EURIS, a Specification Method for Distributed Interlockings

Fokko van Dijk[1], Wan Fokkink[2], Gea Kolk[1], Paul van de Ven[1], and Bas van Vlijmen[3]

[1] Holland Railconsult, PO Box 2855, 3500 GW Utrecht, The Netherlands, fjvandijk@hr.nl, gpkolk@hr.nl, phjvandeven@hr.nl
[2] University of Wales Swansea, Singleton Park, Swansea SA2 8PP, Wales, w.j.fokkink@swan.ac.uk
[3] Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands, vlijmen@phil.uu.nl

Abstract. Safety systems for railways have shifted from electronic relays to more computer-oriented approaches. This article highlights the language EURIS from NS Railinfrabeheer, which champions an object-oriented method for the specification of interlocking logics.

1 Introduction

The control and management of a railway system consists of three separate tasks. First, control instructions for the railway yard have to be devised in the logistic layer. Second, control instructions have to be passed on to the infrastructure, which consists of points, signals, sections, level crossings, et cetera. This task is almost always fully automated. Third, it has to be guaranteed that the execution of control instructions does not jeopardise safety; that is, collisions and derailments have to be avoided. This is done by means of an interlocking, which is a medium between the infrastructure and the logistic layer together with its interfaces. An interlocking logic is the embodiment of safety principles and basic rules, according to which a train moves through a railway yard.

Traditionally, an interlocking logic served as a local solution for a specific railway yard, where the logic could be designed to cope with the peculiarities of the railway yard. Modern computer-based railway systems demand a uniform specification method which can be used to formulate interlocking logics for all railway yards. The safety restrictions that are imposed on different railway yards are reasonably consistent, depending mostly on the parameters of autonomous elements such as signals and points. Based on this observation, Middelraad *cum suis* from NS Railinfrabeheer evolved a modular specification method EURIS (EUropean Railway Interlocking Specification) [3], to describe fully automated interlocking logics. EURIS assumes an object-oriented architecture, which consists of a collection of generic building blocks, representing the elements in the infrastructure such as signals and points, and of two clearly separated entities in the outside world, representing the logistic layer and the infrastructure.

The building blocks, which together make up the interlocking logic, communicate with each other by means of data-structures called telegrams. The building blocks can also exchange telegrams with the logistic layer and the infrastructure.

EURIS not only denotes a specification method, it is also the name for a graphically oriented imperative specification language that is based on this method. A Logic and Sequence Chart (LSC) specifies a building block. Each LSC consists of the graphical representation of procedures, which can adapt and test the values of variables, and which can ultimately trigger the transmission of a telegram. Such telegrams can be received by neighbouring building blocks, by the logistic layer, and by the infrastructure. Reversely, each building block can also receive telegrams from neighbouring building blocks, from the logistic layer, and from the infrastructure. The graphical format for LSCs evolved as a compact notation when the Nassi Schneidermann diagrams, which were originally used to express EURIS specifications, became unclear due to deep nestings of if-then-else statements.

A strong advantage of an object-oriented architecture is the possibility to reuse components of a specification. In EURIS, the heart of a specification defines the way that building blocks handle incoming telegrams. When all types of building blocks have been specified in full detail, the specification of a particular railway yard is constructed by simply connecting its separate building blocks in the appropriate manner. A second advantage of the distributed approach is that if the behaviour of say a signal is changed, then this can be taken into account on the level of EURIS by adapting the specification of the corresponding building block. A disadvantage can be that in EURIS applications a procedure such as claiming a route is specified implicitly in the designs of several building blocks, so that adapting such a procedure can become non-trivial.

UniSpec [3, 11] is a particular instance of the EURIS method, which has been developed by NS Railinfrabeheer as a complete set of generic elements to compose interlocking logics for the Dutch railway system. Holland Railconsult has implemented a simulator for UniSpec [9, 10], which enables to animate the behaviour of a UniSpec specification. The simulator is part of a toolset named GUIDE, which is currently used by NS Railinfrabeheer to support both the design and validation of UniSpec specifications. After designing a set of LSCs, the user can join instantiations of these LSCs according to the topology of a railway yard. The result is checked for design rule errors and compiled, after which situations at the railway yard can be simulated via a graphical interface. The simulator enables to locate flaws in an interlocking specification at an early stage of the system engineering process. Furthermore, a simulation session gives a detailed insight of the behaviour of the specification, which can be useful in the communication with customers.

This article presents a description of the EURIS method and its specification language. We describe the features that are present in [3, 7, 9–11], and which have been singled out as being essential by railway experts. There are several interpretations of the semantics of EURIS around, such as the founding article [3], and the implementation of the simulator. In the full version [6] of this article we formulate a semantics for EURIS that, following the simulator, is based on a discrete time model. Bergstra, Fokkink, Mennen, and van Vlijmen [4] presented a more detailed semantics for EURIS in discrete time process algebra [1].

Acknowledgements This research was funded by Holland Railconsult. Daan van der Meij and Peter Middelraad from NS Railinfrabeheer, and Jan Bergstra, Jan Friso Groote, Marco Hollenberg and Wendi Mennen from Utrecht University, are thanked for their support.

2 Overview of EURIS

EURIS is a graphically oriented, parallel, event-driven, weakly typed, imperative specification language. A EURIS specification consists of the graphical description of elements in the form of *LSCs* (see Section 2.3), which are connected by *ports* (see Section 2.2). Each element consists of procedures with an imperative character, called *flows* (see Section 2.1). Elements can send data-structures called *telegrams* (see Section 2.2) to each other via their ports. Reception of a telegram by a element causes the execution of a flow, which is determined by the telegram and the channel via which the element received the telegram. Changing the value of a variable can also cause the execution of a flow (see Section 2.2).

EURIS assumes the two standard data types of Booleans and integers. The Booleans consist of 1, representing *true*, and 0, representing *false*. Three standard functions are defined on the integers: addition, subtraction, and multiplication.

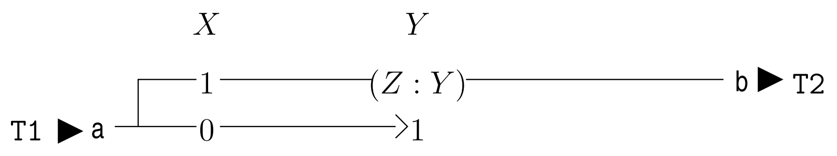
2.1 Flows

We consider a certain element, with a unique element name. A *flow* for this element is a procedure that is built from the following five basic constructs:

- An *execution condition* formulates under which circumstances the flow is executed. There are two possibilities.
  If the execution condition is of the form `T` ► `p`, then the flow is executed if telegram `T` is received at port `p` of the element; see Section 2.2.
  If the execution condition consists of a variable name X, then the flow is executed depending on the (change of) value of X; see Section 2.2.
- A *case* tests the value of a variable; the returned value influences the subsequent execution of the flow.
- An *assignment* adapts the value of a variable.
- A *termination* statement marks the end of the execution of the flow.
- A *send* action `p` ► `T` instructs that telegram `T` is sent out via port `p` of the element. A send action is always followed by termination of the flow.

An element is specified by its flows, where the execution conditions of the flows cover all telegrams that can be received by the ports of the element.

Flows are represented graphically, where the tests and assignments of the flow are connected with each other by continuous lines. A test whether variable X equals value v is denoted either by placing v in the flow below variable X, or by placing the expression $(X = v)$ in the flow. An assignment of value v to variable X is denoted either by placing $>v$ in the flow below variable X, or by placing the expression $(X : v)$ in the flow. The graphical layout of flows is important for the interpretation of tests and assignments; see the picture below.



The execution condition at the left of this flow expresses that it is executed if telegram `T1` is received via port `a`. First, the flow tests the value of the variable X. If this value is 0, then the flow assigns the value 1 to variable Y, after which it terminates. If the value of X is 1, then the flow assigns the value of Y to Z, after which it sends out telegram `T2` via port `b`. Z is a telegram field; if this field does not yet exist then it is created, and otherwise its value is changed.

2.2 Ports, Telegrams, and Variables

A EURIS specification assumes a logistic layer and an infrastructure, and specifies the behaviour of a number of elements. In particular, it is described how these separate entities send messages called telegrams to each other, and how the elements react when they receive a certain telegram from a certain entity. The elements and the logistic layer and the infrastructure can send telegrams to each other via communication channels, which are constructed by the combination of *ports*. EURIS recognises the following two kinds of ports.

– An element has one or more *route* ports. Each route port `p` of an element `e` is linked with exactly one route port $\mathtt{p}'$ of another element $\mathtt{e}'$, establishing a communication channel between `e` and $\mathtt{e}'$. If element `e` sends a telegram into port `p`, then this is received by element $\mathtt{e}'$ through port $\mathtt{p}'$, and vice versa.
– An element may have a port that connects it with the logistic layer, and a port that connects it with the infrastructure. Telegrams can travel from the element to the logistic layer and to the infrastructure via these ports. Vice versa, for each element, the logistic layer and the infrastructure may have a port via which they can send telegrams to this element.

A *telegram* has a unique name and carries a *telegram table*, which assigns Boolean and integer values to telegram variables. A telegram may be passed on between a number of elements, which all update the information in the table of the telegram. If a flow terminates by sending out a telegram, then it attaches the telegram table that was created, or adapted, during the execution of the flow. Telegram variables are only meaningful for an element as long as the flow that belongs to the telegram in the element is being executed.

A *route* telegram is sent from one entity to another entity, where entities can be elements, the logistic layer, or the infrastructure. It consists of a telegram name, a telegram table, and a port name from which it is sent out. An *internal* telegram is generated inside an element by special types of variables, depending on the (change of) value of such a variable. An internal telegram consists of a telegram name together with the name of the variable that produced this telegram, and of an empty telegram table.

EURIS distinguishes several types of internal variables. The initial values of *input* variables, which carry the version symbol '`!`', are latched. The values of internal variables without a version symbol can be adapted without giving rise to the execution of a flow. Finally, variables with a version symbol from {`@`, `&`, `$`, `#`, `?#`, `>>#`} may trigger the execution of a corresponding flow, depending on the (change of) value of such a variable. A detailed description of the behaviour of these variables is given in the full version of this article [6].

Time plays an important role in the specification of an interlocking logic. It enables to model delays; for example, if a train has passed a section, then for safety reasons this section has to be unoccupied for a certain period of time. We assume a discrete time model, in which time progresses in distinct steps called time steps. Methods such as VPI [5, 8] from the General Railway Signal Company and EBS [2] from Siemens, which are used for the implementation of real-life interlockings, and the simulator of EURIS, are based on a discrete time domain. Furthermore, it has been shown in practice that it is technically feasible to synchronise the parallel processes of a EURIS specification on time slices.

2.3 Logic and Sequence Charts

An LSC consists of a number of graphical representations of flows. An LSC takes as basis a list of internal variables, where variables carry their version symbols. Figure 1 presents an example of an LSC. X_1, X_2 and X_3 are internal variables. X_1 is an input variable, which is denoted by the version symbol '`!`'. X_2 is a one-shot variable, which is denoted by the version symbol '`&`'. X_3 does not carry a version symbol. The graphical representations of the flows that make up the LSC are drawn below this list. Thus, we obtain the picture that is displayed in Figure 1. Below each variable name we have drawn an imaginary vertical dashed

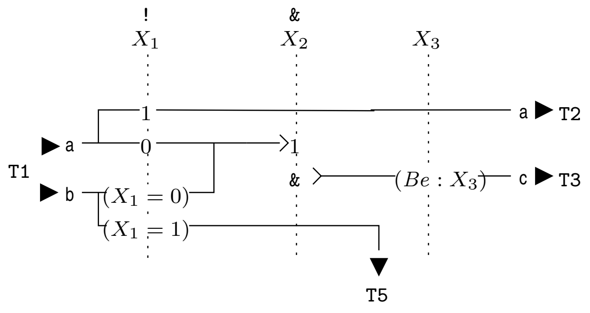
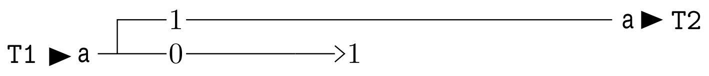
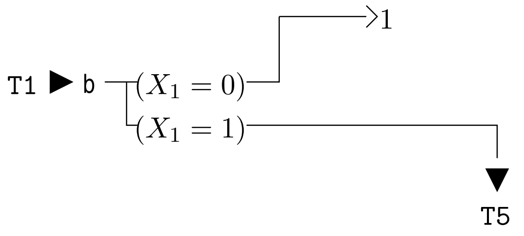


Fig. 1. An LSC Example.

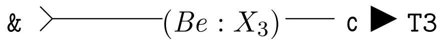
line. Each test and assignment in a flow is placed on such a dashed line, in order to relate it to the variable of which the value is tested or adapted. Some of the flows share the same tail, which means that from some point onwards they have the same functionality. However, flows are independent entities. We explain the meaning of each flow.



The flow above is executed if the telegram with name `T1` is received via port `a`. If the input variable X_1 is 0, then this flow assigns the value 1 to the one-shot variable X_2, after which it terminates. If X_1 is 1, then this flow sends out the telegram with name `T2`, and with the unaltered telegram table, via port `a`.



The flow above is executed if the telegram with name `T1` is received via port `b`. If the input variable X_1 is 0, then this flow assigns the value 1 to the one-shot variable X_2, after which it terminates. If X_1 is 1, then this flow sends out the telegram with name `T5` to the infrastructure.



The flow above, generated by one-shot variable X_2, assigns the value of variable X_3 to variable *Be*. Next, the telegram with name `T3` is sent out via port `c`.

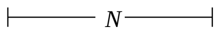
3 UniSpec

UniSpec [3, 11] is a particular instance of the EURIS method that is under development, with the aim to obtain a complete set of generic elements to compose interlocking logics for the Dutch railway system. It transforms the layout of a railway yard into LSCs in three subsequent steps. The topology of the railway yard, and the types of its elements, such as signals and points, are described in a track layout. The elements are provided with ports, which are combined to obtain communication channels between the elements, based on the topology of the track layout. Thus, we obtain the element connection layout. Elements are provided with ports to and from the logistic layer and the infrastructure, and the parameters of the elements and the communication channels are refined, to obtain the logical element connection layout. Finally, this representation is used to obtain the LSCs of the separate elements. These transformation steps have been automated in the simulator [9, 10].

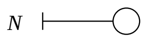
3.1 Track Layout

A track layout of a railway yard is constructed from its separate elements; we distinguish signals, sections, points, crosses, level crossings, and approach monitoring devices for level crossings. Each element in the track layout has a unique name, based on standard (Dutch) railway conventions. We proceed to present the graphical representations of the elements.

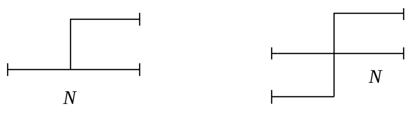
1. A section with name N is represented by



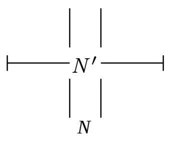
2. A signal with name N is represented by



3. Points and crosses with name N are represented by



4. A level crossing with name N on a section with name N' is represented by



   Each path in a railway yard that leads to a level crossing should encounter an approach monitoring device, which is represented by $\overset{N''}{\blacktriangleright}$. The name N'' always starts with AS, which abbreviates Approach Starting point. An approach monitoring device serves for only one direction, which is determined by the orientation of its graphical representation.

Figure 2 presents an example of a track layout, which is constructed by linking instances of the elements mentioned above.

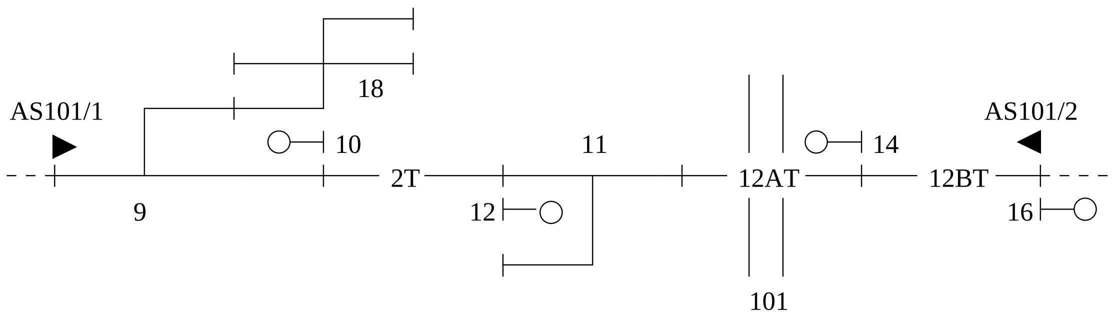
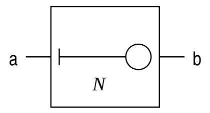


Fig. 2. Example of a Track Layout.

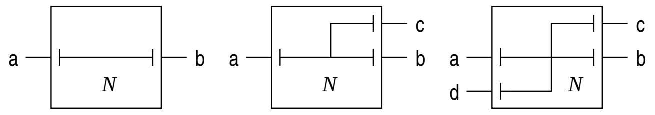
3.2 Element Connection Layout

For each element in a given track layout we generate a graphical element, with a unique name, and with a maximum of four ports called `a`, `b`, `c` en `d`. We discuss these graphical elements.

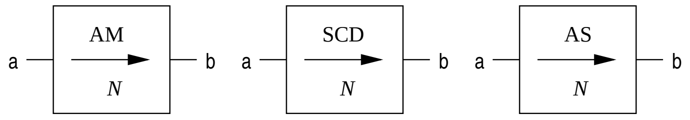
1. A signal *N* that is directed from left to right is represented by



2. Sections, points, and crosses of name *N* are represented by



3. A level crossings is captured implicitly by its monitoring devices: AS (Approach Starting) signals that a train is approaching, while SCD (Signal Clearance Delay) signals that a train has passed the level crossing. AS as well as SCD pass on their information to an element AM (Approach Monitoring). These elements can monitor trains in only one direction, so a level crossing needs two AM elements to monitor approaching trains from opposite directions. The information of the two AM elements is passed on to a central element AMC. Element AMC is represented by a box without ports. Elements AM and AS and SCD of name *N* are represented by



In order to transform a track layout into an element connection layout, all elements are replaced by their boxes. The ports of these boxes are connected to each other, based on the topology of the track layout. Figure 3 depicts how the track layout in Figure 2 is transformed into an element connection layout.

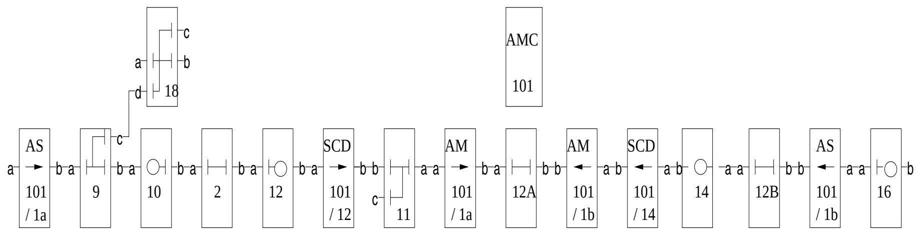


Fig. 3. Example of an Element Connection Layout.

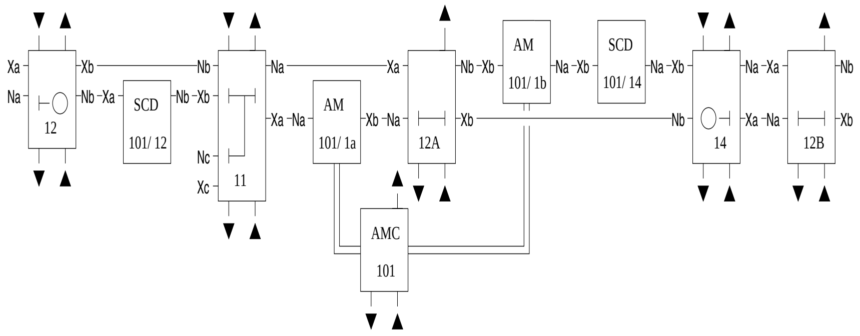


Fig. 4. Example of a Logical Element Connection Layout.

3.3 Logical Element Connection Layout

An element connection layout is transformed into a logical element connection layout as follows. Each pair of linked ports produces one or two directed channels between the elements of the ports. The entry and exit side of a channel are marked 'N' and 'X', respectively. If both ports belong to a bi-directional element (signal, section, points, cross), then two channels are produced, one for each direction. If a port belongs to a uni-directional element (AM, SCD, AS), then one channel is produced in the same direction as this element. An AMC element is connected to each of its AM elements by two channels, one for each direction.

Every element is provided with channels to and from the logistic layer and the infrastructure. Boxes for signals and points are provided with all four channels. Channels from logistic layer to element and from element to infrastructure carry instructions to the signal or points. Channels in the opposite direction carry information on the status of the signal or points. Boxes for sections and crosses are provided with three channels; the channel from logistic layer to element is missing. The channel from element to infrastructure serves to carry instructions on automatic train protection, which automatically halts trains that do not respect the speed limit. AS, SCD, and AM boxes have no channels. AMC boxes have the same three channels as boxes for sections and crosses. The channel from element to infrastructure serves to carry instructions to the level crossing.

An element connection layout is transformed into a logical element connection layout by replacing each element by its corresponding box, and producing directed channels as described above. Furthermore, each box has a private set of internal variables, which are initialised. Figure 4 depicts how part of the element connection layout in Figure 3 is transformed into a logical element connection layout. The initialisation of internal variables is omitted from the picture. Finally, the logical element connection layout is used to produce an EURIS specification. Each type of box is specified by an LSC, which contains the flows that are produced by the box when it receives telegrams. Such specifications are far from trivial; see [3]. Information on channels and the initialisation of variables is not taken into account in LSCs. If a box can be presented in two symmetric ways (as is the case for the box of a signal), then two symmetric LSCs are specified, where each flow from left to right in the one LSC occurs as a flow from right to left in the other LSC, and vice versa.

4 Conclusion

EURIS is a specification method for interlocking logics. The object-oriented approach of EURIS is a strong point in its favour, and its graphical format allows for compact specifications. However, this compactness can obstruct the clarity of EURIS specifications, especially because in practice EURIS is often treated more like a programming language than as a specification language. LARIS [7], a symbolic variant of EURIS, is a first attempt to remedy this imperfection.

UniSpec exemplifies the practical use of EURIS for the specification of interlocking logics. Currently, UniSpec contains special operators to cope with specific situations in Dutch railway yards, which hampers the clarity of its semantics. In future, UniSpec should be polished and reduced to a core set of essential operators with sufficient expressive power, and ideally it should be shown to be a correct implementation of the desired interlocking logic. It is yet unclear how functional requirements such as "if points are occupied then their position should be fixed" can be verified for UniSpec. In future interlocking systems, the topology of railway yards may become more dynamic; e.g., positions of signals may become variable. It remains to be seen whether EURIS can cope with such a shift; the specification of elements in UniSpec is based on a fixed topology of railway yards.

References

1. J.C.M. Baeten en J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8(2):188–208, 1996.
2. J.W.F.M. Beljaars. Prozeß Ablauf Pläne: introduction to a specification method. Report 1992/JBe/3, IB ETS-T&K, 1992. In Dutch.
3. J. Berger, P. Middelraad, and A.J. Smith. EURIS, European railway interlocking specification. UIC, Commission 7A/16, May 1992.
4. J.A. Bergstra, W.J. Fokkink, W.M.T. Mennen, and S.F.M. van Vlijmen. *Railway Logic via EURIS*. Quaestiones Infinitae XXII, Zeno Institute of Philosophy, 1997.
5. W.J. Fokkink. Safety criteria for the vital processor interlocking at Hoorn-Kersenboogerd. In *Proceedings 5th Conference on Computers in Railways (COMPRAIL'96)*, Berlin, pp. 101–110. Computational Mechanics Publications, 1996.
6. F.J. van Dijk, W.J. Fokkink, G.P. Kolk, P.H.J. van de Ven, and S.F.M. van Vlijmen. EURIS, a specification method for distributed interlockings. Technical Report, Department of Computer Science, University of Wales Swansea, 1998.
7. J.F. Groote, M. Hollenberg, and S.F.M. van Vlijmen. LARIS 1.0: language for railway interlocking specification. Report, CWI, Amsterdam, 1998, To appear.
8. J.F. Groote, J.W.C. Koorn, and S.F.M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *Proceedings 10th IEEE Conference on Computer Assurance (COMPASS'95)*, Gaithersburg, pp. 131–150. IEEE, 1995.

9. F. Makkinga. IDEAL, interlocking design and application language guide and reference. Holland Railconsult, 1994.
10. F. Makkinga and F.J. van Dijk. EURIS-simulation tutorial reference. Holland Railconsult, 1995.
11. D. van der Meij and P. Middelraad. UniSpec. NS Railinfrabeheer, 1996.