the system; often passengers must also pass through these when exiting. Sometimes these fare gates are part of the station infrastructure; sometimes they are part of the vehicles themselves.

In order that a comprehensive example be constructed below in 7.1, here it is anticipated that a full station model will be employed.

At a high level, modelling of vehicle depots could be accomplished within the present framework; nevertheless such modelling is system-dependent and is not typically important from a passenger point of view. Rather than maintain a conserved vehicle inventory via such depot modelling, it will be seen to be sufficient to generate trains at certain source points, and to discard these trains in sinks when they have traversed their routes.

6.1.1 Representation

Objects in the system which are represented by ZBOXEN are trains, tracks, bounds, platforms, corridors, concourses, gates, 'outgates' and stations; each of these is a particular ZTYPE as described in 6.1.4 below.

A generic template for the construction of stations is shown in Fig. 6.1.

Each train, represented by a ZBOX, is deployed on a ZPATH and successively visits ZBOXEN representing track intervals and representing bounds, which are the stops next to platforms where passengers may board or alight.

Each passenger is also a ZBOX; a passenger is created via a ZDEMAND by copying a prototype passenger and placing the ZBOX in a gate at the station of origin. A ZPATH is assigned which controls the subsequent travel. The passenger[1] successively visits ZBOXEN representing a gate, corridor, concourse, corridor and

[1]As described in detail in the Zimulator documentation, visiting a ZBOX implies *progression* through it, and moving to the

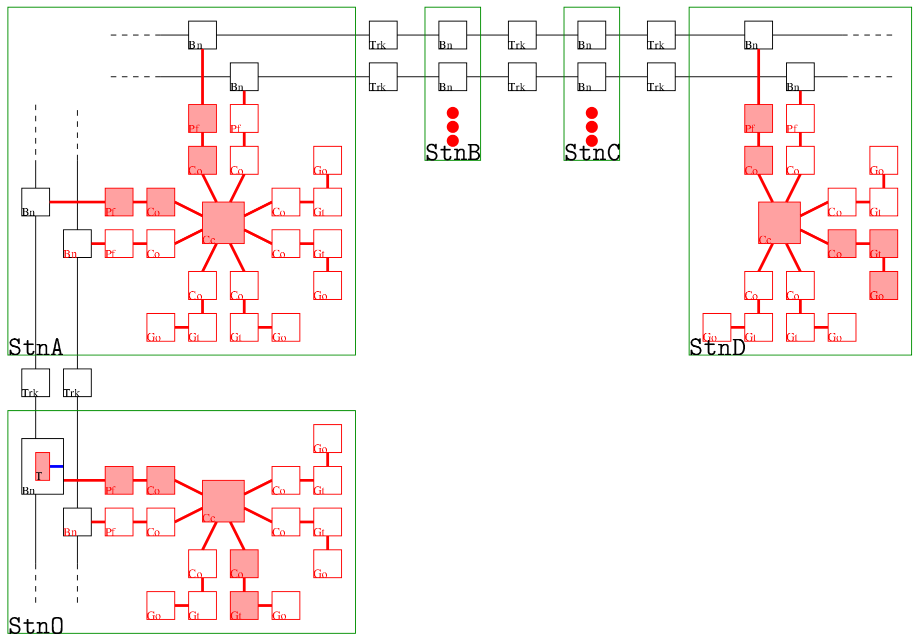

Figure 6.1: Five stations along sections of two transit lines: The red ZBOXEN may contain passengers; the black may contain trains. ZLINKS which are red allow passengers; those which are black allow trains. The mnemonics `Stn`, `Pf`, `Bn`, `Co`, `Cc`, `Gt`, `Go` and `Trk` refer to a ZBOX representing a Station, Platform, Bound, Corridor, Concourse, Gate, 'GateOut' and Track Segment, respectively. The blue ZLINK is an implied ZLINK, between the Train `T` and Bound, and is traversable by passengers. The circular dots represent details not shown.

platform, by traversing each and then shifting via ZLINK to the next. The passenger then shifts to a train within the bound ZBOX associated with the platform; this is made possible by an implicit ZLINK from the train to the bound, traversible by passengers. If later in the trip a transfer is to be undertaken, the passenger shifts out to a platform, then to a corridor, concourse, corridor and new platform at the transfer station, before shifting to a train in the bound associated with the new platform. The passenger disembarks by shifting out of the train to a platform, then eventually to a gate and finally an 'outgate' which is a sink via which the passenger quits the system.

In this way, a passenger's full trip from station 'O' to station 'D' in the diagram would involve visiting all the shaded ZBOXEN.

The station construction displayed in Fig. 6.1 is general enough to model many real-world stations. Of course, some systems are constructed such that common transfers may be made by simply walking across a platform; this could be modeled, for example, by instead using a simpler structure with a short corridor ZBOX between the platforms.

It is emphasized that correct interpretation of configurations like that shown in Fig. 6.1 involves recognizing that time spent by a ZBOX occurs in *progressing* through various container ZBOXEN, and not in instantaneous *shifting* from one to the next along ZLINKS.

6.1.2 zsyntax files

The data mentioned in the previous section are used to build the metro system using ZBOXEN. The metro system is thereby specified in the zsyntax files listed below.

Separation of zsyntax simulator input into particular files is entirely arbitrary; the simulator is simply provided with zsyntax input, and it can be stored in one or many files as seen fit by the user.

6.1.3 Simulator input files

The simulator expects the system to be expressed in ZSYNTAX.

There are several sets of information which must be expressed in order to model the system; these include:

next ZBOX involves *shifting* to it via a ZLINK.

- The zsystem itself (a few global parameters)
- A number of ztypes covering each kind of obejcts in the system to be modeled by a zbox
- Static zboxen describing each metro station as in Fig. 6.1
- Static zboxen describing each track segment in the network.
- Static zlinks describing connections of the track segments to the bounds at the stations.
- Prototype zboxen from which trains and passengers are created.
- zpaths which describe the paths through the system taken by each train line.
- A number of zschedules which indicate the times for deployment of trains on paths.
- A number of zdemand objects which describe passengers with intended destinations to be injected into the system.

In Sec. 6.1.4 below is described the mapping between metro-system parameters and the individual attributes of Simulator objects like zboxen.

In the Madrid example in Chapter 6, the above list will be distributed among six files which will be discussed in detail.

6.1.4 System parameters

The main concept in modelling a system is that progress of a zbox through its container corresponds to progression of some modeled degree of freedom. In the case of a Track zbox, the position of the Train zbox inside is the spatial position of the train on that section of track; in the case of a Bound zbox, the progression simply corresponds to the time taken while the train is waiting at a Bound (i.e. dwelling time).

In this way most of the parameters associated with the metro system are expressed as attributes of certain ztypes, and so are exhibited by the various zboxen in the system.

Some of these parameters are shown and described below, by extracting example pieces of zsyntax. Since these extracts are not in context, the particular numerical values (e.g. `L=1380` or `S=100`) are unimportant and are merely illustrative.

Model parameters represented in the `Train` ztype include train capacity and train speed:

```
[ztype A=Train n=1 q=1 C={Pax 1} m=Bag l=30 L=1380 N=1380 S=0 χ=<TrainDoors>
  v=20 R=S]
```

- `m=Bag` – Passengers need not traverse when contained in a train.
- `L=1380` or `N=1380` – controls the capacity of a train. If L is used, then it is measured in the same units as `Pax`.l. Train capacity could be determined via some kind of calibration, and will be discussed later.
- `v=20` – Speed of a train. Length units match those of `Track`.L, and m/s are used here.
- `S=0` – Space ‘between’ passengers.

Model parameters represented in the `Bound` ztype include dwelling time:

```
[ztype A=Bound n=1  m=Pipe C={Train 1} L=45 V=1 N=1]
```

- `m=Pipe` – Controls how trains may stop at a bound.
- `L=45` – Controls how long a train stops at a bound, in conjunction with `Train`.v and `Train`.l.
- `V=1` – Limiting velocity for trains; this limits `Train`.v.

The dwelling time in this example is 30 s; since the train has length 30, it can progress 15 out of 45 with limited velocity 1.

Model parameters represented in `Track` include travel time between stations:

```
[ztype A=Track n=1 m=Pipe C={Train 1} L=1000 S=100]
```

- `m=Pipe` – Trains may not pass each other on a track.
- `L=1000` – The distance between two subsequent stations. This is overridden in specific zboxen of this ztype.
- `S=100` – Minimal distance between successive trains.

Model parameters represented in `Concourse` and `Corridor` include walking distances within stations:

```
[ztype A=Concourse n=1 m=Bag C={Pax 1} N=10000]
[ztype A=Corridor n=1 m=Span C={Pax 1} L=200 W=10]
```

- `L=200 W=10` – Capacity of a corridor will be 2000 passengers. The walking distance through is 200 m. These are overridden by each `Co` ZBOX. In particular, L could be determined via calibration.
- `N=10000` – The capacity of the `Concourse` is typically overestimated in order not to produce an artificial bottleneck in the system.

The walking *times* within the system are then determined by passenger walking speeds, which are specified either as an attribute of a prototype Passenger within a ZSOURCE inside ZDEMAND, or specified within the ZSOURCE as the parameters of a log-normal distribution.

Model parameters represented in `Platform`:

```
[ztype A=Platform n=1 m=Bag C={Pax 1} L=4000]
```

- `m=Bag` – Passengers do not traverse a platform; they simply wait there.
- `L=4000` – Capacity of a platform. This is not usually important for simulation, unless there is special interest in saturation of platforms.

The ZTYPES of `Gate` and `Platform` have a containment type of 'Bag', thereby having only capacity but no traversal attributes.

6.1.5 Calibration

In very general terms, there may be 'ground truth' sources of data which can be used to calibrate aspects of the simulation. For example, there may be sensors on train platforms which can be used to calibrate [effective] train speeds. There may be a farecard system which records the precise times of passengers 'tapping' in an out of the system; this could be used to calibrate not only train speeds but also walking times within the system and possibly other properties like effective train capacities.

In section 8 an illustrative example of such calibration will be given, calibrating the two-parameters of the log-normal distribution used for passenger walking speeds. Although the system approximately modeled is a real-world system, the 'ground-truth' data utilised will be synthetic.

Chapter 7

Madrid Metro

7.1 Madrid Metro

In this chapter will be developed a Zimulator model for the Madrid Metro system. The model is to be constructed using only public sources of data, which are in cases of insufficiency supplemented with synthetic data. The resultant model is intended here solely to function as an illustrative example of metro-system simulation.

All public data are extracted from Wikipedia.[1] All source addresses are included as fields within the raw metro-system data files described below.

The level of detail of the model will be influenced by the form and details of available data. Such a model is sufficient for illustrative purposes; in the case of quantitative application to a metro system, actual system data would be expected to produce a more realistic model.

7.1.1 Metro system public data

Available data describe the geometrical layout of the metro network, in addition to the transit lines and connectivity. The extracted data have been stored in three source files, the form of which is briefly described here. This file format is nothing more than an intermediate tool to create the zsyntax files which describe the system. It is a simple ad-hoc format particular to the Madrid system, but may be useful, with possible embellishments, in describing other metro systems.

`Line_List.csv` – This defines an index, code, name and source location for each of the 16 metro lines. An example row is:

```
11,s,L11,Plaza Elíptica - La Fortuna,https://en.wikipedia.org/wiki/Line_11_(Madrid_Metro)
```

The field 's' indicates that the line is 'straight' as opposed to a loop line, which would be indicated by 'l'.

`Station_List.csv` – This defines a three-letter code, name, latitude and longitude for each metro station.
An example row is:

```
SLS,San Lorenzo,40.4744713,-3.6395754,https://en.wikipedia.org/wiki/San_Lorenzo_(Madrid_Metro)
```

The three-letter codes are generated for the purpose of the simulation, and are not official.

`Itinerary_List.csv` – This file defines the paths taken by each line in the system, as a list of stations. An example sequence of rows is:

```
5,1,ADO,0
5,2,ECE,802.249
5,3,LEA,1407.06
. . .
5,30,PAL,735.651
5,31,CAT,579.064
5,32,CDC,1226.36
```

This section of 32 rows in total indicates that transit line number 5 travels through 32 stops, from ADO to CDC. The fourth field indicates the distance in metres to the given stop from the previous stop. In the present case the distances have been estimated using the latitude and longitude coordinates of each station, taking the geodesic distance and multiplying arbitrarily by 1.1.

These CSV files will be taken to be the raw system data; below in 7.1.4 these data will be used to engineer the system in terms of Zimulator objects.

[1] Data are taken from https://en.wikipedia.org/wiki/Madrid_Metro and associated pages.

7.1.2 Synthetic data

The data mentioned in the previous section do not describe the system comprehensively; while the infrastructure is thereby specified, the supply and demand remain undetermined. These are generated synthetically.

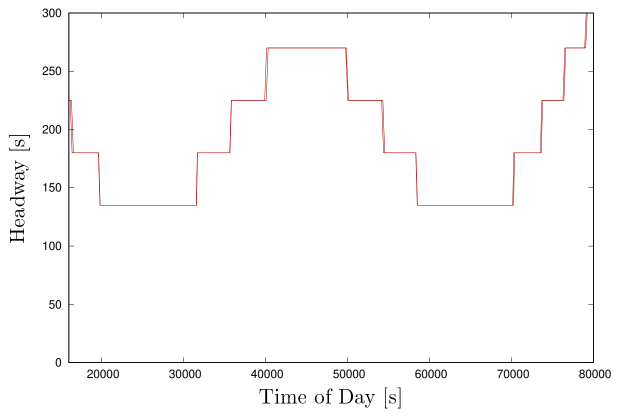

Figure 7.1: Headway depends on time of day.

Trains are supplied on each of the lines at intervals, all day long from approximately 04:30 to 25:00. The headway, shown in Fig. 7.1, (time between subsequent train deployments) is modeled very simply, to accommodate a higher train frequency during peak hours of near 07:00 and 19:00.

The passenger demand is generated to represent the general flow within an urban area; the flow of commuters is generally towards the city centre during the morning peak period, and generally away from the centre during the evening peak. There is no intention to model accurately the flow in the actual city; the intention here is to provide an example day of demand which is not entirely unrealistic. The sequence of passenger flows throughout the day is indicated in Fig. 7.2.

7.1.3 Station template

The arrangement of each station in the metro system is that presented earlier in Fig. 6.1, with four gates associated with each station. This is an arbitrary choice. At a station at which n transit lines stop, there are $2n$ platforms, so as to accommodate each direction of each line. In a more detailed model, the number of gates, number of platforms, and also the connectivity among them could be adjusted to reflect actual station layout.

7.1.4 zsyntax files

The data mentioned in the previous section are used to build the metro system using ZBOXEN. The Metro system is thereby specified in the zsyntax files listed below.

Separation of zsyntax input into particular files is entirely arbitrary; the simulator is simply provided with zsyntax input, and it can be stored in one or many files as seen fit by the user.

- `00_StaticTypes.zim` – defines overall system and ZTYPES for all objects.
- `_01_Stations.zim` – contains a description of all stations.
- `_02_Tracks.zim` – defines all tracks and their connections to stations.
- `_03_Paths.zim` – defines the paths through the system which are followed by trains.
- `_04_Schedules.zim` – specifies the times at which trains are deployed.
- `_05_Demand.zim` – contains ZDEMAND objects describing deployment of passengers during the day.

In the following subsections the content of each of these zsyntax input files will be explained in detail.

Static types: `00_StaticTypes.zim`

The ZSYSTEM is defined, with particular reference and simulation times, and a certain level of reporting:

```
[zsystem Madrid T_0=0 t_0=05:00 t_1=23:00 Δt=600.0 R=S ]
```

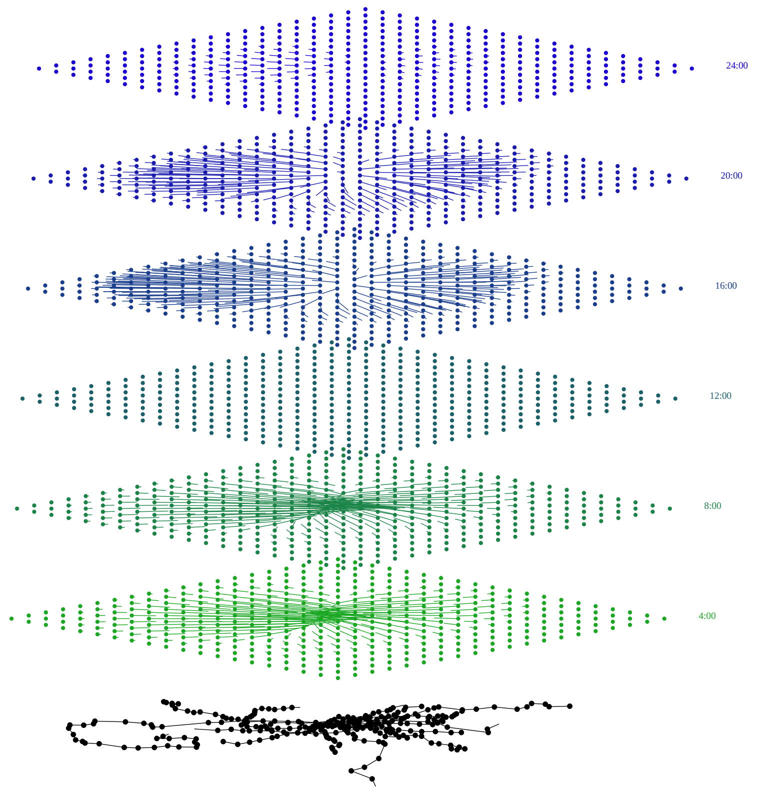

Figure 7.2: Large-scale demand throughout the day. The figure shows the directional local net flow of passengers in a two-dimensional plane, at various times during the day. For spatial reference, a schematic of the transit lines is shown below the first slice.

The ZTYPES in the system are next defined. A discussion of how the values specified here relate to metro-system parameters such as dwelling times and walking speeds can be found above in 6.1.4.

There are ZTYPES for dynamical objects in the system:

```
[ztype A=Pax n=1 v=2.0 l=1 R=S2
  Z={ Train 1 Platform 1 Concourse 1 Corridor 1 } ]
[ztype A=Train n=1 q=1 C = { Pax 1 } m=Bag l=100 L=1380
  N=1380 S=0 χ=<TrainDoors> v=21.04 R=S1d ]
```

Passengers may *sleep*, as described in Chapter 2, when waiting inside certain containers. The train capacity and size could be over-ridden by trains deployed on each line.

A number of objects are able to contain passengers:

```
[ztype A=Station n=1 C={ Concourse 1 Gate 1
    OutGate 1 Platform 1 Bound 1 Corridor 1 } ]
[ztype A=Platform n=1 m=Bag C={Pax 1} N=3000 ]
[ztype A=Concourse n=1 m=Bag C={Pax 1} N=3000 ]
[ztype A=Corridor n=1 m=Span C={Pax 1} L=200 W=5 ]
[ztype A=Gate n=1 m=Bag C={Pax 1} $=2 ]
[ztype A=OutGate n=1 C={ Pax 1 } m=Sink ]
```

These lines define that a `Station` may contain a number of other objects, and then define those object types, in correspondence with the station model which was chosen earlier. The capacity for the Platform, Concouse and Corridor could be overridden at each station if more detail were required.

Passengers will utilise the above objects; trains will utilise the following.

```
[ztype A=Bound n=1 m=Pipe C={Train 1} L=124 V=1 N=1 ]
[ztype A=Track n=1 m=Pipe C={Train 1} L=1000 S=100 ]
```

The L parameter for a Bound determines the dwelling time at the platform. The length of 1000 for a Track is always over-ridden with an actual track length, as shown below.

Finally, the types for train sources and sinks are defined:

```
  [ztype A=TrainSink n=1 C={ Train 1 } m=Sink ]
  [ztype A=TrainSource n=1 m=Bag C={Train 1} ]
  [zlink TrainDoors A={ Pax 1 0 } ]
```

The last row here describes the implicit ZLINK which connects a train to its container. As described earlier, this is utilised by passengers when the train is inside a bound ZBOX for boarding or alighting.

Station descriptions: `_01_Stations.zim`

This file contains a specification of every station in the metro system. As an example of a single such station:

```
# ----------- Station: PDA -- Puerta de Arganda -----------
[zbox PDA i=latlon:40.4013157,-3.5959768 A=Station n=1 ]
[zbox PDA_Cc A=Concourse n=1 z=<PDA> ]
# ----------- Gates for PDA
[zlink μ=<PDA_Cc> ν=<PDA_GtCo_1> A={ Pax . 0 } ]
[zbox PDA_GtCo_1 A=Corridor n=1 L=110 z=<PDA> ]
[zlink μ=<PDA_GtCo_1> ν=<PDA_Gt_1> A={ Pax . 0 } ]
[zbox PDA_Gt_1 A=Gate n=1 z=<PDA> ]
[zlink μ=<PDA_Gt_1> ν=<PDA_Gt_1_out> A={ Pax . 0 } ]
[zbox PDA_Gt_1_out A=OutGate n=1 z=<PDA> ]
[zlink μ=<PDA_Cc> ν=<PDA_GtCo_2> A={ Pax . 0 } ]
[zbox PDA_GtCo_2 A=Corridor n=1 L=120 z=<PDA> ]
[zlink μ=<PDA_GtCo_2> ν=<PDA_Gt_2> A={ Pax . 0 } ]
[zbox PDA_Gt_2 A=Gate n=1 z=<PDA> ]
[zlink μ=<PDA_Gt_2> ν=<PDA_Gt_2_out> A={ Pax . 0 } ]
[zbox PDA_Gt_2_out A=OutGate n=1 z=<PDA> ]
[zlink μ=<PDA_Cc> ν=<PDA_GtCo_3> A={ Pax . 0 } ]
[zbox PDA_GtCo_3 A=Corridor n=1 L=130 z=<PDA> ]
[zlink μ=<PDA_GtCo_3> ν=<PDA_Gt_3> A={ Pax . 0 } ]
[zbox PDA_Gt_3 A=Gate n=1 z=<PDA> ]
[zlink μ=<PDA_Gt_3> ν=<PDA_Gt_3_out> A={ Pax . 0 } ]
[zbox PDA_Gt_3_out A=OutGate n=1 z=<PDA> ]
[zlink μ=<PDA_Cc> ν=<PDA_GtCo_4> A={ Pax . 0 } ]
[zbox PDA_GtCo_4 A=Corridor n=1 L=140 z=<PDA> ]
[zlink μ=<PDA_GtCo_4> ν=<PDA_Gt_4> A={ Pax . 0 } ]
[zbox PDA_Gt_4 A=Gate n=1 z=<PDA> ]
[zlink μ=<PDA_Gt_4> ν=<PDA_Gt_4_out> A={ Pax . 0 } ]
[zbox PDA_Gt_4_out A=OutGate n=1 z=<PDA> ]
```

The `i` field encodes information which will simply pass through the Zimulator unprocessed; the station latitude and longitude is useful for simple plotting of output data. The remainder of the station is a set of static ZBOXEN and ZLINKS as described in 6.1.1 and shown in Fig. 6.1.

Track network: `_02_Tracks.zim`

This file contains a description of all tracks which link stations, along each metro line's route. It is noted that lines (apparently) do not share track segments within the Madrid system; if this were the case then generation of this file would be slightly more complex. The first few lines of the file follow:

```
# ----------- Line L1(Pinar de Chamartín - Valdecarros) :
[zbox Train_proto/L1 A=Train n=1 π=1 z=<Source_L1>]
[zsource Train_Source_L1 ϕ=<Train_proto/L1> m=One o=Container ]
[zbox Source_L1 A=TrainSource n=1 ]
[zbox Sink_L1 A=TrainSink n=1 ]
```

Each metro line has its own prototype Train ZBOX and Train ZSOURCE, which are used to deploy 'fresh' trains on the line. The prototype train is contained in the ZBOX labeled `Source_L1`, and then copied to produce deployed trains which begin in this same container. Each metro line also has its own sink, where trains which have run the whole line quit the system. The next few lines of the file are:

```
[zbox Trk_L1_0_PDC_BBA L=978.921 A=Track n=1
  i=latlon2:40.4801375,-3.6667999,40.4768117,-3.6763701 ]
[zlink μ=<PDC_Bn_L1_0> ν=<Trk_L1_0_PDC_BBA> A={ Train . 1 } ]
[zlink μ=<Trk_L1_0_PDC_BBA> ν=<BBA_Bn_L1_0> A={ Train . 1 } ]
[zbox Trk_L1_0_BBA_CCH L=822.981 A=Track n=1
  i=latlon2:40.4768117,-3.6763701,40.472101,-3.6826857 ]
[zlink μ=<BBA_Bn_L1_0> ν=<Trk_L1_0_BBA_CCH> A={ Train . 1 } ]
[zlink μ=<Trk_L1_0_BBA_CCH> ν=<CCH_Bn_L1_0> A={ Train . 1 } ]
[zbox Trk_L1_0_CCH_AIA L=878.787 A=Track n=1
  i=latlon2:40.472101,-3.6826857,40.4668983,-3.6891989 ]
[zlink μ=<CCH_Bn_L1_0> ν=<Trk_L1_0_CCH_AIA> A={ Train . 1 } ]
[zlink μ=<Trk_L1_0_CCH_AIA> ν=<AIA_Bn_L1_0> A={ Train . 1 } ]
```

These represent the track segments travelling from PDC to BBA, BBA to CCH and CCH to AIA stations. Each of these segments is connected via ZLINKS to the Bound ZBOX at the station at each end. This is an implementation of the diagram in Fig. 6.1, with trains expected to subsequently visit Bound and Track ZBOXEN. As with Station ZBOXEN, the `i` field is unused by the simulator, but appears in output as a convenience.

Train trajectories: `_03_Paths.zim`

Train ZBOXEN follow ZPATHS through the system which are defined here. An example ZPATH from this file is:

```
# ----------- Line ML1_1(Pinar de Chamartín - Las Tablas) complete path:
[zpath Train_ML1_1_Line A=Train n=1 m=Open Λ={ [zs ϕ=<LTL_Bn_ML1_1>]
 [zs ϕ=<Trk_ML1_1_LTL_LAE>] [zs ϕ=<LAE_Bn_ML1_1>] [zs ϕ=<Trk_ML1_1_LAE_MTM>]
 [zs ϕ=<MTM_Bn_ML1_1>] [zs ϕ=<Trk_ML1_1_MTM_BIB>] [zs ϕ=<BIB_Bn_ML1_1>]
 [zs ϕ=<Trk_ML1_1_BIB_LDV>] [zs ϕ=<LDV_Bn_ML1_1>] [zs ϕ=<Trk_ML1_1_LDV_IOS>]
 [zs ϕ=<IOS_Bn_ML1_1>] [zs ϕ=<Trk_ML1_1_IOS_VDC>] [zs ϕ=<VDC_Bn_ML1_1>]
 [zs ϕ=<Trk_ML1_1_VDC_FDL>] [zs ϕ=<FDL_Bn_ML1_1>] [zs ϕ=<Trk_ML1_1_FDL_PDC>]
 [zs ϕ=<PDC_Bn_ML1_1>] [zs ϕ=<Sink_ML1>] } ]
```

This defines the trajectory for the ML1 line, travelling in direction 1. (Arbitrarily the two directions along a route have been termed 0 and 1.) Within ths ZPATH are 18 ZSTOP objects, referencing ZBOXEN corresponding to alternating Bounds and Tracks in succession. Lastly, the Sink for the ML1 line is visited. A train following such a ZPATH will quit the system upon arrival there.

Train schedules: `_04_Schedules.zim`

Here are expressed all the times during the day when a train is to be deployed on each of the metro lines. This is done with a ZSCHEDULE object for each line, for example:

```
[zschedule Sch_ML3_1 T_0 = 0 T={ 15618 15897 16169 16434 16692 16943 17188
 . . . . .
 89065 89524 89990 } S=<Train_Source_ML3> P=<Train_ML3_1_Line> ]
```

The provided source is the ZSOURCE defined for the metro line, ML3 in this case. The path given the Train will be the ML3 direction-1 ZPATH. It can be seen that the first train to be deployed on ML3 will be at 4:20:18.

Passenger demand: `_05_Demand.zim`

In this example, the passengers are all provided at once in a single ZDEMAND object which looks like the following:

```
[zdemand Madrid_Day_pax
  S=[zsource Pax_Src ϕ=[zb Pax A=Pax n=1 π=1] m=One o=Teleport]
  T_0=0 Reference=%s
  L={ <TDM_Gt_1> <TDM_Gt_1_out> <TDM_Gt_2> <TDM_Gt_2_out>
    <TDM_Gt_3> <TDM_Gt_3_out> <TDM_Gt_4> <TDM_Gt_4_out>
    <RAR_Gt_1> <RAR_Gt_1_out> <RAR_Gt_2> <RAR_Gt_2_out> <RAR_Gt_3> <RAR_Gt_3_out>
    . . .
    <DEL_Gt_1> <DEL_Gt_1_out> <DEL_Gt_2> <DEL_Gt_2_out>
    <DEL_Gt_3> <DEL_Gt_3_out> <DEL_Gt_4> <DEL_Gt_4_out>
  }
  D={
    1972 1671 14400 1
    . . .
    294 1563 89999 1
  } # total of 1750015 passengers for the day.
]
```

A ZSOURCE from which to procure ‘fresh’ passengers is provided, within which is a prototype passenger ZBOX. Of course, these could have been provided elsewhere (say, in `00_StaticTypes.zim`) and referenced, but as they are conceptually express attributes of the demand, they appear here in-line.

`L` is a list of ZBOXEN in the system which can be specified as origins or destinations for the Passengers deployed. Conveniently, even indices correspond to origins, while odd indices correspond to destinations; this is merely a convention.

`D` specifies a list of passengers to deploy; each set of four numbers specifies an origin and destination index in the set L, a time of day, and a number of passengers.

7.1.5 Running the simulation

The model can be run using the command-line interface to the core simulator. The resulting reporting output will be parsed and analysed in Sec. 7.1.6 below. In the present section, it will be shown how to produce and interpret verbose output during a simulation; this is useful in 'debugging' while assembling a model of a metro system and understanding that it functions as intended.

A bash script `02_Run_simulation.sh` is provided, which simply contains (without `z=30`) the command

```
java  -jar CL.jar -razo z=30 zl=45 I=00_StaticTypes.zim \
             I=_01_Stations.zim     I=_02_Tracks.zim \
             I=_03_Paths.zim        I=_04_Schedules.zim \
             I=_05_Demand.zim       R=out/madrid_01.zo
```

indicating that the above collection of input files is to be used to generate an output `madrid_01.zo`. While the simulation is running, the system status will be displayed on the terminal every 30 seconds, up to a maximum of 45 rows of text. The simulation could equivalently be run via:
`$ ./02_Run_Simulation.sh z=30`

Since periodic verbose updates have been enabled with `z=30`, as the simulation runs the terminal will display output like the following.

```
X=2130 T_0=0.0 t_0=05:00:00.000 t_1=23:00:00.000 t=05:11:30.000
1.06%  133.48× (avg 95.34×) N[0]=4 // marks:1,31471
zobjects: [0]:    46 [T]:  6376 [Z]: 25938 [S]: 13881
zpaths: 722
zboxen:
|       [0]     [T]     [Z]             [S]     Total   ztype
|       46      6343    25938   (66%)   6487    38814
|       ·       ·       ·               275     275     Station
|       ·       ·       ·               275     275     Concourse
|       ·       ·       ·               1100    1100    Gate
|       ·       ·       ·               1100    1100    OutGate
|       ·       ·       ·               654     654     Platform
|       ·       ·       ·               654     654     Bound
|       ·       ·       ·               1754    1754    Corridor
|       1       91      ·               16      108     Train
|       45      6252    25938   (80%)   1       32236   Pax
|       ·       ·       ·               626     626     Track
|       ·       ·       ·               16      16      TrainSink
|       ·       ·       ·               16      16      TrainSource
|       ·       ·       ·               275     275     Concourse       ∈ Station
|       ·       ·       ·               1100    1100    Gate    ∈ Station
|       ·       ·       ·               1100    1100    OutGate ∈ Station
|       ·       ·       ·               654     654     Platform        ∈ Station
|       ·       ·       ·               654     654     Bound   ∈ Station
|       ·       ·       ·               1754    1754    Corridor        ∈ Station
|       45      ·       ·               ·       45      Pax     ∈ Gate
|       ·       ·       14834   (100%)  ·       14834   Pax     ∈ Platform
|       ·       35      ·               ·       35      Train   ∈ Bound
|       ·       6252    ·               ·       6252    Pax     ∈ Corridor
|       ·       ·       11104   (100%)  ·       11104   Pax     ∈ Train
|       ·       56      ·               ·       56      Train   ∈ Track
|       1       ·       ·               16      17      Train   ∈ TrainSource
```

Some of the information pertains only to internal degrees of freedom, but generally the output is intended to indicate progress of the simulation, especially during debugging of a system.

$X = 2130$ indicates that 2130 passes of the system have been computed. The base time T_0 and simulation window (t_0, t_1) are indicated, as is the current simulation time t. In this example, at t =05:11:30 the simulation is 1.06% complete, running at a relative speed of 133.48× (i.e. the simulator is running this factor faster than reality), and has averaged a relative speed of 95.34× so far. The latest number of passes $N[0]$ of the [0] system list required for state resolution is 4. The indicator 'marks' is not of interest to the user.

The columns [0], $[T]$, $[Z]$ and $[S]$ are the four 'Syslists' which are maintained internally, representing ZBOXEN which are either in discrete states, timed states, sleeping, or static, respectively. The $\in$ symbols, when present, indicate first-level containment. In this example, there are 32236 Passenger ZBOXEN, distributed among Gates, Platforms, Corridors and Trains. 1104 of them happen to be riding trains at 05:11:30.

This $z = 30$ display is one method of seeing how the system is developing. Another way is to enable *verbosity*. As described, verbosity (as distinguished from *reporting*) is not intended to be parsed by machine, but is another method by which a human can monitor a running simulation.

```
$ ./02_Run_simulation.sh -v
```

The result is that many, many rows of output will be dumped to the terminal, indicating information about ZBOXEN and other objects and their state transitions.

Passengers in the present model are generated via the prototype passenger ZBOX φ=`[zb Pax A=Pax n=1` π`=1]`. Therefore, the ZDEMAND gives them labels based on this ZBOX label; they will be `Pax/1`, `Pax/2`, etc. A single such passenger can be followed by specifying that verbosity should be applied only to one label. Following the progress of passenger, say, 6502 in this way can be thus accomplished with:

```
$ ./02_Run_simulation.sh v=Pax/6502
```

which produces on the terminal a number of rows narrating the life and times of that particular passenger:

```
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'BRN_Gt_3'  t=18140.0=05:02:20.000   State:M with  Ψ.t=18140.0=05:02:20.000  t=18140.0=05:02:20.000 x=0.0 δx=0.0 δt=0.0
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'BRN_Gt_3'  t=18140.0=05:02:20.000  MOVE state within a m=5 Bag zbox.BRN_Gt_3
zbox: 26802 [D]  'Pax/6502'  (9,1)  ∊ 'BRN_Gt_3'  t=18140.0=05:02:20.000   State:D with  Ψ.t=18140.0=05:02:20.000  t=18140.0=05:02:20.000 x=0.0 δx=0.0 δt=0.0
zbox: 26802 [D]  'Pax/6502'  (9,1)  ∊ 'BRN_Gt_3'  t=18140.0=05:02:20.000  MOVED. currently inside BRN_Gt_3 Checking what to do next
zbox: 26802 [S]  'Pax/6502'  (9,1)  ∊ 'BRN_Gt_3'  S mode in BRN_Gt_3; desired next is zbox BRN_GtCo_3
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'BRN_GtCo_3'  t=18140.0=05:02:20.000  Following zpath. This index is 1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'BRN_GtCo_3'  t=18140.0=05:02:20.000  New container: __________________ -> BRN_GtCo_3
```

. . . (many lines)

```
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3'  t=20435.96007604563=05:40:35.960  Following zpath. This index is 22
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3'  t=20435.96007604563=05:40:35.960  New container: _________________ -> NNN_Gt_3
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3'  t=20435.96007604563=05:40:35.960   State:M with  Ψ.t=20435.96007604563=05:40:35.960  t=20435.96007604563=05:40:35.960 x=0.0 δx=0.0 δt=0.0
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3'  t=20435.96007604563=05:40:35.960  MOVE state within a m=5 Bag zbox.NNN_Gt_3
zbox: 26802 [D]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3'  t=20435.96007604563=05:40:35.960   State:D with  Ψ.t=20435.96007604563=05:40:35.960  t=20435.96007604563=05:40:35.960 x=0.0 δx=0.0 δt=0.0
zbox: 26802 [D]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3'  t=20435.96007604563=05:40:35.960  MOVED. currently inside NNN_Gt_3 Checking what to do next
zbox: 26802 [S]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3'  S mode in NNN_Gt_3; desired next is zbox NNN_Gt_3_out
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3_out'  t=20435.96007604563=05:40:35.960  Following zpath. This index is 23
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3_out'  t=20435.96007604563=05:40:35.960  New container: _________________ -> NNN_Gt_3_out
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3_out'  t=20435.96007604563=05:40:35.960  Container NNN_Gt_3_out is a sink; Pax/6502 disappears.
```

It is noted that the Passenger ends his trip by quitting the system at a *Sink* `NNN_Gt_3_out` as expected. Transitions to a new container always contain the line of several underscores; it is easier to follow the progress of `Pax/6502` by:

```
$ ./02_Run_simulation.sh v=Pax/6502 | grep _____
```

which appears on the terminal as:

```
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'BRN_GtCo_3'  t=18140.0=05:02:20.000  New container: _________________ -> BRN_GtCo_3
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'BRN_Cc'  t=18204.5=05:03:24.500  New container: _________________ -> BRN_Cc
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'BRN_PfCo_ML2_1'  t=18204.5=05:03:24.500  New container: _________________ -> BRN_PfCo_ML2_1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'BRN_Pf_ML2_1'  t=18253.5=05:04:13.500  New container: _________________ -> BRN_Pf_ML2_1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'Train_proto/ML2/1'  t=18253.5=05:04:13.500  New container: _________________ -> Train_proto/ML2/1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'CJC_Pf_ML2_1'  t=18834.716730038024=05:13:54.716  New container: _________________ -> CJC_Pf_ML2_1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'CJC_PfCo_ML2_1'  t=18883.716730038024=05:14:43.716  New container: _________________ -> CJC_PfCo_ML2_1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'CJC_Cc'  t=18932.716730038024=05:15:32.716  New container: _________________ -> CJC_Cc
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'CJC_PfCo_L10_1'  t=18932.716730038024=05:15:32.716  New container: _________________ -> CJC_PfCo_L10_1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'CJC_Pf_L10_1'  t=18981.716730038024=05:16:21.716  New container: _________________ -> CJC_Pf_L10_1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'Train_proto/L10/7'  t=19123.614068441064=05:18:43.614  New container: _________________ -> Train_proto/L10/7
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'TTR_Pf_L10_1'  t=19608.58365019011=05:26:48.583  New container: _________________ -> TTR_Pf_L10_1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'TTR_PfCo_L10_1'  t=19608.58365019011=05:26:48.583  New container: _________________ -> TTR_PfCo_L10_1
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'TTR_Cc'  t=19657.58365019011=05:27:37.583  New container: _________________ -> TTR_Cc
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'TTR_PfCo_L1_0'  t=19657.58365019011=05:27:37.583  New container: _________________ -> TTR_PfCo_L1_0
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'TTR_Pf_L1_0'  t=19706.58365019011=05:28:26.583  New container: _________________ -> TTR_Pf_L1_0
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'Train_proto/L1/12'  t=19838.02091254753=05:30:38.020  New container: _________________ -> Train_proto/L1/12
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Pf_L1_0'  t=20322.46007604563=05:38:42.460  New container: _________________ -> NNN_Pf_L1_0
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_PfCo_L1_0'  t=20322.46007604563=05:38:42.460  New container: _________________ -> NNN_PfCo_L1_0
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Cc'  t=20371.46007604563=05:39:31.460  New container: _________________ -> NNN_Cc
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_GtCo_3'  t=20371.46007604563=05:39:31.460  New container: _________________ -> NNN_GtCo_3
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3'  t=20435.96007604563=05:40:35.960  New container: _________________ -> NNN_Gt_3
zbox: 26802 [M]  'Pax/6502'  (9,1)  ∊ 'NNN_Gt_3_out'  t=20435.96007604563=05:40:35.960  New container: _________________ -> NNN_Gt_3_out
```

It is apparent that the passenger travelled from station BRN on `Train_proto/ML2/1` to station CJC, then from CJC on `Train_proto/L10/7` to TTR, and finally on `Train_proto/L1/12` until NNN where he quit the system via `NNN_Gt_3`.

To follow the trajectory of one of these trains,

```
$ ./02_Run_simulation.sh v=Train_proto/L10/7 | grep _____
```

which produces the terminal output:

```
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'DEL_Bn_L10_1'  t=18700.0=05:11:40.000  New container: ________________ -> DEL_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_DEL_JVJ'  t=18724.0=05:12:04.000  New container: ________________ -> Trk_L10_1_DEL_JVJ
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'JVJ_Bn_L10_1'  t=18753.705323193917=05:12:33.705  New container: ________________ -> JVJ_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_JVJ_CVC'  t=18777.705323193917=05:12:57.705  New container: ________________ -> Trk_L10_1_JVJ_CVC
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'CVC_Bn_L10_1'  t=18949.140684410646=05:15:49.140  New container: ________________ -> CVC_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_CVC_AEA'  t=18973.140684410646=05:16:13.140  New container: ________________ -> Trk_L10_1_CVC_AEA
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'AEA_Bn_L10_1'  t=19016.58174904943=05:16:56.581  New container: ________________ -> AEA_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_AEA_CJC'  t=19040.58174904943=05:17:20.581  New container: ________________ -> Trk_L10_1_AEA_CJC
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'CJC_Bn_L10_1'  t=19123.614068441064=05:18:43.614  New container: ________________ -> CJC_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_CJC_CDC'  t=19147.614068441064=05:19:07.614  New container: ________________ -> Trk_L10_1_CJC_CDC
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'CDC_Bn_L10_1'  t=19213.20342205323=05:20:13.203  New container: ________________ -> CDC_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_CDC_ATÁ'  t=19237.20342205323=05:20:37.203  New container: ________________ -> Trk_L10_1_CDC_ATÁ
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'ATÁ_Bn_L10_1'  t=19276.50950570342=05:21:16.509  New container: ________________ -> ATÁ_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_ATÁ_LAG'  t=19300.50950570342=05:21:40.509  New container: ________________ -> Trk_L10_1_ATÁ_LAG
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'LAG_Bn_L10_1'  t=19387.724334600756=05:23:07.724  New container: ________________ -> LAG_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_LAG_PPP'  t=19411.724334600756=05:23:31.724  New container: ________________ -> Trk_L10_1_LAG_PPP
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'PPP_Bn_L10_1'  t=19479.737642585547=05:24:39.737  New container: ________________ -> PPP_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_PPP_PDE'  t=19503.737642585547=05:25:03.737  New container: ________________ -> Trk_L10_1_PPP_PDE
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'PDE_Bn_L10_1'  t=19560.344106463876=05:26:00.344  New container: ________________ -> PDE_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_PDE_TTR'  t=19584.344106463876=05:26:24.344  New container: ________________ -> Trk_L10_1_PDE_TTR
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'TTR_Bn_L10_1'  t=19608.58365019011=05:26:48.583  New container: ________________ -> TTR_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_TTR_AMN'  t=19632.58365019011=05:27:12.583  New container: ________________ -> Trk_L10_1_TTR_AMN
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'AMN_Bn_L10_1'  t=19652.30798479087=05:27:32.307  New container: ________________ -> AMN_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_AMN_GMG'  t=19676.30798479087=05:27:56.307  New container: ________________ -> Trk_L10_1_AMN_GMG
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'GMG_Bn_L10_1'  t=19735.861216730034=05:28:55.861  New container: ________________ -> GMG_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_GMG_NMN'  t=19759.861216730034=05:29:19.861  New container: ________________ -> Trk_L10_1_GMG_NMN
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'NMN_Bn_L10_1'  t=19803.92015209125=05:30:03.920  New container: ________________ -> NMN_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_NMN_SAA'  t=19827.92015209125=05:30:27.920  New container: ________________ -> Trk_L10_1_NMN_SAA
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'SAA_Bn_L10_1'  t=19856.53231939163=05:30:56.532  New container: ________________ -> SAA_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_SAA_CCU'  t=19880.53231939163=05:31:20.532  New container: ________________ -> Trk_L10_1_SAA_CCU
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'CCU_Bn_L10_1'  t=19912.37642585551=05:31:52.376  New container: ________________ -> CCU_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_CCU_AIA'  t=19936.37642585551=05:32:16.376  New container: ________________ -> Trk_L10_1_CCU_AIA
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'AIA_Bn_L10_1'  t=19980.95817490494=05:33:00.958  New container: ________________ -> AIA_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_AIA_CCH'  t=20004.95817490494=05:33:24.958  New container: ________________ -> Trk_L10_1_AIA_CCH
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'CCH_Bn_L10_1'  t=20041.935361216725=05:34:01.935  New container: ________________ -> CCH_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_CCH_BBE'  t=20065.935361216725=05:34:25.935  New container: ________________ -> Trk_L10_1_CCH_BBE
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'BBE_Bn_L10_1'  t=20111.46768060836=05:35:11.467  New container: ________________ -> BBE_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_BBE_FFU'  t=20135.46768060836=05:35:35.467  New container: ________________ -> Trk_L10_1_BBE_FFU
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'FFU_Bn_L10_1'  t=20218.690114068435=05:36:58.690  New container: ________________ -> FFU_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_FFU_TOT'  t=20242.690114068435=05:37:22.690  New container: ________________ -> Trk_L10_1_FFU_TOT
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'TOT_Bn_L10_1'  t=20277.7186311787=05:37:57.718  New container: ________________ -> TOT_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_TOT_TEM'  t=20301.7186311787=05:38:21.718  New container: ________________ -> Trk_L10_1_TOT_TEM
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'TEM_Bn_L10_1'  t=20320.634980988587=05:38:40.634  New container: ________________ -> TEM_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_TEM_LTL'  t=20344.634980988587=05:39:04.634  New container: ________________ -> Trk_L10_1_TEM_LTL
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'LTL_Bn_L10_1'  t=20457.515209125468=05:40:57.515  New container: ________________ -> LTL_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_LTL_RDL'  t=20481.515209125468=05:41:21.515  New container: ________________ -> Trk_L10_1_LTL_RDL
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'RDL_Bn_L10_1'  t=20527.950570342196=05:42:07.950  New container: ________________ -> RDL_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_RDL_LAA'  t=20551.950570342196=05:42:31.950  New container: ________________ -> Trk_L10_1_RDL_LAA
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'LAA_Bn_L10_1'  t=20619.678707224324=05:43:39.678  New container: ________________ -> LAA_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_LAA_MAE'  t=20643.678707224324=05:44:03.678  New container: ________________ -> Trk_L10_1_LAA_MAE
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'MAE_Bn_L10_1'  t=20743.773764258545=05:45:43.773  New container: ________________ -> MAE_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_MAE_MDL'  t=20767.773764258545=05:46:07.773  New container: ________________ -> Trk_L10_1_MAE_MDL
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'MDL_Bn_L10_1'  t=20816.347908745236=05:46:56.347  New container: ________________ -> MDL_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_MDL_MDF'  t=20840.347908745236=05:47:20.347  New container: ________________ -> Trk_L10_1_MDL_MDF
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'MDF_Bn_L10_1'  t=20904.749049429647=05:48:24.749  New container: ________________ -> MDF_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_MDF_ATL'  t=20928.749049429647=05:48:48.749  New container: ________________ -> Trk_L10_1_MDF_ATL
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'ATL_Bn_L10_1'  t=20980.935361216718=05:49:40.935  New container: ________________ -> ATL_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_ATL_RCR'  t=21004.935361216718=05:50:04.935  New container: ________________ -> Trk_L10_1_ATL_RCR
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'RCR_Bn_L10_1'  t=21056.6939163498=05:50:56.693  New container: ________________ -> RCR_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Trk_L10_1_RCR_HIS'  t=21080.6939163498=05:51:20.693  New container: ________________ -> Trk_L10_1_RCR_HIS
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'HIS_Bn_L10_1'  t=21151.796577946756=05:52:31.796  New container: ________________ -> HIS_Bn_L10_1
zbox: 67342 [M]  'Train_proto/L10/7'  (8,1)  ∊ 'Sink_L10'  t=21175.796577946756=05:52:55.796  New container: ________________ -> Sink_L10
```

The train follows the trajectory expected of a ZBOX following the **Train_L10_1_Line** ZPATH, which from the zsyntax file **_03_Paths.zim** looks like this:

```
# ---- Line L10_1(Hospital Infanta Sofía - Tres Olivos - Puerta del Sur)
[zpath Train_L10_1_Line A=Train n=1 m=Open
 Λ={ [zs φ=<DEL_Bn_L10_1>] [zs φ=<Trk_L10_1_DEL_JVJ>] [zs φ=<JVJ_Bn_L10_1>]
  [zs φ=<Trk_L10_1_JVJ_CVC>] [zs φ=<CVC_Bn_L10_1>] [zs φ=<Trk_L10_1_CVC_AEA>]
  [zs φ=<AEA_Bn_L10_1>] [zs φ=<Trk_L10_1_AEA_CJC>] [zs φ=<CJC_Bn_L10_1>]
  [zs φ=<Trk_L10_1_CJC_CDC>] [zs φ=<CDC_Bn_L10_1>] [zs φ=<Trk_L10_1_CDC_ATÁ>]
  [zs φ=<ATÁ_Bn_L10_1>] [zs φ=<Trk_L10_1_ATÁ_LAG>] [zs φ=<LAG_Bn_L10_1>]
  [zs φ=<Trk_L10_1_LAG_PPP>] [zs φ=<PPP_Bn_L10_1>] [zs φ=<Trk_L10_1_PPP_PDE>]
  [zs φ=<PDE_Bn_L10_1>] [zs φ=<Trk_L10_1_PDE_TTR>] [zs φ=<TTR_Bn_L10_1>]
  [zs φ=<Trk_L10_1_TTR_AMN>] [zs φ=<AMN_Bn_L10_1>] [zs φ=<Trk_L10_1_AMN_GMG>]
  [zs φ=<GMG_Bn_L10_1>] [zs φ=<Trk_L10_1_GMG_NMN>] [zs φ=<NMN_Bn_L10_1>]
  [zs φ=<Trk_L10_1_NMN_SAA>] [zs φ=<SAA_Bn_L10_1>] [zs φ=<Trk_L10_1_SAA_CCU>]
  [zs φ=<CCU_Bn_L10_1>] [zs φ=<Trk_L10_1_CCU_AIA>] [zs φ=<AIA_Bn_L10_1>]
  [zs φ=<Trk_L10_1_AIA_CCH>] [zs φ=<CCH_Bn_L10_1>] [zs φ=<Trk_L10_1_CCH_BBE>]
  [zs φ=<BBE_Bn_L10_1>] [zs φ=<Trk_L10_1_BBE_FFU>] [zs φ=<FFU_Bn_L10_1>]
  [zs φ=<Trk_L10_1_FFU_TOT>] [zs φ=<TOT_Bn_L10_1>] [zs φ=<Trk_L10_1_TOT_TEM>]
  [zs φ=<TEM_Bn_L10_1>] [zs φ=<Trk_L10_1_TEM_LTL>] [zs φ=<LTL_Bn_L10_1>]
  [zs φ=<Trk_L10_1_LTL_RDL>] [zs φ=<RDL_Bn_L10_1>] [zs φ=<Trk_L10_1_RDL_LAA>]
  [zs φ=<LAA_Bn_L10_1>] [zs φ=<Trk_L10_1_LAA_MAE>] [zs φ=<MAE_Bn_L10_1>]
  [zs φ=<Trk_L10_1_MAE_MDL>] [zs φ=<MDL_Bn_L10_1>] [zs φ=<Trk_L10_1_MDL_MDF>]
  [zs φ=<MDF_Bn_L10_1>] [zs φ=<Trk_L10_1_MDF_ATL>] [zs φ=<ATL_Bn_L10_1>]
  [zs φ=<Trk_L10_1_ATL_RCR>] [zs φ=<RCR_Bn_L10_1>] [zs φ=<Trk_L10_1_RCR_HIS>]
  [zs φ=<HIS_Bn_L10_1>] [zs φ=<Sink_L10>]
}]
```

Two points are emphasized here. Firstly, in each of the above runs, the full simulation is indeed being executed; the `-v` option and the `v=...` merely determine levels of verbosity for either the system as a whole or individual ZBOXEN. Secondly, the above output is (as mentioned repeatedly) not intended to be parsed beyond debugging of the system specification. The *reporting* output, which is indeed intended for this purpose, is investigated below.

7.1.6 Simulation output

Upon completion (which is most efficiently accomplished without `-v`), the file `madrid_01.zo` will contain the *reporting* output, every row beginning with `R:`. It is of course much like the example reporting output exhibited in Chapter 5.

```
R: zbox  i=latlon:40.4123454,-3.70466 ztype=Station,1,1 t=18000.0 R=- state=M
  Z.n=19 label=TDM t0=18000.0 x0=-0.0 t1=18000.0 x1=-0.0 δt=0.0 l=1 L=0
  Z.n=19 SpaceInside=-55 Z={ TDM_Cc(2,1) TDM_GtCo_1(8,1) TDM_Gt_1(3,1)
    TDM_Gt_1_out(4,1) TDM_GtCo_2(8,1) TDM_Gt_2(3,1) TDM_Gt_2_out(4,1)
    TDM_GtCo_3(8,1) TDM_Gt_3(3,1) TDM_Gt_3_out(4,1) TDM_GtCo_4(8,1)
    TDM_Gt_4(3,1) TDM_Gt_4_out(4,1) TDM_Bn_L1_0(6,1) TDM_Pf_L1_0(5,1)
    TDM_PfCo_L1_0(8,1) TDM_Bn_L1_1(6,1) TDM_Pf_L1_1(5,1) TDM_PfCo_L1_1(8,1)}
R: zbox ztype=Concourse,2,1 t=18000.0 R=- state=M Z.n=0 label=TDM_Cc
  z.label=TDM z.L=0 z.W=1 t0=18000.0 x0=0.0 t1=18000.0 x1=0.0 δt=0.0 l=1 L=0 z.n=19
R: zbox ztype=Corridor,8,1 t=18000.0 R=- state=M Z.n=0 label=TDM_GtCo_1
  z.label=TDM z.L=0 z.W=1 t0=18000.0 x0=0.0 t1=18000.0 x1=0.0 δt=0.0 l=1 L=110 z.n=19
R: zbox ztype=Gate,3,1 t=18000.0 R=- state=M Z.n=0 label=TDM_Gt_1 z.label=TDM z.L=0
  z.W=1 t0=18000.0 x0=0.0 t1=18000.0 x1=0.0 δt=0.0 l=1 L=0 z.n=19

  . . . (many rows)

R: zbox ztype=Train,9,1 t=38338.9315589354 R=- state=M Z.n=32 label=Train_proto/L6/15
  z.label=MLM_Bn_L6_0 z.L=124 z.W=1 t0=38338.9315589354 x0=0.0 t1=38338.9315589354
  x1=0.0 δt=0.0 l=100 L=1380 Z.n=32 SpaceInside=1348 D =" MLM_Bn_L6_0"
R: zbox ztype=Train,9,1 t=38338.9315589354 R=- state=M Z.n=0 label=Train_proto/L6/57
  z.label=Trk_L6_0_MLM_PPA z.L=749 z.W=1 t0=38338.9315589354 x0=0.0 t1=38341.260456273805
  x1=49.0 δt=2.328897338403042 l=100 L=1380 Z.n=0 SpaceInside=1380 D=""
R: zbox ztype=Train,9,1 t=38338.9315589354 R=- state=D Z.n=32 label=Train_proto/L6/15
  z.label=MLM_Bn_L6_0 z.L=124 z.W=1 t0=38338.9315589354 x0=0.0 t1=38362.9315589354
  x1=24.0 δt=24.0 l=100 L=1380 Z.n=32 SpaceInside=1348 D=""
```

with these reporting rows indicating each ZBOX movement which occured during the simulation. Since the command-line interface has been used, these are written to a file during the simulation; if another interface had been used, they would be available for parsing as they are generated.

The included *TravelTimeTool* tool can used to produce plots and perform comparisons in terms of passenger travel times. Use of this tool to calibrate the passenger walking speed in the system is the subject of the following chapter.

Chapter 8

Calibration

In metro systems where passengers “tap in” and “tap out” using fare cards, tokens or similar means, explicit travel time data may be available for every passenger. In a given system, more or less may indeed be measured; for the purpose of an example calibration as shown here, it is assumed in particular that passenger travel-time data are available.

In order to exhibit a concrete calibration procedure, The Madrid Metro model prepared in the last chapter is used. The scope is restricted to calibration of the parameters of the walking-speed distribution in the system. For simplicity, a single log-normal walking-time distribution is applied to the system as a whole.

In calibrating only the walking-speed distribution, it is implicitly assumed that other aspects of the system are specified correctly; in particular, corridor lengths, train schedules, train speeds and track lengths are assumed to have been finalised.

In order to calibrate the walking-speed distribution, a reference set of data must be supplied which consists of travel times. Since in the present context no such real-world set is available for Madrid, a synthetic set of travel times is generated and used.

8.1 Travel-time tool

The travel-time tool can be run with no parameters

```
$ java -jar TravelTimeTool.jar
```

in order to see documentation and options.

In Fig. 8.1 is shown a distribution of the travel times appearing in the simulated system for trips beginning between 05:00 and 09:00. The variable plotted is the mean travel time for *comparable trips*, which are defined as trips with the same origin and destination, occuring within the same time interval (in this case, the interval size is 24 minutes). To generate the distribution of travel times shown in Fig. 8.1,

```
$ java -jar TravelTimeTool.jar TTD 05:00,09:00,16 0,3600,30 madrid_01.zo
```

In addition to the ‘TTD’ mode, there are modes which generate synthetic reference data (‘SYN’), compare against reference data (‘TTC’), and evaluate an objective function for calibration (‘OBJ’).

These modes are described below; they can be demonstrated in sequence by running

```
$ ./01_Run_ZIM_SYN_TTD_TTC_OBJ.sh
```

8.2 Synthetic reference data

To exhibit how calibration can be performed, a reference set of travel times can be generated, based on a simulation run, with a command like:

```
$ java -jar TravelTimeTool.jar SYN 100 out/madrid_01.zo > _SYN_TravelTimeData.csv
```

The file `_SYN_TravelTimeData.csv` produced will be the reference data, used in here in place of actual ground-truth measurements.

8.3 Comparison

To see the discrepancy between the simulation output and the reference data: `java -jar TravelTimeTool.jar TTC 05:00,09:00,16 -600,600,20 _SYN_TravelTimeData.csv out/madrid_01.zo`

For this example, the time interval between 05:00 and 09:00 is considered. The result is shown in Fig. 8.2.

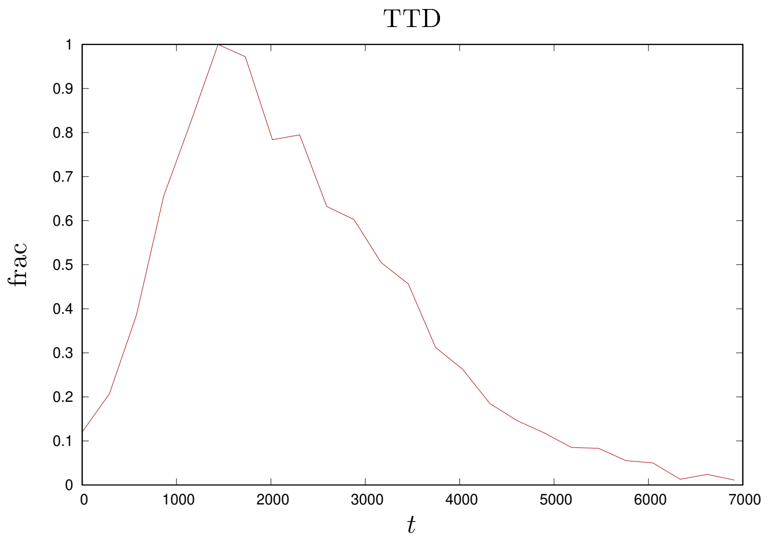

Figure 8.1: Distribution of travel time; $\mu = 2404$ s, $\sigma = 1270$ s.

This shows that the travel times (compared within 'bins' as described before) are 100 seconds faster than

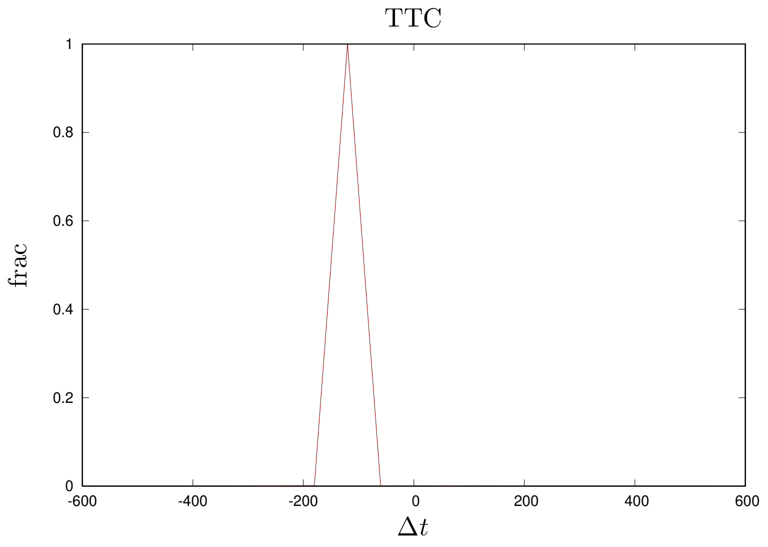

Figure 8.2: Distribution of travel-time error

the reference data. (Fluctuations average out, making this distribution very narrow.) The mean simulation travel-time error is -100 s with a standard deviation of 7.37 s.

8.4 Objective

The parameters of the walking-speed distribution are to be adjusted so that the simulated travel times correspond as closely as possible to the reference data. The objective used for this purpose is

$$\Phi(\mu, \sigma) := \sum_{o,d,\tau} (\tilde{T}_{o,d,\tau} - T_{o,d,\tau})^2$$

where μ and σ are the mean and standard deviation of the log-normal distribution[1] used for walking speeds and specified in the ZSOURCE in `00_StaticTypes.zim`. These quantities are each measured in m/s.

[1]It is noted that they are not the mean and standard deviation of the associated normal distribution; this was described in Sec. 2.2.9

τ labels discrete time intervals during the day; these are statistical bins enabling a definition of *comparable* trips. Two trips are comparable when they share the same origin o, the same destination d and begin during the same interval τ. The specification `05:00,09:00,16` in the above commands indicates that there will be 16 intervals of 15 minutes each; in this case $\tau = 0 \ldots 15$.

$\tilde{T}_{o,d,\tau}$ is the reference travel-time mean for trips from origin o to destination d during time interval τ. $T_{o,d,\tau}$ is the mean generated by simulation.

8.5 Calibration

The calibrated values for μ and ν are those which minimise Φ. The method used here to approximate this minimum is a discrete search algorithm which is analogous to gradient descent;

$$(\mu, \sigma) = \mathrm{LocalMin}(\Phi; \mu_0, \sigma_0)$$

with parameters $E = 1.075$, $R = 0.7$, $s_0 = 1.0e - 8$, $\delta = 0.01$, $i_{max} = 100$ and initial estimate $\mu_0 = 2.0$, $\sigma_0 = 0.7$. $\hat{C}$ is set to constrain the parameters within sane ranges; $1 < \mu < 10$ and $0.05 < \sigma < 1.0$ which are not expected to be saturated.

Each evaluation of $\Phi(\mu, \sigma)$ corresponds to a simulation run; Φ depends on the simulation output and the reference data. The LocalMin algorithm is as follows.

```
algorithm LocalMin($f; \mathbf{x}_0$) to find $\mathbf{x}$ near $\mathbf{x}_0$ which sufficiently
minimises $f(\mathbf{x})$ subject to constraints $\hat{C}$.
    Choose values for constant parameters:
        $E$ — enthusiasm
        $R$ — reluctance
        $\delta$ — perturbation
        $i_{max}$ — maximum iterations
    Set initial values
        $\mathbf{x} \leftarrow \mathbf{x}_0$ — initial guess
        $s \leftarrow s_0$ — descent factor
        $i \leftarrow 0$ — iteration counter
    Loop:
    Evaluate $f$ to numerically define gradient of $f$ at $\mathbf{x}$:
        Evaluate $f_{base} \leftarrow f(\mathbf{x})$ (or use $f_{new}$ if $i > 0$ and last step was enthusiastic)
        for $j$ from 1 to $d$ do
            Evaluate $f_j \leftarrow f(\mathbf{x} + \delta\hat{\mathbf{j}})$
        $\mathbf{g}$ defined by $g_j = (f_j - f_{base})/\delta$ is a numerical gradient of $f$ at $\mathbf{x}$.
    Use $\mathbf{g}$ to search for a better $\mathbf{x}$:
        $\Delta\mathbf{x} \leftarrow s\mathbf{g}$
        Tentative new estimate: $\mathbf{x}' \leftarrow \mathbf{x} + \Delta\mathbf{x}$
        Apply constraints: $\mathbf{x}' \leftarrow \hat{C}\mathbf{x}'$
        if $\mathbf{x}' = \mathbf{x}$ then
            Quit with result $\mathbf{x}$
        Evaluate $f_{new} \leftarrow f(\mathbf{x}')$
        if $f_{new} < f_0$ then
            Update estimate: $\mathbf{x} \leftarrow \mathbf{x}'$
            Exhibit enthusiasm: $s \leftarrow sE$
        else if $f_{new} = f_0$ then
            Quit with result $\mathbf{x}$
        else
            Exhibit reluctance: $s \leftarrow sR$
    $i \leftarrow i + 1$
    Check:
        if $i > i_{max}$ then quit with result $\mathbf{x}$
        if $|\Delta\mathbf{x}| < dx$ then quit with result $\mathbf{x}$
```

The notation is that $\mathbf{x}$ is of dimension d, and $\hat{\mathbf{j}}$ is a unit vector in the j direction, with $j = 1 \ldots d$.

8.6 Running calibration

A script performing the above calibration procedure can be run; the synthetic data generated with the script described above is used.

```
$ ./02_Run_Zim_Minimise_Objective.awk
```

After some computation, the result can be found in the file `_00_StaticTypes.zim`:

$$v_\mu = 1.329, \quad v_\sigma = 0.162.$$

8.7 Calibrated model

Running again the Travel-Time Tool,

```
$ ./03_Run_TTC_post_calib.sh
```

a comparison bewteen the simulator output and the synthetic reference data now appears as shown in Fig. 8.3. The calibration procedure has reduced the mean travel-time error from -100 s to -19 s

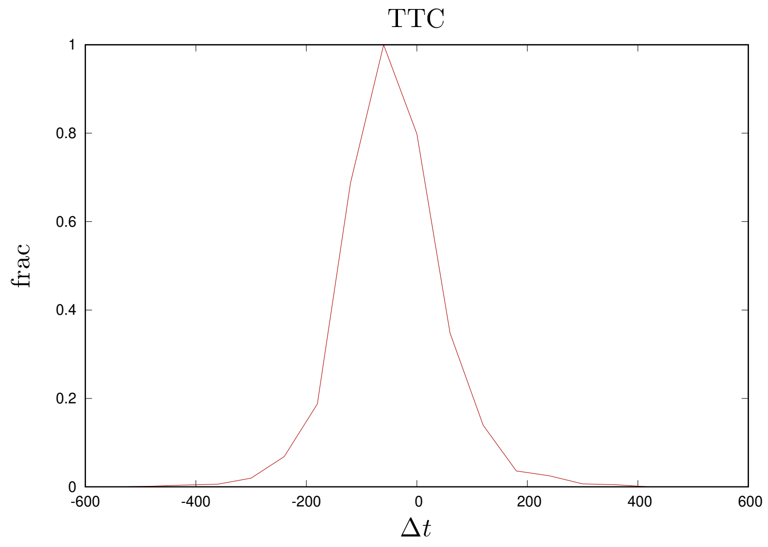

Figure 8.3: Distribution of travel-time error after calibration; $\mu = -15$ s, $\sigma = 90$ s.

The new global distribution of travel times is shown in the following chapter, in Fig. 9.1; it now has a mean of 2333 s and a standard deviation of 998 s.

In Chapter 9 various results will be extracted from the calibrated simulation.

Chapter 9

Simulation results

In this chapter several examples are shown of extraction of results from the model output. The examples are again in the context of the Madrid Metro as in the previous two chapters.

In the following, a time interval will be specified in the same way as was done for calibration in Chapter 8; a starting time and ending time will be specified. This is sufficient for some overall measurements.

In order to perform comparisons, the notion of *comparable* from Sec. 8.4 is used; a number of sub-intervals will be specified, and a quantity of interest will be averaged within each of these sub-intervals. This will be used, for example, in describing the effect of a disruption in Sec. 9.4.

9.1 Travel-time distribution

As before, the travel-time distribution over all travelled origin-destination pairs is taken to be a canonical measure of the performance of the metro system. The distribution could be extracted for a list of any subset of origin stations $\mathcal{O}$, any subset of destination stations $\mathcal{D}$, and any time interval during the day.

For the system as a whole, for the whole day, the distribution is shown in Fig. 9.1

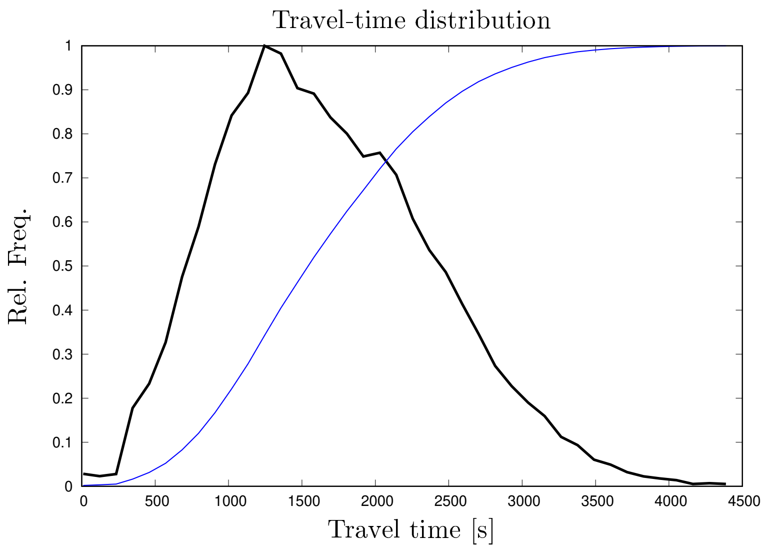

Figure 9.1: Distribution of travel time; $\mu = 2333$ s, $\sigma = 998$ s.

9.2 Platform occupancy

Platform occupancy appears in the simulation report as the number of ZBOXEN located inside a ZBOX of type Platform, whenever entered by a passenger.

A script is provided to extract and collect these data from report lines. To extract the occupancy at a given platform, say `DEL_Pf_L10_1`, as a function of time,

```
$ ./05_FindPlatformOccupancy DEL_Pf_L10_1 out/madrid_08.zo
```

The result is shown in Fig. 9.2 for a portion of the morning. The sawtooth shape is as expected; passengers

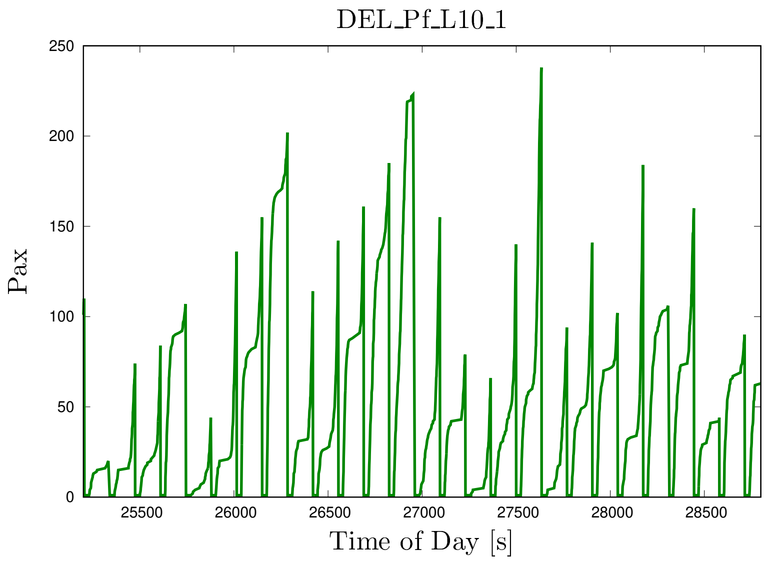

Figure 9.2: Passengers waiting at DEL__Pf__L10__1 from 07:00 to 08:00

arrive at the platform over time, and leave together on a train. The intervals with zero count are the dwelling intervals of a train; passengers do not accumulate on the platform when a train is present.

9.3 Train occupancy

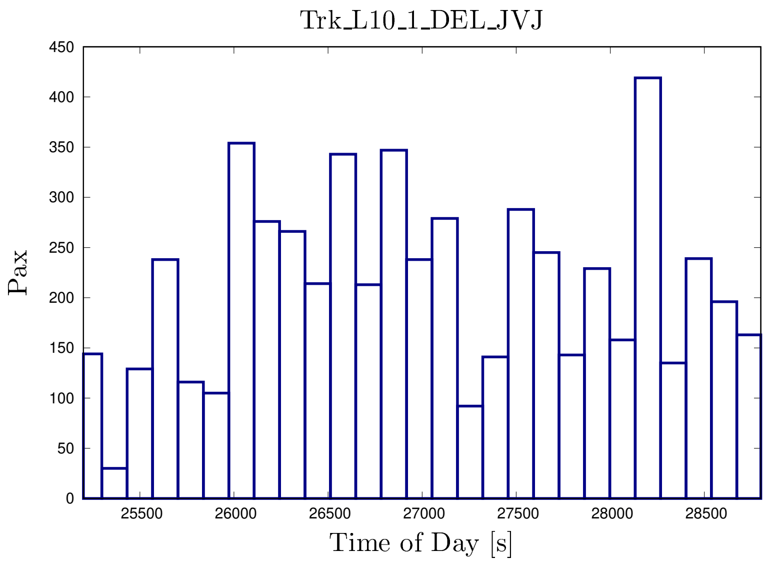

Figure 9.3: Passengers on trains leaving DEL__Bn__L12__0 on Trk__L10__1__DEL__JVJ, during the interval from 07:00 to 08:00

Train occupancy, when leaving a platform (in fact the `Train` zbox leaves a `Bound` zbox), appears in the simulation report as the number of zboxen located inside a zbox of type `Train`, whenever this container enters a zbox of type `Track` (after having left a `Bound`).

To instead extract train occupancy when arriving at a platform, it is necessary to consider entry to the `Bound` zbox rather than the following `Track` zbox.

A script is provided to extract and collect these data from report lines. To extract the occupancy of trains leaving a given platform, say `DEL_Bn_L12_0`, the track segment `Trk_L10_1_DEL_JVJ` is identified:

```
$ ./05_FindTrainOccupancyAt Trk_L10_1_DEL_JVJ out/madrid_08.zo
```
The result is shown in Fig. 9.3.

Similarly, the occupancy of a train during its path through the system can be extracted.
```
$ ./05_FindRunningTrainOccupancy Train_proto/L3/2 out/madrid_08.zo
```
The result is shown in Fig. 9.4.

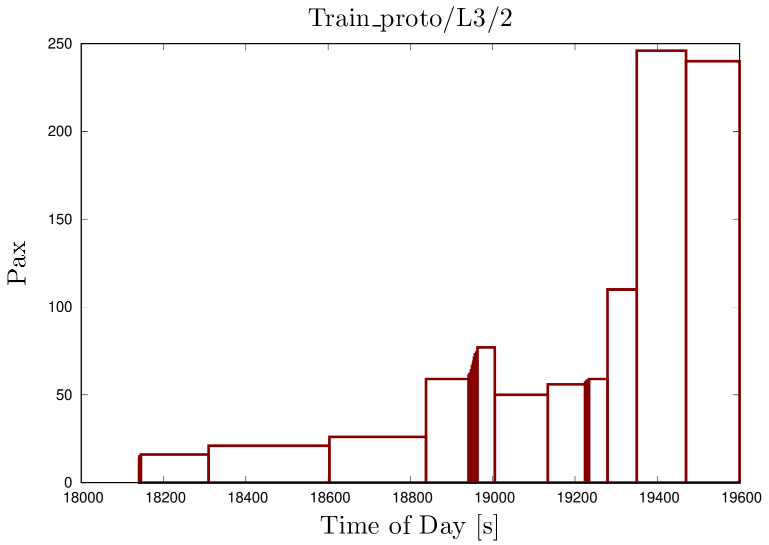

Figure 9.4: Passenger count on single train Train__proto/L3/2

9.4 An event and consequences

One way to simulate a blockage of a track is simply to insert a slow-moving blockage in the desired track ZBOX, using input like the following, which can be found in `ZimInput/08_BlockedTrackIncident.zim`.

```
[zbox Blockage A=Train n=1 v=1.7 ]
[zbox BlockageWaitingRoom A=TrainSource n=1]
[zlink μ=<BlockageWaitingRoom> ν=<Trk_L10_1_DEL_JVJ> A={ Train . 1 } ]
[zlink μ=<Trk_L10_1_DEL_JVJ> ν=<BlockageSink> A={ Train . 1 } ]
[zbox BlockageSink A=TrainSink n=1]
[zschedule Deploy_Blockage
 S = [zsource ϕ=<Blockage> m=One o=Teleport ]
 P = [zpath A=Train n=1 m=Open Λ={ [zstop ϕ=<BlockageWaitingRoom>]
      [zstop ϕ=<Trk_L10_1_DEL_JVJ>] [zstop ϕ=<BlockageSink>] } ]
 T_0=0 T={ 07:30 } ]
```

At 07:30 the blockage is shifted into the track between DEL and JVJ stations. The velocity is specified such that the track is blocked by this ‘Train’ for 6 minutes. The simulation including blockage can be run with `$ ./08_AddIncident_RunSimulation.sh`
which will also produce a TTD analysis file using the Travel-time tool. The travel-time distributions are shown in Fig. 9.5.

Although the effect of the incident is not significant in terms of the travel-time distribution, it can be measured in terms of total travel-time cost, measured in Passenger-seconds. With no incident, for trips beginning within the interval 05:00-10:00, 2180644293 passenger-seconds are reported by the travel-time tool; in the incident case, 2183288493. Therefore, the cost of this delay on the line between DEL and JVJ stations is determined to be $(2183288493 - 2180644293)/3600 = 734$ passenger-hours.[1]

The effect of the incident is most obvious when platform and train occupancies are compared with the base case without incident. In Fig. 9.6 is shown this comparison for the downstream JVJ platform and trains leaving DEL. It is clear that in the incident case, trains which have been held back and arrive more closely in time after the incident is over are underutilised by passengers.

[1]No attempt here is made to ascertain the standard deviation or other properties of this quantity; the present goal is to illustrate extraction of data from simulation output.

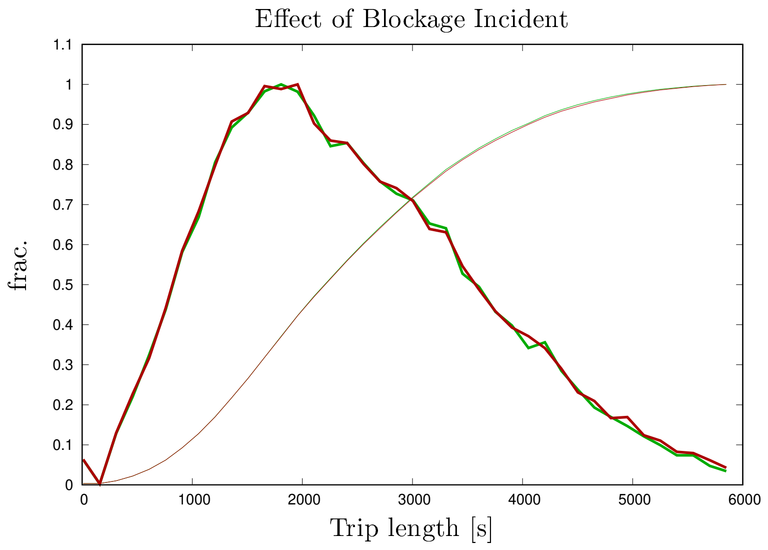

Figure 9.5: Comparison of global travel-time distribution for base case (green) and incident (blockage) case (red): The distribution is for trips originating between 05:00 and 10:00, and the blockage is active from 07:30 to 07:50. The blockage has no measurable effect in terms of this global distribution.

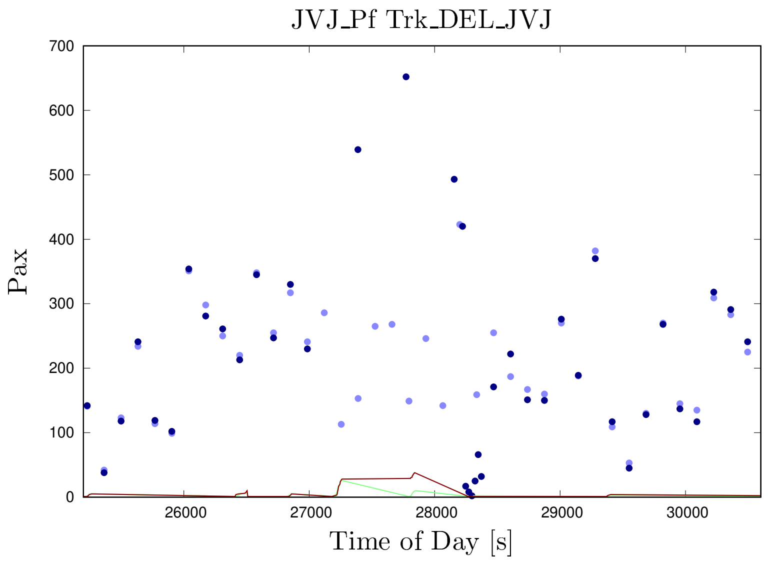

Figure 9.6: Passengers waiting at JVJ__Pf__L10__1 and riding on trains to this platform, from 07:00 to 08:30: The base case platform crowd is plotted in green, the incident case in red. The incident begins at 07:30 = 27000 s. Train occupancy in the base case is plotted in light blue, the incident case in dark blue. It is noted that the results even before the incident are not identical, due to the stochastic sampling of passenger walking speeds.