The Safety Guaranteeing System at Station Hoorn–Kersenboogerd

J.F. Groote, S.F.M. van Vlijmen
Utrecht University, department of Philosophy
Heidelberglaan 8, Utrecht, the Netherlands
E-mail: jfg@phil.ruu.nl, vlijmen@phil.ruu.nl

J.W.C. Koorn
University of Amsterdam, department of Mathematics and Computer Science
Kruislaan 403, Amsterdam, the Netherlands
E-mail: koorn@fwi.uva.nl

Abstract

At the Dutch station Hoorn–Kersenboogerd, computer equipment is used for the safe and in time movement of trains. The computer equipment can be divided in two layers. A top layer offering an interface and means to help a human operator in scheduling train movement. And a bottom layer which checks whether commands issued by the top layer can safely be executed by the rail hardware and which acts appropriately on detection of a hazardous situation. The bottom layer is implemented with a programmable piece of equipment namely a Vital Processor Interlocking[1] (*VPI*). This paper introduces the most important features of the *VPI* at Hoorn–Kersenboogerd. This particular *VPI* is modelled in μCRL. Furthermore, the paper touches upon correctness criteria and tool support for *VPIs*, and suggests ways for verification of properties of *VPIs*. Experiments show that it is indeed possible to efficiently verify these correctness criteria.
1991 Mathematics subject classification: 68Q40, 68Q45.
1990 CR Categories: D.1.3, D.2.1, F.1.2.
Key words & Phrases: formal specification, formal verification, safety critical system, interlocking, railway yard, μCRL, propositional logic, modal logic, ASF+SDF, ASF+SDF Meta-environment.

1 Introduction

Under pressure of modern society railway companies as the Dutch Railway Company (NS) have to operate more flexible and more cost-effective. This is an important reason for the increased use of computer equipment, like the Vital Processor Interlocking (*VPI*), in control systems for railway yards. The *VPI* is a piece of equipment built in the U.S.A. by the General Railway Signal Company; programming for *VPI* is done by NS and other suppliers of NS. The construction of control systems for railway yards is an intricate task at which railway companies have reached a high level of safety and precision. To keep up this high

[1]® *VPI* and Vital Processor Interlocking are registered trademarks of the General Railway Signal Company.

level of quality and to keep costs under control, NS is looking for techniques that can help to design and validate the new computerised control equipment quickly and cheaply. This has motivated us to investigate whether formal methods could be of any help.

Train movement at the Dutch station Hoorn–Kersenboogerd is controlled by electromechanical devices, computers, and human operators. The electromechanical devices perform low level operations, e.g., switching of points, detection of failures and detection of occupation of track sections. The computer equipment serves the human operator to complete his mission: the safe and in time movements of trains. At station Hoorn–Kersenboogerd the computer equipment consists of two layers. The top layer offers a graphical interface and means that assist the human operator to guide trains according to a timetable. The top layer also translates commands of the human operator into commands to the bottom layer. The task of the bottom layer is to check whether these commands can safely be issued to the electromechanical devices, and to act appropriately when a hazardous situation is detected. If a command is considered unsafe it is rejected. If a hazardous situation occurs, then emergency actions are performed, in the worst case all signals are turned red. The bottom layer is implemented with a *VPI*. A *VPI* can be characterised as a Programmable Logic Controller (PLC) that is tuned for use in a railway environment. A *VPI* is an instance of a system that is called an *interlocking* in railway jargon; an interlocking is a system that is responsible for guaranteeing safe operation of a railway yard. The most important software loaded into a *VPI* is called the Vital Logic Code.

In this paper the main features of the *VPI* and Vital Logic Code for the situation at Hoorn–Kersenboogerd are modelled in μCRL [21, 24, 22, 23]. Furthermore, the paper touches upon the issue of safe behaviour of a railway yard. Some approaches that could be followed for the verification of correctness criteria are presented. Tools that could be or have been constructed are discussed. Experiments show that it is indeed possible to efficiently verify these correctness criteria.

Acknowledgements

We wish to thank Stan Nelissen, Gea Kolk, Peter Musters, Robert Straatman, André Klap and Frits Makkinga of NS for introducing us to the enchanting world of railways and especially of *VPIs*.

1.1 Related Work

The literature in which railway yards are discussed can roughly be divided into two categories. One in which railway yards are used as a good illustration of general techniques and ideas, and one in which is focussed specifically on railway related questions.

Examples in the first category are [17], [10], [31], [16] and [30]. These books and papers are mentioned because they highlight a number of important aspects, e.g., specification, correctness criteria, the relation between criteria and implementation, and scheduling optimisation. In [17] trains sets are studied; a complete specification in COLD-1 is presented of a class of railway yards, including trains, rudimentary user interface and interlocking philosophy. Broy also gives a full presentation, but a much simpler system with extra attention to verification of some invariants [10]. In [31] a general framework for the requirements analysis of

safety-critical systems is presented. A distinction is made between *safety requirements* and *mission requirements*. It turns out that the requirements presented in this paper are mostly mission requirements. These specify the situations that should be avoided, whereas safety requirements specify what should happen when something goes wrong after all. M. Feather discusses in [16] how to derive an implementation from a specification, this in contrast to "verification of an implementation after some unknown invention process". In [30] a solution is presented for the Merlin-Randell problem of train journeys.

In the other category of the literature is reported on research into specific systems being designed or used by railway companies and their suppliers. Here it makes sense to further categorise into elementary unit systems and free wired systems. This distinction is presented in [42] where it is argued that extrapolation of the two usual approaches, elementary unit and free wired, do not lead to systems which fully exploit the possibilities offered by programmable electronics; a full treatment of interlocking specification and design is given for implementation by means of programmable electronics.

In elementary unit systems, every element of the track, e.g., signal and point, is an autonomous device, which communicates with neighbouring elements; layout of the railway yard is reflected in the structure of the interlocking. In the free wired case, larger chucks of a railway yard are controlled more or less centrally, the layout of the railway yard is less obvious in the structure. The *VPI* is typically a free wired control system.

To begin with the elementary unit systems, P. Middelraad of NS initiated the development of a language for the specification of these systems. This language is currently is use with NS and also received internationally attention [4, 5]. Formal study of the language is taking place in an academic setting [3, 9, 37]. Siemens is an important supplier of elementary unit based interlockings, among other things, work is done on verification by means of process algebra [18].

A free wired system used in the U.K. by British Rail is the Solid State Interlocking (*SSI*). *SSIs* received considerable attention in the literature [1, 12, 14, 28, 34, 35]. The *SSI* is a more complex machine than the *VPI*, one reason for this is that the *SSI* is used to take more complex scheduling decisions than the *VPI*. Therefore, the more involved methods for verification as presented in these documents were not followed in this paper. But the route using the HOL theorem prover seems attractive also in the *VPI* case [41]. Some Swedish interlockings seem to have been verified using tautology checking in a way that is similar to what is presented in this paper [38]. It is not clear whether formulas in, e.g., the classes $JUST_{\mathcal{MF}}$ and $NEXT_{\mathcal{MF}}$, see Section 4.1, were verified too. Finally, in France a consortium of GEC Alsthom, Matra Transport and CSEE executed a number of projects in the railway sector where formal specification and formal verification were applied [15].

Contents

1 Introduction 1

1.1 Related Work 2

2 Introduction to the *VPI* 4

2.1 Specifying a *VPI* 6

2.2 Vital Logic Code 6

2.3 Static semantics 8

3 Interpretation of *VLC and *VPIs* in μCRL** **9**
3.1 States 9
3.2 Modelling of a program 10
3.3 Temporal behaviour and evaluation of assignments 11
3.4 Transformations on *VPI* programs 13
3.4.1 Elimination of timer assignments 13
3.4.2 Substitution transformation of *VPI* programs 15

4 Verification of Properties of *VPIs* **17**
4.1 What properties 18
4.2 Using Propositional Logic 21

5 Tool support for *VPI* **27**
5.1 Transformation tools 27
5.2 Program slicing 28
5.3 Tautology checkers 31

6 Concluding Remarks and Future Research **32**

1 A verification example **36**

2 μCRL specification of the *VPI* **40**

3 *SDF* specification of *VLC** **46**

2 Introduction to the *VPI*

In this section a short introduction to the *VPI* is given, i.e., its relation to other 'rail equipment', its components and its operation. This section does not provide a detailed description of *VPIs*. Detailed information can be found in [27, 36]; these documents are confidential. However, the presentation here is sufficient for understanding the main operating principles necessary to understand our approach to specification and verification.

In Figure 1, a schematic view of *VPIs* and their environment is presented. The box with the text 'Presentation & Scheduling' is the 'top layer' from the introduction, it presents all equipment for interfacing with traffic control personnel, software for scheduling and routing decisions. At this layer commands are generated to be executed by the track side modules. These commands are all quite simple: 'turn point x to reverse' [2] or 'turn signal A to a colour better than or equal to yellow'. A *VPI* can only make some small decisions, e.g., turn a signal red just after a train has passed or turn A to green (that's better than yellow) if that is considered safe. A *VPI* acts as an intelligent filter between Presentation & Scheduling and the track side modules. In general, one can say that a *VPI* is not aware of the scheduling, routing or planning strategy, nor of the timetable that has to be followed. Its only responsibility is to

[2] In Figure 1, point 1 is *normal* and point 2 is *reverse*

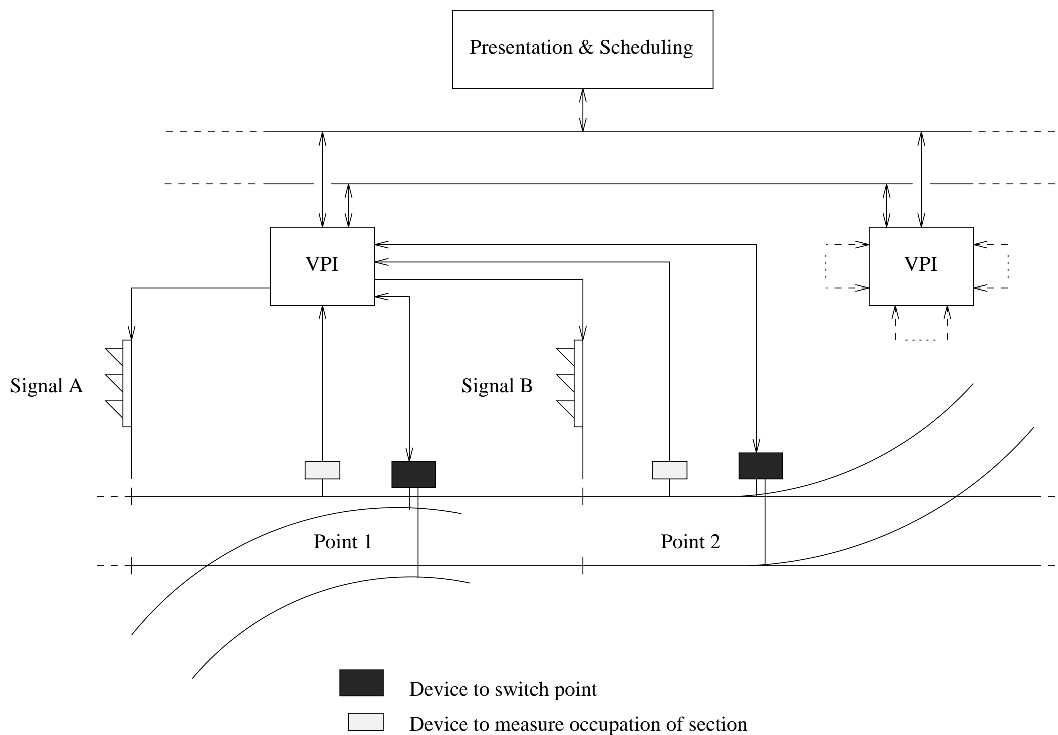

Figure 1: VPIs and their environment

guarantee safety of the railroad. Often multiple *VPIs* are used, each controlling his own set of track side modules. Along a bus *VPIs* receive commands from the Presentation & Scheduling layer and *VPIs* supply the Presentation & Scheduling layer with status information. *VPIs* communicate among themselves on a separate bus.

Operation of a *VPI* is as follows. A *VPI* executes endlessly *control cycles*. In each control cycle a fixed set of inputs is read. The input values are latched, this means that during the rest of the control cycle their value cannot change. Next, the *VPI* sequentially executes a program that takes as arguments the inputs just read, the set of internal variables, and the set of outputs. During execution of the program, internal variables and outputs may (internally) change value, but outputs are transmitted simultaneously to the outside world *after* the program is completely executed. Now some short idle time may follow; a control cycle should take precisely 1 second. Inputs, internal variables, and outputs are essentially all two valued. Typical inputs are commands from the top layer, measurements from the track side modules, and detections of power failure. Typical outputs are status information to the top layer and commands to points and signals.

In the program introduced in the above, control decisions are specified. Here is decided whether a command from the top layer can safely be executed. When a *VPI* considers a command unsafe, the *VPI* just rejects the command. This is not necessarily a hazardous situation, only the top layer is not functioning properly according to the *VPI*. A hazardous situation occurs on power failure, or when something unexpected is measured. In such a situation, emergency actions will be performed, in the worst case all signals are turned red.

$$X_{f,g} = X_{f,g}(\vec{v})$$

$$X_{f,g}(\vec{s}) = \sum_{\vec{\imath} \in \vec{B}_n} read(\vec{\imath}) \cdot send(g(f(\vec{\imath}, \vec{s}))) \cdot X_{f,g}(f(\vec{\imath}, \vec{s}))$$

Figure 2: Abstract specification of a *VPI*.

This may happen when a *VPI* detects malfunctioning of a power supply or detects some internal failure. If after some fixed period of time, 'everything' seems stable again, the *VPI* continuous to regulate.

2.1 Specifying a *VPI*

In the previous section, an informal description of the operation of a *VPI* was given. Furthermore, it was pointed out which features of a *VPI* will be specified in this paper. Most freedom is in the way the calculation that takes place in a control cycle is specified and executed. If we abstract from this, the calculation can be viewed as a function f, from inputs and the joined set of internal variables and outputs to the set of internal variables and outputs. These observations leads to the following abstract model of a *VPI* in process algebra [2].

Let *Bool* be a set with two distinct elements denoted by 0 and 1: $Bool = \{0, 1\}$. Define $\vec{B}_n$, $n \in \mathbb{N}$, as the set of vectors of elements from *Bool* of length n. Define *STEP* as the set of functions $f{:}\vec{B}_n \times \vec{B}_m \to \vec{B}_m$, for $n, m \in Nat$. Define *SHOW* as the set of functions $g{:}\vec{B}_m \to \vec{B}_k$, for $m, k \in Nat, m \geq k$. *STEP* contains 'next step' functions, these functions correspond to *VPI* programs. *SHOW* is the set of functions that filters those elements from a vector that are considered outputs.

Let $f \in STEP$, $f \equiv f : \vec{B}_n \times \vec{B}_m \to \vec{B}_m$, and $g \in SHOW$, $g \equiv g : \vec{B}_m \to \vec{B}_k$. Furthermore, $\vec{s}$ is variable over $\vec{B}_m$, $\vec{v} \in \vec{B}_m$ is an initial state. A *VPI* is now characterised as the process defined by $X_{f,g}$ in Figure 2. This process starts with the initial state $\vec{v}$. It then reads an arbitrary input vector $\vec{\imath}$ from $\vec{B}_n$. Next it sends a selection $g(..)$ of the new state $f(\vec{\imath}, \vec{s})$ to the outside world and finally starts all over again from this new state.

This model seems adequate for reasoning about *VPIs*. A meta assumption is that the time from *read* action to *read* action is precisely 1 second. In section 2.2 and 3 is elaborated on f, its specification and its evaluation. Note that the form of this specification, which seems to be the natural one for a *VPI*, is close to the UNITY format of [11] and the Linear Process Operator of [8]. This is interesting because there is some experience with verification of specifications in this format [7, 11].

2.2 Vital Logic Code

VPI programs are specified in a number of files. The most important part of the program is specified in a language which we will call here Vital Logic Code (*VLC*). In this section some of the syntactical and semantical aspects of *VLC* are introduced. Other program parts specify, e.g., mappings of variable names to hardware ports, names of inputs and output

ports.

An introduction to *VLC* seems appropriate because this may clarify the way the specification in μCRL is set-up. The text of the *VLC* program as operational at Hoorn–Kersenboogerd is not listed in this paper.

To make it easier for us to work with *VPI* programs we decided to change the syntax of *VLC* slightly. There were two reasons for this:

- The layout of *VLC* is important for the correct interpretation. Take for instance the following expression.

  $\texttt{BOOL}\ x = a$
  $* b$

  Here is '$* b$' is a comment whereas this is not the case in

  $\texttt{BOOL}\ x = a$
  $\ \ * b$

  Here x becomes the value of $a * b$ upon execution. This makes it hard to specify tools with purely context free generators like the ASF+SDF Meta-environment [29] in which the notion of 'the beginning of a line' does not exist.

- Except for lists of inputs and outputs, most of the contents of the program parts other than the *VLC* part are not important from a specification/verification point of view.

The syntax of the adapted *VLC* which we will call *VLC** is specified in Appendix 3 using the *SDF* formalism [26]. *VLC** differs in the following respects from *VLC*.

- The comment convention is changed. Comments start with a '%' instead of the '*'.
- A declaration of inputs is added[3]
- A declaration of code system inputs is added[3]
- A declaration of outputs is added.
- *VLC** has constants *TRUE* and *FALSE* whereas *VLC* does not have these constants. The reason to add them is because simplification of expressions may result in one of these constants. E.g., in the one of the *VLC* programs that was studied the following fragment was found

  $SWINIT = .N.OPSTART * .N.VRDFRNT\text{-}DI$
  $OPSTART = SWINIT + OPSTART + VRDFRNT\text{-}DI$

  Following the interpretation of the symbols given in Section 3 this amounts always to *TRUE* as the value that is assigned to *OPSTART* after the first and all following control cycles.

```
DIRECT INPUT SECTION
I
OUTPUT SECTION
U
CODE SYSTEM SECTION
CURRENT RESULT SECTION
R
SELF-LATCHED PARAMETER SECTION
V
TIMER EXPRESSION RESULT SECTION
Q
BOOLEAN EQUATION SECTION
 APPLICATION = Example
  TIME DELAY = 2 SECONDS BOOL Q = I
  BOOL R = Q + V
  BOOL V = Q * .N.R
  BOOL U = V
END BOOLEAN EQUATION SECTION
```

Figure 3: An example VLC^* program.

In the rest of the text we will simply consider *VPI* programs to be expressions in VLC^*.

In Figure 3 an example program is given. There are six variable declaration sections. The most important part of the program is the part below `BOOLEAN EQUATIONS SECTION`. This part essentially consists of a list of assignments like $R = Q + V$ and timer assignments like `TIME DELAY` = 2 `SECONDS BOOL` $Q = I$. Such a list can be cut into sublists by sentences like `APPLICATION` = *Example*; these sentences have no operational meaning. The assignments are executed sequentially and top down. All variables in a *VLC* program are essentially two valued and can be considered propositions in the sense of propositional logic[4]. Expressions at the right of the = sign in Figure 3 can easily be interpreted by reading *not* for $.N.$, *or* for $+$ and *and* for $*$. For example, $Q * .N.R$ is read as Q *and not* R. Timer assignments are slightly more complicated, they delay the assignment of *TRUE* to the variable at the left hand side by the number of seconds specified in the assignment.

2.3 Static semantics

Here a number of static semantic criteria are given. Purpose of this listing is to gain a deeper understanding of the programs in VLC^* and to gain the confidence that the μCRL

[3]Inputs are connected to track side devices, code system inputs are connected to the Presentation & Scheduling layer.

[4]In reality the variables are not two valued. A *VPI* performs run-time checks and for that purpose extra information is stored. A typical check is on variables in the `CURRENT RESULT SECTION`; it is checked whether these are not referenced before being assigned a value. Such a variable has the value *undefined* before it is assigned a value. Furthermore, to handle timer assignments, counters are used.

specification of the *VPI* at Hoorn–Kersenboogerd models the relevant characteristics. Suppose P is a *VLC** program, let IN and OUT, CSS, CRS, $SLPS$ and $TERS$ be sets of the names declared in the `DIRECT INPUT SECTION`, the `OUTPUT SECTION`, the `CODE SYSTEM SECTION`, the `CURRENT RESULT SECTION`, the `SELF-LATCHED PARAMETER SECTION` and the `TIMER EXPRESSION RESULT SECTION` of P respectively.

- The sets IN, OUT, CSS, CRS, $SLPS$ and $TERS$ are mutually disjunct.
- Every variable that occurs in an assignment of P is declared in a variable declaration section of P.
- Every variable in $OUT \cup CRS \cup SLPS \cup TERS$ appears once and only once at the left in an assignment of P.
- Variables in $IN \cup CSS$ are not assigned to.
- A variable declared in the CRS is assigned a value before it is referenced.

As a result there are as many assignments as there are variables in $OUT \cup CRS \cup SLPS \cup TERS$. Outputs are latched and it is allowed to refer to the value of outputs. Therefore one could view them as externally visible self-latched variables.

3 Interpretation of *VLC** and *VPIs* in μCRL

Here, the semantical and operational semantical aspects of *VLC** and *VPIs* are studied in the context of μCRL.

In Section 2.1 an abstract specification of a *VPI* was given. We will now look closely at the state of a *VPI*, the modelling of function f and the evaluation of f given some input. A central notion in *VLC** is the variable. Both in the μCRL specification and in propositional logic, objects exist that correspond to variables. These objects are called identifiers and atomic propositions respectively.

Notation 3.1. Let θ be a type (sort) declared in the μCRL specification in appendix 2. By $x \in \theta$ is denoted that x is a closed term of type θ. Furthermore, given $x, y \in \theta$; with $x = y$ is denoted that x and y are provably equal in the specification.

3.1 States

When a *VPI* starts executing its first control cycle after a reboot, all variables are initially set to F. However, in the specification no specific values for identifiers in initial states are specified or requested. The state, i.e., the information that is conveyed from one control cycle to the other, can, in general, be modelled with a truth valuation for the variables, i.e., with a mapping from variables to the set $\{T, F\}$ (see Section 3.4). In the state also temporal information is kept that is necessary for the proper evaluation of the timer assignments like `TIMEDELAY` $= 2$ `SECONDS BOOL` $Q = I$. In simple assignments like $V = Q * .N.R$, V is assigned the truth value calculated for $Q * .N.R$. A timer assignment delays the assignment of T to the variable at the left side. In case of the assignment `TIME DELAY` $= 2$ `SECONDS BOOL` $Q = I$, the

following holds, if I evaluates to F then Q becomes F, and if I evaluates to T then Q becomes T iff I evaluated to T the last 2 control cycles too. To model this information, one needs counters. In the specification the state is therefore modelled with a mapping of identifiers to $\mathbb{Z}$. These mappings are specified by type *Valuation*, terms in *Valuation* are constructed from the empty list *emptyvaluation* and a function that adds an *Identifier* and an *Integer* to a valuation.

$$\begin{array}{lll} \texttt{func} \; \textit{emptyvaluation} & : & \to \textit{Valuation} \\ \phantom{\texttt{func}} \; \textit{add} & : \textit{Identifier}\#\textit{Integer}\#\textit{Valuation} & \to \textit{Valuation} \end{array}$$

An example state for the program P in Figure 3 might be

$$add(Q, int(0nat, S(S(0nat))), add(V, int(S(0nat), 0nat), emptyvaluation)).$$

Note that Q and V are the only relevant identifiers to keep from control cycle to control cycle; Q and V are the only identifiers that are referenced before being assigned a value. The naturals are specified with zero $0nat$, and successor function S, i.e., $S(x)$ is the natural that corresponds with x plus 1. For example, $S(0nat)$ is 1. The integer numbers are specified as a pair of natural numbers; $int(x, y)$ is read as $x - y$. For example, $int(0nat, 0nat)$ is 0, $int(0, S(S(0nat)))$ is -2. In the rest of the paper the usual numeral notation for integers and naturals is used most of the time. The state above is then, somewhat simpler, written as

$$add(Q, -2, add(V, 1, emptyvaluation)).$$

The intuition behind the integers in a valuation is as follows. A value greater than or equal to 1 signals that the corresponding identifier is T, a number less than or equal to 0 signals that the corresponding identifier is F. What we gain is a combination of a counter and a truth value in one. For example, the value assigned to Q in the above is -2. This signals that in the last control cycle I was 0 (F) and therefore the counter of Q was reset to 0 minus the delay of 2 control cycles. Suppose that the input I stays T for some time, then one will see the following states pass by:

$$\begin{array}{l} add(Q, -1, add(V, 0, emptyvaluation)) \\ add(Q, 0, add(V, 0, emptyvaluation)) \\ add(Q, 1, add(V, 0, emptyvaluation)) \\ add(Q, 1, add(V, 0, emptyvaluation)) \\ \dots \end{array}$$

Note that the 'truth' of I is delayed to have effect on Q.

3.2 Modelling of a program

In this section the focus is on the modelling of a program in μCRL. A first step is a simplification of *VLC**. In the μCRL specification no variable declaration sections are specified. They are not needed to perform the static checks needed in the μCRL specification. Furthermore, partitioning of assignments in applications is not modelled. The program in Figure 3 in μCRL is depicted in Figure 4. It amounts to a list of assignments.

$$
\begin{aligned}
P \equiv\ & add(\ ta(2, Q, ex(I)),\\
& add(\ a(R, or(ex(Q), ex(V))),\\
& add(\ a(V, and(ex(Q), not(ex(R)))),\\
& add(\ a(U, ex(V)), emptyprogram))))
\end{aligned}
$$

Figure 4: The example of Figure 3 in μCRL.

The term in Figure 4 is of type *Program*.

`func`	*emptyprogram*	:	$\rightarrow$ *Program*
	add	: *Assignment#Program*	$\rightarrow$ *Program*

Programs are constructed again as lists, in the same as way terms of *Valuation* were constructed.

The elements of a *Program* are objects of type *Assignment*. A term of type *Assignment* is either a simple assignment or a timer assignment.

`func`	*a*	: *Identifier#Expression*	$\rightarrow$ *Assignment*
	ta	: *Natural#Identifier#Expression*	$\rightarrow$ *Assignment*

Examples of both, taken from Figure 4, are $a(R, or(ex(Q), ex(V)))$ and $ta(2, Q, ex(I))$, these terms represent the *VLC** terms `BOOL` $R = Q + V$ and `TIMEDELAY` = 2 `SECONDS` `BOOL` $Q = I$. Finally type *Expression* is introduced.

`func`	*e*	: *Identifier*	$\rightarrow$ *Expression*
	and, *or*	: *Expression#Expression*	$\rightarrow$ *Expression*
	not	: *Expression*	$\rightarrow$ *Expression*

Atomic expressions are constructed by injecting terms of type *Identifier* into *Expression* by means of the function ex. For example, $ex(Q)$ is an atomic expression. A more complex term is $and(ex(Q), not(ex(R)))$, this term represents the *VLC** term $Q * .N.R$. Note that Q and R should be declared as constants of type *Identifier*. As these constants vary from *VPI* application to *VPI* application they are not declared in the specification in Appendix 2; in the specification, identifiers are simply generated from the natural numbers.

3.3 Temporal behaviour and evaluation of assignments

Recall the description, in Section 2, of the actions that take place during the control cycle of a *VPI*. An abstract view on this activity was specified in Figure 2. The μCRL specification does not differ much. It essentially reads input, $read(inp)$, calculates a new state $evaluate(p, add(inp, v))$, makes a part of this resulting state, which is called the visible part, available to the outside world, $send(normalize(select(\ldots)))$, and start all over again $VPI(..)$. Added are a number of tests on the consistency of the program p, the input inp and the state v. The notation $P_1 \lhd T \rhd P_2$ is read 'if T holds then continue as process P_1 otherwise as

process P_2.

$$VPI(p : Program, show : Identifiers, v : Valuation) =$$

$$\sum_{inp \in Valuation} (\begin{array}{l} read(inp) \cdot \\ send(normalize(select(show, evaluate(p, add(inp, v))))) \cdot \\ VPI(p, show, select(keep(p), evaluate(p, add(inp, v)))) \\ \lhd \\ and(singleassignment(p), and(eq(inputs(p), ids(inp)), eq(keep(p), ids(v)))) \\ \rhd \\ \delta) \end{array}$$

In the *VPI* the assignments of a program are evaluated one after the other in a top down fashion. Given a state, the evaluation of an assignment results in a next state that forms input for the evaluation of the next assignment in the program. In μCRL this evaluation works similarly. There are a number of functions involved, the most important are listed below.

$$\begin{array}{llll} \texttt{func} & evaluate & : Program \# Valuation & \rightarrow Valuation \\ & evaluate & : Assignment \# Valuation & \rightarrow Valuation \\ & evaluate & : Expression \# Valuation & \rightarrow Integer \end{array}$$

For example, the term $evaluate(add(a_1, add(a_2, add(a_3, emptyprogram))), v)$ where a_1 to a_3 are terms of type *Assignment* and v a term of type *Valuation*, reduces by evaluation of a_1 on v, and next, the evaluation of a_2 on the result of this etc. Given the axioms in the specification we see that $evaluate(add(a_1, add(a_2, add(a_3, emptyprogram))), v)$ is equal to $evaluate(a_3, evaluate(a_2, evaluate(a_1, v)))$. Thus the evaluation of a list of assignments on a valuation is equal to the nested evaluation of the single assignments in the list were the first assignment is deepest in the nesting.

The evaluation of an assignment a given a valuation v can easily be explained by looking at the axioms

$$\begin{array}{lcl} evaluate(a(id, e), v) & = & add(id, evaluate(e, v), v) \\ evaluate(ta(i, id, e), v) & = & if(\; eq(evaluate(e, v), 1), \\ & & \quad add(id, up(retrieve(id, v)), v), \\ & & \quad add(id, int(0nat, n), v)). \end{array}$$

Adding an identifier and its value to a valuation simply overrules any value that was previously assigned to the identifier by the valuation. This is expressed by the following axiom, which also expresses a restricted form of commutation.

$$add(id1, i1, add(id2, i2, v2)) = if(eq(id1, id2), add(id1, i1, v2), add(id2, i2, add(id1, i1, v2))).$$

Finally, we will discuss the evaluation of terms of type *Expression*. There is really nothing special about this, essentially it is the evaluation of a propositional expression given a valuation. The only complication is that we have chosen to use the functions *not*, *and*, and

or that were already specified as operations of type *Bool* to do the work. For that purpose two functions were added *Bool2Int* and *Int2Bool*, which obey the following equalities.

$Bool2Int(T) = 1$
$Bool2Int(F) = 0$
$Int2Bool(int(S(x), 0nat)) = T$
$Int2Bool(int(0nat, x)) = F$

We end this section with some examples.

$Bool2Int(and(Int2Bool(-2), T)) = 0$
$evaluate(ex(X), emptyvaluation) = 0$
$evaluate(ex(X), add(X, 1, emptyvaluation)) = 1$
$evaluate(ta(1, X, not(ex(X))), add(X, 1, emptyvaluation)) = add(X, -1, emptyvaluation)$

3.4 Transformations on *VPI* programs

Two transformations are studied: elimination of timer assignments and transformation by substituting variables by their expression.

3.4.1 Elimination of timer assignments

Timer assignments are syntactic sugar, i.e., timer assignments can in principle be translated in other semantically equivalent VLC^* code. Suppose we have a timer assignment

`TIME DELAY` $= n$ `SECONDS` $x = \phi$

Here x is a variable, ϕ is some expression and n is a natural number. The interpretation is the following. If $n = 0$, then x is set to the value of ϕ. If $n > 0$, then x is set to *true* iff ϕ is *true* and ϕ was *true* the n control cycles before the current control cycle. In other words, if $n = 0$, then the timer assignment is equivalent to the assignment

$x = \phi$

If $n > 0$ and $x \notin \alpha(\phi)$, i.e., x does not occur in ϕ, then the expression is equivalent to the two assignments

$x = (x + x') * \phi$
`TIME DELAY` $= n - 1$ `SECONDS` $x' = \phi$

The disjunction $(x + x')$ expresses that x' is only relevant when x is F. When $x \in \alpha(\phi)$, things get a little more complicated.

$x'' = \phi$
$x = (x + x') * x''$
`TIME DELAY` $= n - 1$ `SECONDS` $x' = x''$

Here x'' and x' are fresh and non-identical identifiers in the `CURRENT RESULT SECTION` resp. `TIMER EXPRESSION RESULT SECTION`. Note that, in either case, the order in which assignments are evaluated, i.e., sequentially from top to bottom, is crucial. Of course, in practice,

this 'enrollment' is cumbersome for large n. Now we will reformulate the removal of timer assignments formally.

Two claims are presented on *VPIs* in the context of the μCRL specification; i, i', ... ; id, id', ... ; ids, ids', ... ; e, e', ... ; a_1, a_2, ... are closed terms of type *Integer*; *Identifier*; *Identifiers*; *Expression* and *Assignment* respectively.

The μCRL process *VPI* can turn into δ when fed with an improper valuation. In order to get some control on what is fed to *VPIs*, the following set of proper valuations is defined.

Definition 3.2. Define the function $delay : Identifier \times Program \rightarrow \{.., -1, 0, 1\}$ as follows.

$$delay(id, P) = \begin{cases} -n & \text{if } \texttt{TIME DELAY} = n \texttt{ SECONDS } id = e \text{ is in } P, \\ 0 & otherwise. \end{cases}$$

Let $P \in Program$; the set of proper valuations of P is defined by:

$$PV(P) \stackrel{\text{def}}{=} \{v \in Valuation \mid \ ids(v) = keep(P) \text{ and } \forall id \in Identifier \\ in(id, ids(v)) = T \rightarrow delay(id, P) \leq evaluate(id, v) \leq 1\}.$$

In the following definition, the notion of strong bisimilarity between processes is used. Two processes X and Y are strongly bisimilar, denoted $X \leftrightarrow Y$, if there is a relation between the states of X and Y such that in related states X and Y can perform the same actions and these actions lead again to related states.

Definition 3.3. Given two programs P, $P' \in Program$, and a set of identifiers $show \in Identifiers$ such that $in(show, outputs(P)) = T$ and $in(show, outputs(P')) = T$. P and P' are called *VPI-operationally equivalent over show* iff
$\forall v \in PV(P) \exists v' \in PV(P'). VPI(P, show, v) \leftrightarrow VPI(P', show, v')$ and
$\forall v' \in PV(P') \exists v \in PV(P). VPI(P', show, v') \leftrightarrow VPI(P, show, v)$.

Proposition 3.4.
Let $P \equiv add(a_1, \ldots add(ta(n, id, e), \ldots add(a_m, emptyprogram) \ldots) \ldots) \in Program$, $m \geq 1$.

- *If* $n = 0$, *then* $P \equiv add(a_1, \ldots add(a(id, e), \ldots add(a_m, emptyprogram) \ldots) \ldots)$
- *If* $n > 0$ *and* $in(id, ids(e)) = F$, *then let*
  $$P' \equiv \ add(a_1, \ldots \\ add(a(id, and(or(ex(id), ex(id')), e)), \\ add(ta(n-1, id', e), \ldots \\ add(a_m, emptyprogram) \ldots)) \ldots),$$
  where id' *is a fresh identifier, i.e.,* $in(id', ids(P)) = F$.
- *If* $n > 0$ *and* $in(id, ids(e)) = T$, *then let*
  $$P' \equiv \ add(a_1, \ldots \\ add(a(id'', e), \\ add(a(id, and(or(ex(id), ex(id')), ex(id''))), \\ add(ta(n-1, id', ex(id'')), \ldots \\ add(a_m, emptyprogram) \ldots))) \ldots),$$
  where id' *and* id'' *are fresh and non-identical identifiers, i.e.,* $in(id', ids(P)) = F$, $in(id'', ids(P)) = F$ *and* $eq(id', id'') = F$.

Then, P and P' are VPI-operationally equivalent over outputs(P).

Definition 3.5. Let $P \in$ *Program*; P is called *timer free* iff P does not contain a timer expression, i.e., P does not contain a subterm of the form $ta(n, id, e)$; P is called *single assignment* iff $singleassignment(P) = T$.

$$
\begin{aligned}
&add(\ a(Q, and(or(ex(Q), ex(Q1)), ex(I))),\\
&add(\ a(Q1, and(or(ex(Q1), ex(Q2)), ex(I))),\\
&add(\ a(Q2, ex(I)),\\
&add(\ a(R, or(ex(Q), ex(V))),\\
&add(\ a(V, and(ex(Q), not(ex(R)))),\\
&add(\ a(U, ex(V)), emptyprogram))))))
\end{aligned}
$$

Figure 5: A timer free variant for the program in Figure 4.

In Figure 5 a timer free version of the program in Figure 4 is presented. Note that the way new names are generated, here by numbering, is not important as long as generated names are unique and are not used before.

Proposition 3.6. *For all $P \in$ Program there is a $P' \in$ Program such that P' is timer free and P and P' are VPI-operationally equivalent over outputs(P).*

This proposition gives way to verification of properties of *VPI* programs by verification of the properties on the timer free variants of these programs.

3.4.2 Substitution transformation of *VPI* programs

In this section transformations on programs are discussed that we will call *substitution transformations*. Given an assignment $a(Q, e)$ in a program P; the idea is that under certain conditions an occurrence of Q in P can be replaced by e. This gives, say, P', and P and P' are *VPI*-operationally equivalent. Sometimes one can remove $a(Q, e)$ from the resulting program P', giving P'' that is still *VPI*-operationally equivalent to P. In some cases substitution transformations lead to a program that is only equivalent, as far as external behaviour is concerned, when assignments are executed concurrently. This is all exemplified below.

Consider the following program.

$$
\begin{aligned}
&add(a(Q, and(ex(R), ex(W))),\\
&add(a(U, or(ex(Q), ex(I))), emptyprogram))
\end{aligned}
$$

Now replace $ex(Q)$ in the expression of the second assignment by $and(ex(R), ex(W))$. This gives

$$
\begin{aligned}
&add(a(Q, and(ex(R), ex(W))),\\
&add(a(U, or(and(ex(R), ex(W)), ex(I))), emptyprogram)).
\end{aligned}
$$

It is clear that sequential execution of the resulting two assignments would result in the same valuation for every input valuation as the original program, i.e., the two programs are *VPI*-operational equivalent. This way of substitution is formalised in the following claim.

Proposition 3.7. *Given $P \in$ Program, $P \equiv add(a_1, \ldots, add(a_n, emptyprogram)\ldots)$, and given $1 \leq i < j \leq n$ such that a_i is not a timer assignment, i.e., $a_i \equiv a(id, e)$. Furthermore, for each $a_k \equiv a(id', e')$ or $a_k \equiv ta(h, id', e')$, $i < k < j$, it holds that $in(id', ids(e)) = F$, i.e., assignments a_k do not assign to identifiers that occur in e. Then, P and $add(a_1, \ldots add(a_i, \ldots add(replace(id, e, a_j), \ldots add(a_n, emptyprogram)\ldots)\ldots)\ldots)$ are VPI-operationally equivalent.*

Below an example is given in which the conditions specified in Claim 3.7 are not met; $ex(Q)$ cannot be replaced by the expression of the first assignment because the expression of the first assignment contains an identifier, R, that is assigned to in the second assignment:

$$\begin{aligned} P \equiv \quad & add(a(Q, and(ex(R), ex(W))), \\ & add(a(R, not(ex(R))), \\ & add(a(U, or(ex(Q), ex(I))), emptyprogram))). \end{aligned}$$

After substitution for $ex(Q)$ one obtains

$$\begin{aligned} P' \equiv \quad & add(a(Q, and(ex(R), ex(W))), \\ & add(a(R, not(ex(R))), \\ & add(a(U, or(and(ex(R), ex(W)), ex(I))), emptyprogram))). \end{aligned}$$

It is clear that both programs behave differently, consider, for instance, the following input:

$$add(W, 1, add(R, 0, add(I, 0, emptyvaluation))).$$

The value assigned to U on output will not be the same, 0 respectively 1. However, concurrent execution of the assignment of P' gives the same output as P for every input valuation. With concurrent execution it is meant that, given an input, each assignment is evaluated in the context of this input, whereas in a sequential execution an assignment is evaluated in the context that resulted from the execution of the previous assignment. The observation on concurrent execution leads to a substitution transformation that we will call expansion.

Given a timer free program P, the expanded version of P is obtained by applying the following algorithm to P.

1. Take the next (top down) assignment of the program, suppose this is $a(id, e)$.
2. For each identifier x in e, if x is assigned a value in an assignment that occurs above the current assignment then replace x by its expression, otherwise, do nothing.
3. If this was the last assignment, then stop, otherwise continue at 1.

Expanded programs need a concurrent execution mechanism. Expansion of programs is considered not attractive as many megabytes of code are generated. Expansion may be useful to show how an output or an internal variable is affected directly by the inputs. Because of this direct relation, assignments to variables in the `CURRENT RESULT SECTION` can be removed from an expanded program. Furthermore, the expressions of assignments in an expanded program are directly amenable to logic investigation.

4 Verification of Properties of *VPIs*

A model of a system in a formal specification language and the verification means for the model can be viewed as a theory for the system. Whether predictions made by a theory make sense in practice is beforehand unclear. The relation between the system and the theory is informal; only time can tell if the theory is appropriate. This is of course also true for the model presented in this paper in μCRL. As a result, everything introduced below for verification of *VPIs* is intuitionally appropriate but still needs empirical justification.

Given a μCRL specification S of a *VPI* loaded with program P. S can be viewed as a denotation of a transition system $T(P)$.

In [24] a general description is presented of how a μCRL specification generates a transition system. Sometimes we will abuse terminology and speak of the transition system generated by a program without specifying the visible output and initial state.

Suppose there is a generic set of correctness criteria C, i.e., criteria in C are formulated for *VPIs* loaded with an arbitrary VLC^* program. Let $C(P)$ be the instantiation of C for the specific program P. Verification comes down to showing that $C(P)$ holds on the transition system $T(P)$.

Example 4.1. Consider the transition system depicted in Figure 6 of

$$\begin{array}{ll} VPI(& add(a(X, and(or(ex(X), ex(X')), ex(I))), add(a(X', ex(I)), emptyprogram)), \\ & add(X, add(X', add(I, emptyids))), \\ & add(X, 0, add(X', 0, emptyvaluation) \ \ ). \end{array}$$

This is a μCRL process — timer free — based on the VLC^* program with as only assignment

$\texttt{TIME DELAY} = 1\ \texttt{SECONDS}\ X = I.$

The states of the transition diagram are elements from $\{0,1\} \times \{0,1\} \times \{0,1\} \cup \{0,1\} \times \{0,1\}$. Where the first position of a tuple gives the value for X, the second the value for X' and the third, if there is one, the value for I. Note that this is just a shorter but less precise notation for terms in *Valuation*. The same notation is used to shorten the arguments of the actions that figure as the labels; $read(< 1 >)$ means $read(add(I, int(S(0nat), 0nat), emptyvaluation))$ etc. $\square$

To express properties of transition systems, modal logics are very useful. In [25] a modal logic for μCRL is proposed. There are various means to verify properties expressed in a modal logic on a transition system. Some of them are very systematical and other more ad hoc. In Section 4.1 is investigated to some extent what properties might be candidates for verification. Also some examples of modal formulas are given.

A strategy that is often feasible for checking a criterion on a transition system is to have a computer check the criterion on every state that is reachable in zero or more steps from a start state. For the *VPI* this is not feasible, the state space can be simply enormous; the number of states is at least $2^{|v|}$ for a timer free program, where $\mid v \mid$ is the number of variables that occur in a program. The state space can be reduced considerably by *slicing* a program, i.e., taking into account only those states that are 'related' to the criterion, but the number of states can still be very large.

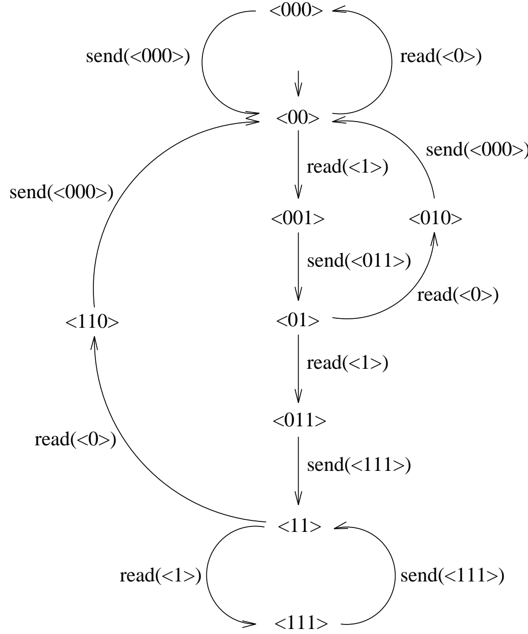

Figure 6: The transistion diagram of the *VPI* in example 4.1.

One means that seems to be of particular importance in the context of *VPIs* is propositional logic. Therefore, from Section 4.2, the paper is dedicated to verification using propositional logic.

4.1 What properties

We have not investigated what the precise properties are that one would like to verify on interlockings and on the *VPI* in particular. In this respect it is noteworthy that the British division of GEC Alsthom executed a thorough hazard analysis for British Rail [14]. From our own superficial study of correctness criteria we have got the impression that correctness criteria will have the following structure: 'After a $send(v)$ action, if $\phi(v)$ holds, then in the near future ψ holds and in the recent past θ held'.

Before we can formalise these criteria in the modal logic of μCRL as defined in [25], a set of formulas is presented called the *local propositions.* These are formulas expressing properties of elements in *Valuation.*

In the definition just below, two notions are used that are taken form [25]: $\mathcal{V}$ is used to denote a set of typed variables, $\mathcal{MF}_{Sig,\mathcal{V}}$ is the set of modal formulas given signature Sig and set of variables $\mathcal{V}$.

Definition 4.2. Let $P \in Program$, let v be a variable of type *Valuation* and $\langle v{:}Valuation\rangle \in \mathcal{V}$. $LP(v,P) \subset \mathcal{MF}_{Sig,\mathcal{V}}$, read *local propositions* , is a set of formulas inductively defined as follows

- if $e \in Expression$ and $in(ids(e), ids(P)) = T$, then $evaluate(e,v) = 0 \in LP(v,P)$ and $evaluate(e,v) = S(0) \in LP(v,P)$;

- if $\phi, \psi \in LP(v, P)$, then $\neg\phi, \phi \vee \psi, \phi \wedge \psi, \phi \rightarrow \psi, \phi \leftrightarrow \psi \in LP(v, P)$.

Definition 4.3. Let $P \in \mathit{Program}$. The notation $\mathcal{J}^k(\phi)$, $k \in \mathbb{N}$, is defined as follows: $\mathcal{J}^0(\phi) \stackrel{\text{def}}{=} \phi$ and $\mathcal{J}^{k+1} \stackrel{\text{def}}{=} \mathcal{J}(\mathcal{J}^k(\phi))$. In the same way $\mathcal{N}^0(\phi) \stackrel{\text{def}}{=} \phi$ and $\mathcal{N}^{k+1} \stackrel{\text{def}}{=} \mathcal{N}(\mathcal{N}^k(\phi))$.

$$
\begin{aligned}
& \mathit{VPI}_{\mathcal{MF}}(P) \stackrel{\text{def}}{=} \\
& \{\Box\forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\phi \rightarrow \psi \wedge \\
& \qquad\qquad \mathcal{J}^{2\cdot i_1+1}(\forall v_1\colon \mathit{Valuation}.\overline{\boxed{send(v_1)}}\psi_1) \wedge \ldots \wedge \\
& \qquad\qquad \mathcal{J}^{2\cdot i_n+1}(\forall v_n\colon \mathit{Valuation}.\overline{\boxed{send(v_n)}}\psi_n) \wedge \\
& \qquad\qquad \mathcal{N}^{2\cdot j_1-1}(\forall w_1\colon \mathit{Valuation}.\overline{\boxed{send(w_1)}}\theta_1) \wedge \ldots \wedge \\
& \qquad\qquad \mathcal{N}^{2\cdot j_m-1}(\forall w_m\colon \mathit{Valuation}.\overline{\boxed{send(w_m)}}\theta_m)) \\
& \mid \phi, \psi \in LP(v, P); n, m \in \mathbb{N}; i_1, \ldots, i_n, j_1, \ldots, j_m \in \mathbb{N} - \{0\}; \\
& \psi_1 \in LP(v_1, P), \ldots, \psi_n \in LP(v_n, P); \theta_1 \in LP(w_1, P), \ldots, \theta_m \in LP(w_m, P)\}
\end{aligned}
$$

One may wonder why *send* actions only occur in formulas in $\mathit{VPI}_{\mathcal{MF}}(P)$, because *read* actions seem to be the proper way to make input valuations available for logical investigations. Yet, this information can also easily be provided in a *send* action, which we need anyway. In that way one can keep the formulas as short as possible.

There are three subsets of formulas of $\mathit{VPI}_{\mathcal{MF}}(P)$ that prove to be useful in the next section, there it is investigated how formulas in $\mathit{VPI}_{\mathcal{MF}}(P)$ can possibly be verified using techniques for proving theorems in propositional logic.

Definition 4.4. Let $P \in \mathit{Program}$.

- $$\mathit{STATIC}_{\mathcal{MF}}(P) \stackrel{\text{def}}{=} \{\Box\forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}\phi \mid \phi \in LP(v, P)\}$$

- $$\begin{aligned} & \mathit{JUST}_{\mathcal{MF}}(P) \stackrel{\text{def}}{=} \\ & \{\Box\forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\phi \rightarrow \mathcal{J}^{2\cdot k+1}(\forall v'\colon \mathit{Valuation}.\overline{\boxed{send(v')}}\psi)) \\ & \mid \phi \in LP(v, P), k \in \mathbb{N} - \{0\}, \psi \in LP(v', P)\} \end{aligned}$$

- $$\begin{aligned} & \mathit{NEXT}_{\mathcal{MF}}(P) \stackrel{\text{def}}{=} \\ & \{\Box\forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\phi \rightarrow \mathcal{N}^{2\cdot k-1}(\forall v'\colon \mathit{Valuation}.\overline{\boxed{send(v')}}\psi)) \\ & \mid \phi \in LP(v, P), k \in \mathbb{N} - \{0\}, \psi \in LP(v', P)\} \end{aligned}$$

Consider the simplified railway yard of Hoorn–Kersenboogerd depicted in Figure 7. Four criteria[5] are presented in an informal way and a formal way using the format of Definition 4.3 and 4.4. Let P be the *VPI* program at the railway yard depicted in Figure 7. In P the variables A_G, A_R, C_G, C_R, S_1, S_2 and LC occur, these are read as follows A shows green, A shows red, C shows green, C shows red, section 1 is free, section 2 is free, and the level crossing is open respectively.

[5]The criteria presented are only caricatures of realistic criteria.

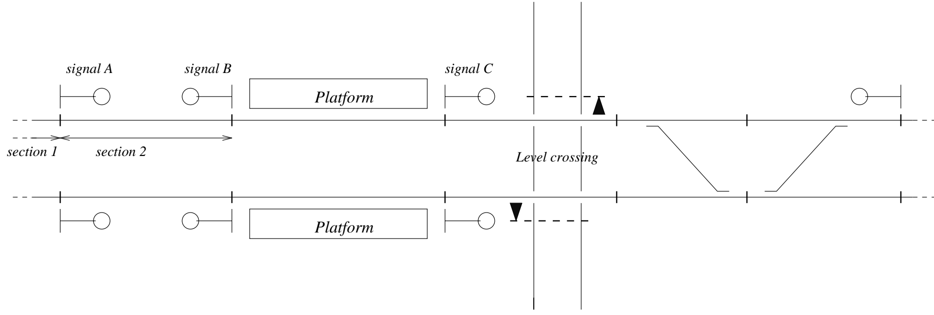

Figure 7: Simplified view on Hoorn–Kersenboogerd.

- Signal A, as any other signal, does not show green and red at the same time.

$$\Box\forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\ \neg(evaluate(A_G, v) = 1 \wedge evaluate(A_R, v) = 1)\ )$$

- If signal A shows green, then signal C does not show red.

$$\Box\forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\ evaluate(A_G, v) = 1 \rightarrow evaluate(C_R, v) = 0\ )$$

- If signal A shows green then section 2 is free and has been free for at least four seconds.

$$\begin{aligned}\Box\forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\quad & evaluate(A_G, v) = 1 \\ & \rightarrow \\ & evaluate(S_2, v) = 1 \wedge \\ & \mathcal{J}^{2\cdot 1+1}(\forall v_1\colon \mathit{Valuation}.\overline{\boxed{send(v_1)}}\, evaluate(S_2, v_1) = 1) \wedge \\ & \mathcal{J}^{2\cdot 2+1}(\forall v_2\colon \mathit{Valuation}.\overline{\boxed{send(v_2)}}\, evaluate(S_2, v_2) = 1) \wedge \\ & \mathcal{J}^{2\cdot 3+1}(\forall v_3\colon \mathit{Valuation}.\overline{\boxed{send(v_3)}}\, evaluate(S_2, v_3) = 1) \wedge \\ & \mathcal{J}^{2\cdot 4+1}(\forall v_4\colon \mathit{Valuation}.\overline{\boxed{send(v_4)}}\, evaluate(S_2, v_4) = 1)\ )\end{aligned}$$

- If section 2 is occupied and signal C shows green, then we can detect that the level crossing is closed in 2 seconds.

$$\begin{aligned}\Box\forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\quad & evaluate(S_2, v) = 0 \wedge evaluate(C_G, v) = 1 \\ & \rightarrow \\ & \mathcal{N}^{2\cdot 2-1}(\forall v'\colon \mathit{Valuation}.\overline{\boxed{send(v')}} evaluate(LC, v') = 0))\end{aligned}$$

Proposition 4.5. *Let $P \in Program$.*

- $(NEXT_{\mathcal{MF}}(P) \cup JUST_{\mathcal{MF}}(P) \cup STATIC_{\mathcal{MF}}(P)) \subset VPI_{\mathcal{MF}}(P)$
- $(NEXT_{\mathcal{MF}}(P) \cap JUST_{\mathcal{MF}}(P) \cap STATIC_{\mathcal{MF}}(P)) = \emptyset$

The following sets will turn out to be useful.

Definition 4.6.

- $LP(v) = \bigcup_{P \in Program} LP(v, P)$
- $VPI_{\mathcal{MF}} = \bigcup_{P \in Program} VPI_{\mathcal{MF}}(P)$
- $STATIC_{\mathcal{MF}} = \bigcup_{P \in Program} STATIC_{\mathcal{MF}}(P)$
- $NEXT_{\mathcal{MF}} = \bigcup_{P \in Program} NEXT_{\mathcal{MF}}(P)$
- $JUST_{\mathcal{MF}} = \bigcup_{P \in Program} JUST_{\mathcal{MF}}(P)$

4.2 Using Propositional Logic

In this section is discussed how propositional logic could be used to verify criteria introduced in Section 4.1. For an introduction to propositional logic the reader is referred to, e.g., [40].

Given a program P and criterion χ; intuitively we have to prove that P implies ϕ. A number of transformations are needed on a program to reach this form of formula from P and χ.

Definition 4.7. Terms in *Expression* are translated to terms in *PROP*, which is the set of all propositions, see [40]. The symbol $\top$ is defined as $\neg\bot$. Suppose e and e' are closed terms of type *Expression*.

$$
\begin{aligned}
&prop(TRUE) = \top\\
&prop(FALSE) = \bot\\
&prop(ex(id)) = id\\
&prop(and(e, e')) = (prop(e) \wedge prop(e'))\\
&prop(or(e, e')) = (prop(e) \vee prop(e'))\\
&prop(.N.e) = \neg(prop(e))
\end{aligned}
$$

In the definitions and claims to come, a name generation convention will be used. The naming convention says the following. Given a program $P \in Program$. Suppose $id \in Identifier$ occurs in P, and we want to replace some occurrences of id by one of the following variants, $id_{\mathcal{J}}$, $id_{\mathcal{J}\mathcal{J}}$, ... ; $id_{\mathcal{N}}$, $id_{\mathcal{N}\mathcal{N}}$, ..., then these variant do not occur in P. Often a renaming ρ is introduced that renames an $id \in Identifier$ in one of the variants above. To denote longer lists of $\mathcal{J}$ and $\mathcal{N}$ subscripts we introduce the following shorthands: $id_{\mathcal{J}^k}$ and $id_{\mathcal{N}^k}$, $k \in \mathbb{N}$. Here $id_{\mathcal{J}^0} \stackrel{\text{def}}{=} id$, $id_{\mathcal{J}^1} \stackrel{\text{def}}{=} id_{\mathcal{J}}$, $id_{\mathcal{J}^2} \stackrel{\text{def}}{=} id_{\mathcal{J}\mathcal{J}}$ etc..

The function *prop* is further overloaded by defining *prop* also on modal formulas. First *prop* is defined on formulas in $LP(v)$. Then *prop* will be defined on $STATIC_{\mathcal{MF}}$, $JUST_{\mathcal{MF}}$ and $NEXT_{\mathcal{MF}}$ too.

Definition 4.8. Let I be the identity on *Identifier*.

- $\chi \in LP(v)$. Let ρ be a renaming.
  if $\chi \equiv evaluate(ex(id), v) = 0$, then $prop_\rho(\chi) \stackrel{\mathrm{def}}{=} \neg\rho(id)$
  if $\chi \equiv evaluate(ex(id), v) = S(0)$, then $prop_\rho(\chi) \stackrel{\mathrm{def}}{=} \rho(id)$
  if $\chi \equiv \neg\phi$, then $prop_\rho(\chi) \stackrel{\mathrm{def}}{=} \neg(prop_\rho(\phi))$
  if $\chi \equiv \phi \vee \psi$, then $prop_\rho(\chi) \stackrel{\mathrm{def}}{=} (prop_\rho(\phi) \vee prop_\rho(\psi))$
  if $\chi \equiv \phi \wedge \psi$, then $prop_\rho(\chi) \stackrel{\mathrm{def}}{=} (prop_\rho(\phi) \wedge prop_\rho(\psi))$
  if $\chi \equiv \phi \rightarrow \psi$, then $prop_\rho(\chi) \stackrel{\mathrm{def}}{=} (prop_\rho(\phi) \rightarrow prop_\rho(\psi))$
  if $\chi \equiv \phi \leftrightarrow \psi$, then $prop_\rho(\chi) \stackrel{\mathrm{def}}{=} (prop_\rho(\phi) \leftrightarrow prop_\rho(\psi))$

- $\chi \equiv \Box\forall v\colon Valuation.\overline{\boxed{send(v)}}\phi \in STATIC_{\mathcal{MF}}$, then $prop(\chi) \stackrel{\mathrm{def}}{=} prop_I(\phi)$.

- $\chi \equiv \Box\forall v\colon Valuation.\overline{\boxed{send(v)}}(\phi \rightarrow \mathcal{J}^{2\cdot k+1}(\forall v'\colon Valuation.\overline{\boxed{send(v')}}\psi)) \in JUST_{\mathcal{MF}}$. Let ρ be such that $\forall id \in Identifier$ holds $\rho(id) \stackrel{\mathrm{def}}{=} id_{\mathcal{J}^k}$; $prop(\chi) \stackrel{\mathrm{def}}{=} prop_I(\phi) \rightarrow prop_\rho(\psi)$.

- $\chi \equiv \Box\forall v\colon Valuation.\overline{\boxed{send(v)}}(\phi \rightarrow \mathcal{N}^{2\cdot k-1}(\forall v'\colon Valuation.\overline{\boxed{send(v')}}\psi)) \in NEXT_{\mathcal{MF}}$. Let ρ be such that $\forall id \in Identifier$ holds $\rho(id) \stackrel{\mathrm{def}}{=} id_{\mathcal{N}^k}$; $prop(\chi) \stackrel{\mathrm{def}}{=} prop_I(\phi) \rightarrow prop_\rho(\psi)$.

Modal formulas refer to current states, and/or past and/or future states. By concatenation and renaming of programs it will be achieved that this notion of past and future can be coded into propositional formulas in a straightforward manner. Below follow two definitions that specify how to concatenate and rename programs. Then, it is presented how to translate programs to propositional logic.

Definition 4.9. First, define the renaming ρ_i for some $i \in \mathbb{Z}$ on *Identifier* as follows

- $\rho_0(id) \stackrel{\mathrm{def}}{=} id$;

- if $i < 0$, then $\rho_i(id) \stackrel{\mathrm{def}}{=} id_{\mathcal{J}^{|i|}}$;

- if $i > 0$, then $\rho_i(id) \stackrel{\mathrm{def}}{=} id_{\mathcal{N}^i}$.

Second, define $timecopy(i, P)$ for some timer free $P \in Program$ and some $i \in \mathbb{Z}$ as follows. If $P \equiv emptyprogram$, then $timecopy(i, P) \stackrel{\mathrm{def}}{=} emptyprogram$. Otherwise, suppose $P \equiv add(a(id_1, e_1), add(a(id_2, e_2), \ldots add(a(id_m, e_m), emptyprogram)\ldots))$, $m \geq 1$, then $timecopy(i, P)$ is the program one obtains by performing the following operation on P.

- replace each occurrence of each identifier id in $inputs(P)$ by $\rho_i(id)$

- for each k, $1 \leq k \leq m$:

  1. replace in $e_1, \ldots, e_k$ each occurrence of id_k by $\rho_{i-1}(id_k)$.
  2. replace $a(id_k, e_k)$ by $a(\rho_i(id_k), e_k)$,
  3. replace in $e_{k+1}, \ldots, e_m$ each occurrence of id_k by $\rho_i(id)$,

Definition 4.10. Define the *n copy just–extension* of a program $P \in \mathit{Program}$ and some $n \in \mathbb{N}$ as the concatenation of $n+1$ timecopies as follows

$$\mathcal{J}^n(P) \stackrel{\text{def}}{=} \mathit{timecopy}(-n, P)\mathit{timecopy}(-n+1, P)\ldots\mathit{timecopy}(0, P).$$

Definition 4.11. Define the *n copy next–extension* of a program $P \in \mathit{Program}$ and some $n \in \mathbb{N}$ as the concatenation of $n+1$ timecopies as follows

$$\mathcal{N}^n(P) \stackrel{\text{def}}{=} \mathit{timecopy}(0, P)\mathit{timecopy}(1, P)\ldots\mathit{timecopy}(n, P).$$

Time for some examples! Suppose P is the program in Figure 5 (page 15). The 0 copy next–extension of P is the same as the 0 copy just–extension of P which is

$$\begin{array}{l}
add(\ a(Q, and(or(ex(Q_{\mathcal{J}}), ex(Q1_{\mathcal{J}})), ex(I))),\\
add(\ a(Q1, and(or(ex(Q1_{\mathcal{J}}), ex(Q2_{\mathcal{J}})), ex(I))),\\
add(\ a(Q2, ex(I)),\\
add(\ a(R, or(ex(Q), ex(V_{\mathcal{J}}))),\\
add(\ a(V, and(ex(Q), not(ex(R)))),\\
add(\ a(U, ex(V)), emptyprogram)))))).
\end{array}$$

The 1 copy just–extension of P is

$$\begin{array}{ll}
add(\ a(Q_{\mathcal{J}}, and(or(ex(Q_{\mathcal{J}\mathcal{J}}), ex(Q1_{\mathcal{J}\mathcal{J}})), ex(I_{\mathcal{J}}))), & \\
add(\ a(Q1_{\mathcal{J}}, and(or(ex(Q1_{\mathcal{J}\mathcal{J}}), ex(Q2_{\mathcal{J}\mathcal{J}})), ex(I_{\mathcal{J}}))), & \longleftarrow\\
add(\ a(Q2_{\mathcal{J}}, ex(I_{\mathcal{J}})), & \\
add(\ a(R_{\mathcal{J}}, or(ex(Q_{\mathcal{J}}), ex(V_{\mathcal{J}\mathcal{J}}))), & \\
add(\ a(V_{\mathcal{J}}, and(ex(Q_{\mathcal{J}}), not(ex(R_{\mathcal{J}})))), & \\
add(\ a(U_{\mathcal{J}}, ex(V_{\mathcal{J}})), & \\
add(\ a(Q, and(or(ex(Q_{\mathcal{J}}), ex(Q1_{\mathcal{J}})), ex(I))), & \leftarrow\\
add(\ a(Q1, and(or(ex(Q1_{\mathcal{J}}), ex(Q2_{\mathcal{J}})), ex(I))), & \\
add(\ a(Q2, ex(I)), & \\
add(\ a(R, or(ex(Q), ex(V_{\mathcal{J}}))), & \\
add(\ a(V, and(ex(Q), not(ex(R)))), & \\
add(\ a(U, ex(V)), emptyprogram)))))))))))). &
\end{array}$$

Note how programs are connected by renaming of identifiers. For example, $Q1_{\mathcal{J}}$, in the assignment marked with $\leftarrow$, refers back to the assignment marked with $\longleftarrow$.

The 1 copy next–extension of P is

$$\begin{array}{l}
add(\ a(Q, and(or(ex(Q_{\mathcal{J}}), ex(Q1_{\mathcal{J}})), ex(I))),\\
add(\ a(Q1, and(or(ex(Q1_{\mathcal{J}}), ex(Q2_{\mathcal{J}})), ex(I))),\\
add(\ a(Q2, ex(I)),\\
add(\ a(R, or(ex(Q), ex(V_{\mathcal{J}}))),\\
add(\ a(V, and(ex(Q), not(ex(R)))),\\
add(\ a(U, ex(V)),\\
add(\ a(Q_{\mathcal{N}}, and(or(ex(Q), ex(Q1)), ex(I_{\mathcal{N}}))),\\
add(\ a(Q1_{\mathcal{N}}, and(or(ex(Q1), ex(Q2)), ex(I_{\mathcal{N}}))),\\
add(\ a(Q2_{\mathcal{N}}, ex(I_{\mathcal{N}})),\\
add(\ a(R_{\mathcal{N}}, or(ex(Q_{\mathcal{N}}), ex(V))),\\
add(\ a(V_{\mathcal{N}}, and(ex(Q_{\mathcal{N}}), not(ex(R_{\mathcal{N}})))),\\
add(\ a(U_{\mathcal{N}}, ex(V_{\mathcal{N}})), emptyprogram)))))))))))).
\end{array}$$

Definition 4.12. Given a timer free program $P \in \mathit{Program}$. The *proposition* of P is a term in PROP defined as follows.

- if $P \equiv \mathit{emptyprogram}$, then $\mathit{prop}(P) \stackrel{\text{def}}{=} \top$.
- if $P \equiv \mathit{add}(a(\mathit{id}, e), \mathit{emptyprogram})$, then $\mathit{prop}(P) \stackrel{\text{def}}{=} \mathit{id} \leftrightarrow \mathit{prop}(e)$.
- if $P \equiv \mathit{add}(a(\mathit{id}, e), p)$, then $\mathit{prop}(P) \stackrel{\text{def}}{=} (\mathit{id} \leftrightarrow \mathit{prop}(e)) \wedge \mathit{prop}(p)$.

Note that the renamings in definition 4.8 and 4.9 were chosen is such a way that atoms in the proposition of a criterion refer to the 'correct' atomic propositions in the proposition of a next/just extented program.

Example 4.13. Let P be the program in Figure 5. Then, $\mathit{prop}(\mathcal{J}^0(P))$ is

$$\begin{array}{l} (Q \leftrightarrow (Q_{\mathcal{J}} \vee Q1_{\mathcal{J}}) \wedge I) \ \wedge \\ (Q1 \leftrightarrow (Q1_{\mathcal{J}} \vee Q2_{\mathcal{J}}) \wedge I) \ \wedge \\ (Q2 \leftrightarrow I) \ \wedge \\ (R \leftrightarrow Q \vee V_{\mathcal{J}}) \ \wedge \\ (V \leftrightarrow Q \wedge R) \ \wedge \\ (U \leftrightarrow V). \end{array}$$

□

Definition 4.14. The *time depth*, td, of a modal formula in $\mathit{JUST}_{\mathcal{MF}}$ or $\mathit{NEXT}_{\mathcal{MF}}$ is the number of control cycles the formula refers to in the past respectively in the future.

$$\begin{array}{l} td(\Box\forall v\colon \mathit{Valuation}.[\overline{\mathit{send}(v)}](\phi \rightarrow \mathcal{J}^{2\cdot k+1}(\forall v'\colon \mathit{Valuation}.[\overline{\mathit{send}(v')}]\psi))) \stackrel{\text{def}}{=} \\ td(\Box\forall v\colon \mathit{Valuation}.[\overline{\mathit{send}(v)}](\phi \rightarrow \mathcal{N}^{2\cdot k-1}(\forall v'\colon \mathit{Valuation}.[\overline{\mathit{send}(v')}]\psi))) \stackrel{\text{def}}{=} \\ k. \end{array}$$

Below will follow the most important claims of this paper. Note that $\mathcal{A}, \mathbb{A}, \zeta, k \models \chi$ holds iff χ holds on every path of $\mathcal{A}$ when starting at the k-th state of such a path (see [25]).

Proposition 4.15. *Given a timer free and single assignment program $P \in \mathit{Program}$. Let $\mathcal{A}(\mathit{VPI}(P, \mathit{ids}(P), v))$ be the transition system generated by $\mathit{VPI}(P, \mathit{ids}(P), v)$ for some $v \in \mathit{Valuation}$. Let $\mathbb{A}$ be a boolean preserving and minimal algebra for the μCRL specification in appendix 2. Let ζ be a substitution. Let $\chi \in \mathit{STATIC}_{\mathcal{MF}}(P)$.*

$$\models \mathit{prop}(\mathcal{J}^0(P)) \rightarrow \mathit{prop}(\chi) \iff \forall v \in \mathit{PV}(P).\mathcal{A}(\mathit{VPI}(P, \mathit{ids}(P), v)), \mathbb{A}, \zeta, 0 \models \chi$$

Proposition 4.16. *Given a timer free and single assignment program $P \in \mathit{Program}$. Let $\mathcal{A}(\mathit{VPI}(P, \mathit{ids}(P), v))$ be the transition system generated by $\mathit{VPI}(P, \mathit{ids}(P), v)$ for some $v \in \mathit{Valuation}$. Let $\mathbb{A}$ be a boolean preserving and minimal algebra for the μCRL specification in appendix 2. Let ζ be a substitution. Let $\chi \in \mathit{NEXT}_{\mathcal{MF}}(P)$.*

$$\models \mathit{prop}(\mathcal{N}^{td(\chi)}(P)) \rightarrow \mathit{prop}(\chi) \iff \forall v \in \mathit{PV}(P).\mathcal{A}(\mathit{VPI}(P, \mathit{ids}(P), v)), \mathbb{A}, \zeta, 0 \models \chi$$

Verification of modal formulas from $JUST_{\mathcal{MF}}$ is not as straightforward as for modal formulas from $NEXT_{\mathcal{MF}}$. In the modal logic, one can only successfully refer to states on a path that lay between the current state and the starting state (inclusive). By the transformation to propositional logic this notion of '*after* or *on* the starting state' gets blurred. Consider the transition diagram in Figure 8 and the following modal formula.

$$\chi \equiv \Box\forall v\colon \mathit{Valuation}.\overline{\textcircled{\mathit{send}(v)}}(\quad \mathit{evaluate}(X, v) = 1 \\ \rightarrow \\ \mathcal{J}^{2\cdot 1+1}(\forall v_1\colon \mathit{Valuation}.\overline{\textcircled{\mathit{send}(v_1)}}\mathit{evaluate}(I, v_1) = 1)$$

χ expresses that if X is true, then in the previous regulation cycle I was true. χ does not hold on every path of the transition diagram. Consider, for instance, the path

$$\langle 11\rangle_1 \xrightarrow{read(\langle 1\rangle)} \langle 111\rangle \xrightarrow{send(\langle 111\rangle)} \langle 11\rangle_2 \ldots$$

In the valuation v, as sent by $send(\langle 111\rangle)$ is $evaluate(X, v)$ equal to 1. However, at state $\langle 11\rangle_2$ one cannot refer to a $send(..)$ three steps back on the path.

But, the proposition that is generated from the program and χ is a tautology. This proposition is

$$\begin{array}{l} ((X_{\mathcal{J}} \leftrightarrow (X_{\mathcal{J}\mathcal{J}} \vee X'_{\mathcal{J}\mathcal{J}}) \wedge I_{\mathcal{J}}) \ \wedge \\ (X'_{\mathcal{J}} \leftrightarrow I_{\mathcal{J}}) \ \wedge \\ (X \leftrightarrow (X_{\mathcal{J}} \vee X'_{\mathcal{J}}) \wedge I) \ \wedge \\ (X' \leftrightarrow I)) \\ \rightarrow (X \rightarrow I_{\mathcal{J}}). \end{array}$$

As a result, one has to keep a distance from the starting point of a path. This distance is minimally twice the time depth of the formula to be verified. Note that these complications do not occur when verifying modal formulas from $NEXT_{\mathcal{MF}}$. Because a *VPI* never terminates, the paths are infinite; there is always a next state.

Proposition 4.17. *Given a timer free and single assignment program $P \in Program$. Let $\mathcal{A}(VPI(P, ids(P), v))$ be the transition system generated by $VPI(P, ids(P), v)$ for some $v \in Valuation$. Let $\mathbb{A}$ be a boolean preserving and minimal algebra for the* μCRL *specification in appendix 2. Let ζ be a substitution. Let $\chi \in JUST_{\mathcal{MF}}(P)$.*

$$\models prop(\mathcal{J}^{td(\chi)}(P)) \rightarrow prop(\chi) \iff \forall v \in PV(P).\mathcal{A}(VPI(P, ids(P), v)), \mathbb{A}, \zeta, 2\cdot td(\chi) \models \chi$$

Lemma 4.18. *Let $\overline{\textcircled{a}}(\phi \rightarrow \psi_1 \wedge \psi_2) \in \mathcal{MF}_{Sig,\mathcal{V}}$, for some signature Sig and some set of variables $\mathcal{V}$, then*

$$\models \overline{\textcircled{a}}(\phi \rightarrow \psi_1 \wedge \psi_2) \leftrightarrow \overline{\textcircled{a}}(\phi \rightarrow \psi_1) \wedge \overline{\textcircled{a}}(\phi \rightarrow \psi_2).$$

By Propositions 4.15, 4.16, 4.17, and Lemma 4.18 one can extend verification to $VPI_{\mathcal{MF}}$.

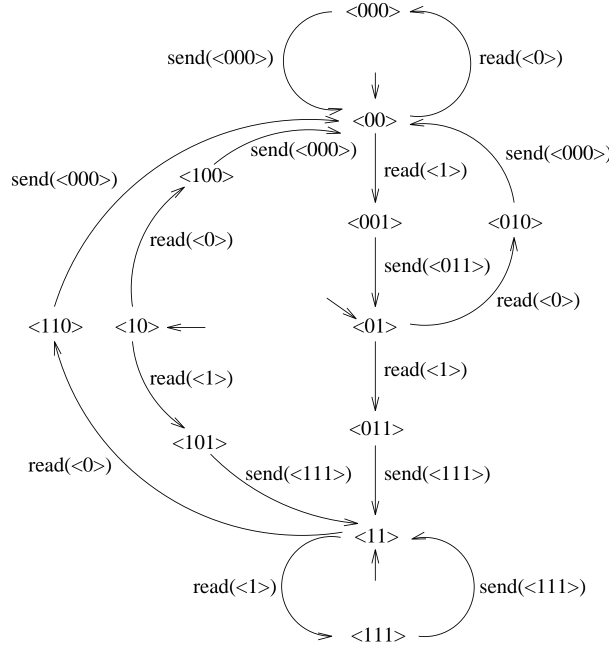

Figure 8: The transition diagram of example 4.1 completed for all proper valuations.

Corollary 4.19. *Let $P \in$ Program be a timer free and single assignment program and let $\chi \in VPI_{\mathcal{MF}}(P)$,*

$$\begin{aligned}\chi \equiv \Box\forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\phi \quad \rightarrow \quad & \psi \ \wedge \\ & \mathcal{J}^{2\cdot i_1+1}(\forall v_1\colon \mathit{Valuation}.\overline{\boxed{send(v_1)}}\psi_1) \wedge \ldots \wedge \\ & \mathcal{J}^{2\cdot i_n+1}(\forall v_n\colon \mathit{Valuation}.\overline{\boxed{send(v_n)}}\psi_n) \wedge \\ & \mathcal{N}^{2\cdot j_1-1}(\forall w_1\colon \mathit{Valuation}.\overline{\boxed{send(w_1)}}\theta_1) \wedge \ldots \wedge \\ & \mathcal{N}^{2\cdot j_m-1}(\forall w_m\colon \mathit{Valuation}.\overline{\boxed{send(w_m)}}\theta_m))\end{aligned}$$

Let $\mathcal{A}(VPI(P, ids(P), v))$ be the transition system generated by $VPI(P, ids(P), v)$ for some $v \in$ Valuation. Let $\mathbb{A}$ be a boolean preserving and minimal algebra for the μCRL specification in appendix 2. Let ζ be a substitution. Then the following holds.

$$\begin{aligned}\models \quad & (prop(\mathcal{J}^0(P)) \rightarrow prop(\phi \rightarrow \psi) \ \wedge \\ & prop(\mathcal{J}^{td(\chi_1)}(P)) \rightarrow prop(\chi_1) \ \wedge \ldots \wedge \ prop(\mathcal{J}^{td(\chi_n)}(P)) \rightarrow prop(\chi_n) \ \wedge \\ & prop(\mathcal{N}^{td(\chi_{n+1})}(P)) \rightarrow prop(\chi_{n+1}) \ \wedge \ldots \wedge \ prop(\mathcal{N}^{td(\chi_{n+m})}(P)) \rightarrow prop(\chi_{n+m})) \\ \Longleftrightarrow & \\ \forall v \in PV(P).&\mathcal{A}(VPI(P, ids(P), v)), \mathbb{A}, \zeta, 2 \cdot max(\{td(\chi_1), \ldots, td(\chi_n)\}) \models \chi\end{aligned}$$

Where

$$
\begin{aligned}
\chi_1 &\equiv \Box\forall v\colon \mathit{Valuation}.\langle\overline{send(v)}\rangle(\phi \rightarrow \mathcal{J}^{2\cdot i_1+1}(\forall v_1\colon \mathit{Valuation}.\langle\overline{send(v_1)}\rangle\psi_1)),\\
&\dots\\
\chi_n &\equiv \Box\forall v\colon \mathit{Valuation}.\langle\overline{send(v)}\rangle(\phi \rightarrow \mathcal{J}^{2\cdot i_n+1}(\forall v_n\colon \mathit{Valuation}.\langle\overline{send(v_n)}\rangle\psi_n)),\\
\chi_{n+1} &\equiv \Box\forall v\colon \mathit{Valuation}.\langle\overline{send(v)}\rangle(\phi \rightarrow \mathcal{N}^{2\cdot j_1-1}(\forall w_1\colon \mathit{Valuation}.\langle\overline{send(w_1)}\rangle\theta_1)),\\
&\dots\\
\chi_{n+m} &\equiv \Box\forall v\colon \mathit{Valuation}.\langle\overline{send(v)}\rangle(\phi \rightarrow \mathcal{N}^{2\cdot j_m-1}(\forall w_m\colon \mathit{Valuation}.\langle\overline{send(w_m)}\rangle\theta_m)).
\end{aligned}
$$

5 Tool support for *VPI*

Automated verification of properties of *VPIs* requires the construction of a set of tools. Of main importance are: a tool that transforms programs and modal formulas into propositions and a tool that proves whether these propositions are tautologies. By means of the ASF+SDF Meta–environment [29] a number of prototype tools were constructed that implement the transformations as presented in this paper. These tools all operate on VLC^*, whereas some operations in this paper operate on μCRL terms representing programs, but this is irrelevant. For the tautology checking various existing tools were used: a binary decision diagram checker [13] developed locally [20]; the resolution theorem prover Otter [32, 33]; and a commercial tool [38].

5.1 Transformation tools

The following transformation tools were constructed.

- A tool for elimination of timer assignments.
- A tool for expansion of timer free programs.
- A tool that implements the n copy just–transformation.
- A tool that implements the n copy next–transformation.
- A tool that translates programs to propositions.
- A simulator for VLC^* programs.
- A tool that performs program slicing.

Most of the transformations are quite simple and will sound familiar now; they will not be discussed in depth. Only program slicing will be introduced shortly in a separate section.

No tool was constructed for the translation of modal formulas. The reason for this is that the modal formulas checked where small; the translation was easily done by hand. However, we are convinced that this tool could have been constructed easily using the ASF+SDF Meta–environment.

All tools, except the simulator, can be used stand alone and as part of the ASF+SDF Meta–environment. The simulator only runs in the Meta–environment. Stand alone, the tools have a very simple interface: read a text file, process the contents, output a text file, terminate. Combinations of tools are then easily constructed using pipes. Static semantical checks are not performed, but could have been specified. Syntax checking is for free with the ASF+SDF Meta–environment.

We tried to generate tools with the ASF+SDF Meta–environment; it is possible to have the system output C code implementing the tool. But, due to the enormous memory consumption of this code, this route turned out not to be practical. Yet, we didn't investigate what the result would be after optimising the specification for the sole purpose of generating efficient code. In the end, the tools were constructed by translating the ASF+SDF specifications by hand using C, Lex and Yacc.

5.2 Program slicing

Intuitively, a slice of P for x is the program one obtains from P by removing all assignments from P that do not contribute to the value of x upon evaluation of P. Slicing of code receives attention in the literature, see [39] for an overview. What is presented here is, using the terminology of [39], backward static slicing, i.e., select in a backward manner a part of the program statements without special assumptions on the input to the program.

Slicing is of practical importance to the verification of *VPI* programs. Suppose one plans to verify that program P obeys $\chi \in VPI_{\mathcal{MF}}$ with a time depth of 100. After all transformations, this gives an impressive proposition and it may take a computer quite some time to find out whether this proposition is a tautology. Now it can be the case that χ only refers to a small number of identifiers. Theorem 5.8 below shows that one can remove that part of the proposition that does not influence the identifiers in χ. This means that the proposition will get smaller and tautology checking will speed up. Of course, this will work if slicing is cheap. The type of slicing considered here has a time complexity that is linear in the number of assignments.

The slicing tool takes as input a VLC^* program and a list of identifiers. The output, the sliced program, can then be translated to a proposition.

Below, slicing will be defined more precisely, followed by a formal treatment in propositional logic.

Definition 5.1. Given a timer free and single assignment program $P \in Program$ and an identifier $id \in Identifier$. This id is the *slicing criterion*, see [39]. Suppose

$$\begin{aligned} P \quad \equiv \quad & add(a(id_1, e_1), \\ & add(a(id_2, e_2), \\ & \ldots \\ & add(a(id_n, e_n), emptyprogram)\ldots)), \end{aligned}$$

then $slice(P, id)$ is the program which is an order preserving selection of assignments in P that satisfies the following:
assignment $a(id_i, e_i)$, $1 \leq i \leq n$, is in $slice(P, id)$ iff

- $eq(id, id_i) = T$, or

- $a(id_j, e_j)$ is in $slice(P, id)$ and $i < j$ and $in(id_i, ids(e_j)) = T$.

It is clear that we can easily extend the notion of slice of a program for an identifier to a slice of a program for an expression, a set of identifiers and to non-timer free programs.

In order to exemplify slicing, suppose that P is the program of Figure 4, i.e.,

$$\begin{array}{lll} P & \equiv & add(\ ta(2, Q, ex(I)), \\ & & add(\ a(R, or(ex(Q), ex(V))), \\ & & add(\ a(V, and(ex(Q), not(ex(R)))), \\ & & add(\ a(U, ex(V)), emptyprogram)))) \end{array}$$

A slice of P for U is P and the same holds for, e.g., the slice of P for an expression as $or(and(ex(R), ex(U)), ex(I))$.

A slice of P for, e.g., R or $or(ex(R), ex(Q))$ is

$$\begin{array}{l} add(ta(2, Q, ex(I)), \\ add(a(R, or(ex(Q), ex(V))), emptyprogram)) \end{array}$$

And a slice of P for I is the empty program because P does not contribute to the value of I at all; I is an input.

A function, *sliceids*, is defined that collects and then renames the identifiers in formulas in $STATIC_{\mathcal{MF}}$, $JUST_{\mathcal{MF}}$ and $NEXT_{\mathcal{MF}}$. These renamed identifiers are needed to properly slice programs with respect to a modal formula.

Definition 5.2. First a function ids_{LP} is defined.

$ids_{LP}(evaluate(e, v) = n), n \in \{0, 1\}$ is the set of identifiers in e
$ids_{LP}(\neg\phi) = ids_{LP}(\phi)$
$ids_{LP}(\phi \vee \psi) \stackrel{\text{def}}{=} ids_{LP}(\phi \wedge \psi) \stackrel{\text{def}}{=} ids_{LP}(\phi \rightarrow \psi) \stackrel{\text{def}}{=} ids_{LP}(\phi \leftrightarrow \psi) \stackrel{\text{def}}{=} ids_{LP}(\phi) \cup ids_{LP}(\psi)$

Now, *sliceids* can be defined.

- $\chi \equiv \Box\forall v\colon Valuation.\overline{\boxed{send(v)}}\phi \in STATIC_{\mathcal{MF}}$, then $sliceids(\chi) \stackrel{\text{def}}{=} ids_{LP}(\phi)$.
- $\chi \equiv \Box\forall v\colon Valuation.\overline{\boxed{send(v)}}(\phi \rightarrow \mathcal{J}^{2\cdot k+1}(\forall v'\colon Valuation.\overline{\boxed{send(v')}}\psi)) \in JUST_{\mathcal{MF}}$, then $sliceids(\chi) \stackrel{\text{def}}{=} ids_{LP}(\phi) \cup \{id_{\mathcal{J}^k} \mid id \in ids_{LP}(\psi)\}$.
- $\chi \equiv \Box\forall v\colon Valuation.\overline{\boxed{send(v)}}(\phi \rightarrow \mathcal{N}^{2\cdot k-1}(\forall v'\colon Valuation.\overline{\boxed{send(v')}}\psi)) \in NEXT_{\mathcal{MF}}$, then $sliceids(\chi) \stackrel{\text{def}}{=} ids_{LP}(\phi) \cup \{id_{\mathcal{N}^k} \mid id \in ids_{LP}(\psi)\}$.

Armed with this machinery, we can reformulate Proposition 4.15, 4.16 and 4.17. For example, the main part of Proposition 4.15 becomes

$$\begin{array}{l} \models prop(slice(\mathcal{J}^0(P), sliceids(\chi))) \rightarrow prop(\chi) \\ \Longleftrightarrow \\ \forall v \in PV(P).\mathcal{A}(VPI(P, ids(P), v)), \mathbb{A}, \zeta, 0 \models \chi \end{array}$$

This reformulation only makes sense when the following holds

Proposition 5.3. *Given a timer free and single assignment program $P \in$ Program and modal formula $\chi \in VPI_{\mathcal{MF}}(P)$.*

$$\models prop(slice(\mathcal{J}^0(P), sliceids(\chi))) \to prop(\chi) \iff \models prop(\mathcal{J}^0(P)) \to prop(\chi).$$

This will be investigated below. Before presenting the results on slicing, some notions will be presented; for a complete and formal treatment of these notions the reader in referred to, e.g., [40].

PROP is the set of all propositional formulas. We define a subset of this, $PROP_a \subset PROP$ is the set of atomic propositions other than $\bot$ and $\top$. For all $\phi \in PROP$, $\alpha(\phi)$ is the set of atomic propositions other than $\top$ and $\bot$ in ϕ. A valuation is a mapping from *PROP* to $\{1, 0\}$. Let σ be a valuation; the interpretation of a formula $\phi \in PROP$ in σ is denoted by $\sigma(\phi)$. This interpretation is defined in the usual way. Furthermore, $\sigma \models \phi$ means that $\sigma(\phi) = 1$ and $\sigma \not\models \phi$ means that $\sigma(\phi) = 0$.

Definition 5.4. The set of hierarchical propositional formulas $HP \subset PROP$ is defined as follows.

- if $x \in PROP_a$, $\phi \in PROP$, and $x \notin \alpha(\phi)$, then $x \leftrightarrow \phi \in HP$.
- if $x \leftrightarrow \phi \in HP$, $\psi \equiv (x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n) \in HP$, $n \geq 1$, $x \notin \{x_1, \ldots, x_n\}$ and $\{x_1, \ldots, x_n\} \cap \alpha(\phi) = \emptyset$, then $(x \leftrightarrow \phi) \wedge (x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n) \in HP$

Lemma 5.5. *Given $\chi \equiv (x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n) \in HP$, $n \geq 1$. For every valuation σ, there is a valuation σ' such that $\sigma'(\chi) = 1$ and for all atoms $y \in PROP_a - \{x_1, \ldots, x_n\}$ it holds that $\sigma(y) = \sigma'(y)$.*

Proof. Given $\chi \in HP$ and some valuation σ. Suppose $\chi \equiv x \leftrightarrow \phi$. So, $x \notin \alpha(\phi)$. Select for σ' a valuation such that $\forall y \in \alpha(\phi).\sigma'(y) = \sigma(y)$, and $\sigma'(x) = \sigma(\phi)$. Suppose $\chi \equiv (x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n)$, $n \geq 1$. By the induction hypothesis there exists a valuation θ such that $\theta((x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_{n-1} \leftrightarrow \phi_{n-1})) = 1$. By definition $x_n \notin \alpha(\phi_n) \cup \alpha((x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_{n-1} \leftrightarrow \phi_{n-1}))$. It is therefore obvious that there is a valuation σ such that $\forall y \in \alpha(\phi_n) \cup \alpha((x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_{n-1} \leftrightarrow \phi_{n-1})), \theta(y) = \sigma'(y)$, and $\sigma'(x_n) = \theta(\phi_n)$. Thus, $\sigma'(\chi) = 1$.

□

Definition 5.6. The set of *VPI* propositions $VPIP \subset PROP$ is defined as follows. If $\mu \in HP$ and $\psi \in PROP$ such that $\alpha(\psi) \subseteq \alpha(\mu)$, then $\mu \to \psi \in VPIP$.

Definition 5.7. Given $\chi \equiv ((x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n)) \to \psi \in VPIP$, $n \geq 1$. An atom x_i, $1 \leq i \leq n$, is said to be *unrelated* to the consequent ψ iff $x_i \notin \alpha(\psi)$ and for all j, $1 \leq j \leq n$ such that $j \neq i$, if $x_i \in \alpha(\phi_j)$, then x_j is unrelated to ψ.

Proposition 5.8. *Given $\chi \equiv ((x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n)) \to \psi \in VPIP$. If x_i, $1 \leq i \leq n$, is not related to ψ, then*
$\models \chi \iff \models ((x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_{i-1} \leftrightarrow \phi_{i-1}) \wedge (x_{i+1} \leftrightarrow \phi_{i+1}) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n)) \to \psi$.
(If $n = i = 1$, the statement is $\models \chi \iff \models \psi$.)

Proof. Given $\chi \equiv ((x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n)) \rightarrow \psi \in \mathit{VPIP}$, $n \geq 1$, and some i, $1 \leq i \leq n$, such that x_i is not related to ψ. Define $A \equiv ((x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n))$, $\chi' \equiv A' \rightarrow \psi$, $A' \equiv ((x_1 \leftrightarrow \phi_1) \wedge \ldots \wedge (x_{i-1} \leftrightarrow \phi_{i-1}) \wedge (x_{i+1} \leftrightarrow \phi_{i+1}) \wedge \ldots \wedge (x_n \leftrightarrow \phi_n))$.
$\Rightarrow$:
Suppose $\models \chi$, and let σ be a valuation. Thus, $\sigma(\chi) = 1$. If $\sigma(\psi) = 1$, then $\sigma(\chi') = 1$. Otherwise, if $\sigma(\psi) = 0$, then $\sigma(A) = 0$. We must show that there is a conjunct a in A' such that $\sigma(a) = 0$. If $\sigma(x_i \leftrightarrow \phi_i) = 1$, there is obviously such an a in A'. Otherwise, we have to proof that there is a conjunct other than $x_i \leftrightarrow \phi_i$ which evaluates to 0 under σ. Extend the notion of 'related to'; x_j is related to x_i iff $x_i \in \alpha(\phi_j)$ or there is a x_k such that x_k is related to x_i and x_j is related to x_k. Then, select from A, $x_i \leftrightarrow \phi_i$, and, in an order preserving way, all conjuncts $x_j \leftrightarrow \phi_j$ in A such that x_j is related to x_i. Name this formula μ. It is clear that $\mu \in \mathit{HP}$. By lemma 5.5, there is a valuation σ' for μ such that $\sigma'(\mu) = 1$ and that leaves everything else untouched, i.e., $\sigma'(\psi) = \sigma(\psi)$ and for all conjuncts a in A and not in μ it holds that $\sigma'(a) = \sigma(a)$. Then, there must be a conjunct a in A and a not in μ such that $\sigma'(a) = 0$, because $\models \chi$ also $\sigma'(\chi) = 1$. This conjunct a is also in A' thus $\sigma(A') = 0$.
$\Leftarrow$:
Suppose $\models \chi'$, and let σ be a valuation. Thus, $\sigma(\chi') = 1$. If $\sigma(\psi) = 1$, then $\sigma(\chi) = 1$. Otherwise, if $\sigma(\psi) = 0$, then $\sigma(A') = 0$. Thus, there is a conjunct a in A' such that $\sigma(a) = 0$. This conjunct occurs also in A, thus $\sigma(A) = 0$, thus $\sigma(\chi) = 1$. □

From Proposition 5.8 follows that Proposition 5.3 holds.

5.3 Tautology checkers

It is well known that satisfiability of propositional formulas is NP-complete. Also, the question whether a formula is a tautology is co-NP complete and as such it is not at all obvious whether it is possible to establish this for formulas expressing the correctness of *VPIs*.

We have initially attempted to establish the correctness of the following three static properties:

- Signal 68 cannot show both red and green.
- If signal 68 shows red, then signal 60 does not show green. Signal 60 immediately precedes signal 68 and should therefore show yellow, flashing yellow or red.
- If signal 70 shows green and track 62CT is occupied, then the barriers of railway crossing 35.0 are closed.

We have tried to prove (or disprove) these properties using three systems; a resolution based system called Otter [32, 33], an improved BDD based theorem prover [13, 20] and a commercially available prover [38]. The problems turned out to large for Otter to handle. However, it should be noted that the structure of formulas tend to transform in relatively small conjunctive normal forms. Therefore, other resolution based theorem provers may turn out to do well on this problem. Plain BDD techniques did not get anywhere, but with the improvement described in [20] sliced formulas could be prove/disproven in times up till a minute whereas the unsliced formulas took approximately an hour to be proven or disproven. All these results

have been obtained on a Sparc server 10 with 256 MB of memory. The system of [38] seems to be able to handle the unsliced formulas in a few seconds. This is very promising. But we neither have gotten the possibility to do elaborate experiments, nor do we have sufficiently understanding of the techniques of [38] to give a firm assessment of this method.

However, we can conclude that although not yet very convincingly, there exist methods and tools to establish the truth or falsity of formulas expressing safety of *VPI* controlled railway yards. Moreover, it is expected that these methods will also turn out to be satisfactory for formulas in the classes $JUST_{\mathcal{MF}}$ and $NEXT_{\mathcal{MF}}$.

6 Concluding Remarks and Future Research

In this paper a model is presented in μCRL of an interlocking as currently in use with NS. Furthermore, a framework for correctness criteria was formulated in the modal logic for μCRL. It was shown how correctness criteria that conform to this framework can be verified automatically. For automated verification a number of tools were specified in ASF+SDF and these specifications were implemented in C.

It turned out that, when verifying formulas taken from $JUST_{\mathcal{MF}}$, one has to obey a certain distance from the start of a path. This does not have to lead to problems when verifying *VPIs* in practice. If a *VPI* is started from a known initial valuation and the time depth of formulas is below, say, 1 or 2 minutes, one can have a model checker verify properties of the transition system that is unfolded up to twice the time depth of the modal formula from $JUST_{\mathcal{MF}}$ at hand. Some ideas for further research are:

- The techniques presented in this paper are in a sense brute force techniques. Is it possible to verify the correctness of *VPIs* in a more analytical way [19]? Following, for instance the approach presented in [35, 41].
- The ideas presented need to be tested. To begin with, a set of generic correctness criteria should be compiled. Then, one can check whether these correctness criteria indeed fit the framework presented.
- Can correctness criteria that refer to states that lay minutes in the past or future still be checked in reasonable time?
- How should one model a set of connected *VPIs*? Do the techniques presented in this paper still apply?
- How do the ideas presented in this paper relate to verification techniques used in, e.g, VLSI design?
- Given a formal representation of a railway yard, one can generate correctness criteria by instantiation of generic correctness criteria [14]. We expect that it is feasible to construct a prototype for a system that does just that with the ASF+SDF Meta–environment.
- Other tools that may be relevant are, for instance, a proposition manipulation tool and a program equivalence tester. With the former one could, e.g., simplify expressions.

With the latter one could test whether a program resulting of an edit operation is *VPI*-operationally equivalent to the original.

- It would be interesting to investigate whether the ToolBus [6] can be used to construct a '*VPI*-workbench' by integration of tools already developed by NS, tools discussed in this paper, and projected tools, e.g., tools proposed in the previous points.

References

[1] S. Anderson and G. Cleland. Formal approaches to safety in programmable electronic systems. Technical report, University of Edinburgh, LFCS, 1993.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[3] A.A. Basten, R.N. Bol, and M. Voorhoeve. Simulating and analyzing railway interlockings in ExSpect. Computing Science Report 94/37, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, September 1994.

[4] J. Berger, P. Middelraad, and A.J. Smith. EURIS, European Railway Interlocking Specification. Technical report, UIC, Committee 7A/16, 1992.

[5] J. Berger, P. Middelraad, and A.J. Smith. The European Railway Interlocking Specification. In *Proceedings*. The Institution of Railway Signal Engineers, January 1993.

[6] J.A. Bergstra and P. Klint. The ToolBus – a component interconnection architecture –. Report P9408, Programming Research Group, University of Amsterdam, 1994.

[7] M.A. Bezem and J.F. Groote. A correctness proof of a one-bit sliding window protocol in μCRL. Technical Report 99, Logic Group Preprint Series, Utrecht University, October 1993. To appear in the Computer Journal.

[8] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. Technical Report 98, Logic Group Preprint Series, Utrecht University, September 1993. To appear in Proceedings Concur'94, Lecture Notes in Computer Science, Springer Verlag.

[9] R.N. Bol, J.W.C. Koorn, L.H. Oei, and S.F.M. van Vlijmen. Syntax and static semantics of the interlocking design and application language. Report P9422, Programming Research Group, University of Amsterdam, 1994. To appear.

[10] M. Broy. Specification of a railway system. Technical Report MIP-8715, Universität Passau, 1987.

[11] J.J. Brunekreef. Process specification in a UNITY format. Report P9329, Programming Research Group, University of Amsterdam, 1993.

[12] G. Bruns. A case study in safety-critical design. In G. von Bochmann and D.K. Probst, editors, *Computer Aided Verification*, volume 663 of *LNCS*, pages 220–233, 1992.

[13] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[14] G.V. Conroy and C. Pulley. Logical methods in the formal verification of safety-critical software. Technical report, UMIST, 1993.

[15] D. Craigen, S. Gerhart, and T. Ralston. *An International Survey of Industrial Applications of Formal Methods–Case Studies*, volume 2. NIST (National Institute of Standards and Technology), 1993.

[16] M.S. Feather. Towards a Derivational Style of Distributed System Design - An Example. *Automated Software Engineering*, 1(1), 1994.

[17] L.M.G. Feijs, H.B.M. Jonkers, and C.A. Middelburg. *Notations for Software Design.* Springer Verlag, FACIT Series, To appear in October 1994.

[18] S. Fischer, A. Scholz, and D. Taubner. Verification in process algebra of the distributed control of track vehicles – a case study. In G. von Bochmann and D.K. Probst, editors, *Computer Aided Verification*, volume 663 of *LNCS*, pages 192–205, 1992.

[19] W.J. Fokkink, 1994. Personal communication.

[20] J.F. Groote. Hiding propositional constants in BDDs. Technical Report 120, Logic Group Preprint Series, Utrecht University, 1994.

[21] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Report CS-R9076, CWI, Amsterdam, 1990.

[22] J.F. Groote and A. Ponse. Proof theory for μCRL. Report CS-R9138, CWI, Amsterdam, 1991.

[23] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, pages 232–251. Workshops in Computing, Springer Verlag, 1994.

[24] J.F. Groote and A. Ponse. Syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes* (ACP94), Workshops in Computer Science, Springer Verlag, 1994.

[25] J.F. Groote and S.F.M. van Vlijmen. A modal logic for μCRL. Technical Report 114, Logic Group Preprint Series, Utrecht University, 1994.

[26] J. Heering and P. Klint. The syntax definition formalism SDF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[27] Ontwerp Voorschrift Electronische Beveiligingssystemen VPL_1, 1993. OV 220.118 t/m 229.110, Deel II aflevering 3B, van Voorschriften Seintechnische Installaties NV Nederlandse Spoorwegen. In Dutch.

[28] M. Ingleby and I. Mitchell. Proving safety of a railway signaling system incorporating geographic data. In Heinz H. Frey, editor, *Proceedings of SAFECOMP '92*. Pergamon Press, 1992.

[29] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[30] M. Koutny. The Merlin-Randell problem of train journeys. *Acta Informatica*, 23:429–463, 1986.

[31] R. de Lemos, A. Saeed, and T. Anderson. A train set as a case study for the requirements analysis of safety-critical systems. *The Computer Journal*, 35(1):30–40, 1992.

[32] W.W. McCune. OTTER 2.0 Users Guide. Technical Report ANL-90/9, Argonne National Laboratory, March 1990.

[33] W.W. McCune. What's new in OTTER 2.2. Technical Report ANL/MCS-TM-153, Argonne National Laboratory, July 1991.

[34] M.J. Morley. Modelling British Rail's Interlocking logic: Geographic data correctness. Technical Report ECS-LFCS-91-186, University of Edinburgh, LFCS, November 1991.

[35] M.J. Morley. Safety in railway signaling data: A behavioural analysis. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, LNCS. Springer Verlag, 1993.

[36] P.J.Th. Musters. VPI Beschrijving van volgordedwang. Technical report, Ingenieursbureau NS, TBS-Productie, 1993. Uitgave D. In Dutch.

[37] L.H. Oei. Pruning the search tree of interlocking design and application language operational semantics. Report P9418, Programming Research Group, University of Amsterdam, 1994.

[38] G. Stålmark and M. Säflund. Modelling and verifying systems and software in propositional logic. In *Proceedings of SAFECOMP '90*, pages 31–36. Pergamon Press, 1990.

[39] F. Tip. A survey of program slicing techniques. Technical report CS-9438, CWI, 1994.

[40] D. van Dalen. *Logic and structure*. Springer Verlag, second edition, 1989.

[41] Wai Wong. *A Formal Theory of Railway Track Networks in Higher-order Logic and its Applications in Interlocking Design*. PhD thesis, University of Warwick, Dept. of Engineering, 1991.

[42] L. Zigterman. *A new approach to specification and design of a railway-interlocking: the 'ROUTE' approach*. PhD thesis, Technische Hogeschool Delft, 1984.

1 A verification example

In this appendix, a small program and some modal formulas are presented. Then, the program and the modal formulas are transformed to propositional logic formulas, which will then be investigated.

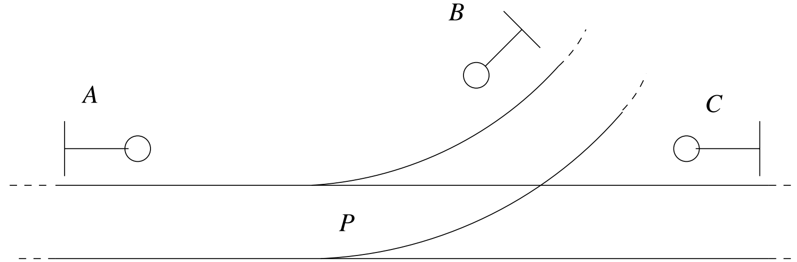

Figure 9: The situation at Little Yard.

Consider the railway yard in Figure 9. The following program 'controls' this railway yard.

DIRECT INPUT SECTION
I
OUTPUT SECTION
$P^r \;\; P^n \;\; A \;\; B \;\; C$
CODE SYSTEM SECTION
$Cmd^A \;\; Cmd^B \;\; Cmd^C \;\; Cmd^r$
CURRENT RESULT SECTION
E
SELF-LATCHED PARAMETER SECTION
TIMER EXPRESSION RESULT SECTION
P
BOOLEAN EQUATION SECTION
APPLICATION $= LY$
BOOL $E = A * B * C + A * B + A * C + B * C$
TIME DELAY $= 1$ SECONDS BOOL $P = I$
BOOL $P^r = .N.A * .N.B * .N.C * Cmd^r$
BOOL $P^n = .N.P^r$
BOOL $A = Cmd^A * .N.Cmd^B * .N.Cmd^C * .N.E * P * P^n$
BOOL $B = Cmd^B * .N.Cmd^A * .N.Cmd^C * .N.E * P * P^r$
BOOL $C = Cmd^C * .N.Cmd^A * .N.Cmd^B * .N.E * P * P^n$
END BOOLEAN EQUATION SECTION

The signals A, B and C can only be green or red. The variables A, B and C are outputs that control the signals with the same name. If A is 1, then A shows green, otherwise A shows red; in the same way the B and C are interpreted. Occupation of the track is measured with input I, if I is 1 the corresponding track level device says the point is free. But, this will be believed only by the VPI if this is measured twice, therefore P is delayed. P^r and P^n indicate

whether the point is reverse or normal position. E is a variable that signals an erroneous situation. Finally, the variables Cmd^A, Cmd^B, Cmd^C, Cmd^r are inputs that correspond to commands issued to the VPI. When Cmd^A is 1, this signals that A should turn green, otherwise, A should turn red. In the same way Cmd^B and Cmd^C are interpreted. When Cmd^r is 1 this signals that the point should move to reverse position, otherwise that the point should move to normal position. Consider the following four modal formulas. The first two will turn out to hold, the last two will turn out not to hold.

- At the most one of the signals A, B and C is green.

$$\chi_1 \equiv \Box \forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\ evaluate(A * B + A * C + B * C, v) = 0\ )$$ [6]

- If signal A, B or C shows green, then I is 1 and also in the previous control cycle I was 1.

$$\begin{aligned}\chi_2 \equiv \Box \forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\ & evaluate(A + B + C, v) = 1 \rightarrow \\ & evaluate(I, v) = 1 \wedge \\ & \mathcal{J}^{2\cdot 1+1}(\forall v_1\colon \mathit{Valuation}.\overline{\boxed{send(v_1)}}\ evaluate(I, v_1) = 1))\end{aligned}$$ [6]

- If A shows green, then the point was already in the normal position.

$$\begin{aligned}\chi_3 \equiv \Box \forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\ & evaluate(A, v) = 1 \rightarrow \\ & \mathcal{J}^{2\cdot 1+1}(\forall v_1\colon \mathit{Valuation}.\overline{\boxed{send(v_1)}}\ evaluate(P^n, v_1) = 1))\end{aligned}$$

- If there is a conflict detected, then the point should not move. An instance of this says, if there is a conflict detected and the point is normal position, then it will stay in normal position.

$$\begin{aligned}\chi_4 \equiv \Box \forall v\colon \mathit{Valuation}.\overline{\boxed{send(v)}}(\ & evaluate(E * P^n, v) = 1 \rightarrow \\ & \mathcal{N}^{2\cdot 1-1}(\forall v'\colon \mathit{Valuation}.\overline{\boxed{send(v')}}evaluate(P^n, v') = 1))\end{aligned}$$ [6]

The question is whether the propositions generated from LY and the formulas χ_1 to χ_4 are tautologies. In other words, do the following statements hold.

- $\overset{?}{\models} prop(slice(\mathcal{J}^0(LY), sliceids(\chi_1))) \rightarrow prop(\chi_1)$
- $\overset{?}{\models} prop(slice(\mathcal{J}^1(LY), sliceids(\chi_2))) \rightarrow prop(\chi_2)$
- $\overset{?}{\models} prop(slice(\mathcal{J}^1(LY), sliceids(\chi_3))) \rightarrow prop(\chi_3)$
- $\overset{?}{\models} prop(slice(\mathcal{N}^1(LY), sliceids(\chi_4))) \rightarrow prop(\chi_4)$

[6] Note that in the modal formulas VLC^* expressions are used instead of the μCRL representations.

Not all transformation steps are given, only the resulting propositions are shown. These propositions are, following the order of the statements above, listed and commented below.

It is not hard to convince oneself that the program obeys χ_1 and χ_2. In accordance with the claims in Section 4, the first two propositions listed below turn out to be tautologies.

- $$\begin{aligned}
&((E \leftrightarrow A_{\mathcal{J}} \wedge B_{\mathcal{J}} \wedge C_{\mathcal{J}} \vee A_{\mathcal{J}} \wedge B_{\mathcal{J}} \vee A_{\mathcal{J}} \wedge C_{\mathcal{J}} \vee B_{\mathcal{J}} \wedge C_{\mathcal{J}}) \ \wedge \\
&(P \leftrightarrow (P_{\mathcal{J}} \vee P'_{\mathcal{J}}) \wedge I) \ \wedge \\
&(P^r \leftrightarrow \neg A_{\mathcal{J}} \wedge \neg B_{\mathcal{J}} \wedge \neg C_{\mathcal{J}} \wedge Cmd^r) \ \wedge \\
&(P^n \leftrightarrow \neg P^r) \ \wedge \\
&(A \leftrightarrow Cmd^A \wedge \neg Cmd^B \wedge \neg Cmd^C \wedge \neg E \wedge P \wedge P^n) \ \wedge \\
&(B \leftrightarrow Cmd^B \wedge \neg Cmd^A \wedge \neg Cmd^C \wedge \neg E \wedge P \wedge P^r) \ \wedge \\
&(C \leftrightarrow Cmd^C \wedge \neg Cmd^A \wedge \neg Cmd^B \wedge \neg E \wedge P \wedge P^n)) \\
&\rightarrow \neg(A \wedge B \ \vee \ A \wedge C \ \vee \ B \wedge C)
\end{aligned}$$

- $$\begin{aligned}
&((E_{\mathcal{J}} \leftrightarrow A_{\mathcal{J}\mathcal{J}} \wedge B_{\mathcal{J}\mathcal{J}} \wedge C_{\mathcal{J}\mathcal{J}} \vee A_{\mathcal{J}\mathcal{J}} \wedge B_{\mathcal{J}\mathcal{J}} \vee A_{\mathcal{J}\mathcal{J}} \wedge C_{\mathcal{J}\mathcal{J}} \vee B_{\mathcal{J}\mathcal{J}} \wedge C_{\mathcal{J}\mathcal{J}}) \ \wedge \\
&(P_{\mathcal{J}} \leftrightarrow (P_{\mathcal{J}\mathcal{J}} \vee P'_{\mathcal{J}\mathcal{J}}) \wedge I_{\mathcal{J}}) \ \wedge \\
&(P'_{\mathcal{J}} \leftrightarrow I_{\mathcal{J}}) \ \wedge \\
&(P^r_{\mathcal{J}} \leftrightarrow \neg A_{\mathcal{J}\mathcal{J}} \wedge \neg B_{\mathcal{J}\mathcal{J}} \wedge \neg C_{\mathcal{J}\mathcal{J}} \wedge Cmd^r_{\mathcal{J}}) \ \wedge \\
&(P^n_{\mathcal{J}} \leftrightarrow \neg P^r_{\mathcal{J}}) \ \wedge \\
&(A_{\mathcal{J}} \leftrightarrow Cmd^A_{\mathcal{J}} \wedge \neg Cmd^B_{\mathcal{J}} \wedge \neg Cmd^C_{\mathcal{J}} \wedge \neg E_{\mathcal{J}} \wedge P_{\mathcal{J}} \wedge P^n_{\mathcal{J}}) \ \wedge \\
&(B_{\mathcal{J}} \leftrightarrow Cmd^B_{\mathcal{J}} \wedge \neg Cmd^A_{\mathcal{J}} \wedge \neg Cmd^C_{\mathcal{J}} \wedge \neg E_{\mathcal{J}} \wedge P_{\mathcal{J}} \wedge P^r_{\mathcal{J}}) \ \wedge \\
&(C_{\mathcal{J}} \leftrightarrow Cmd^C_{\mathcal{J}} \wedge \neg Cmd^A_{\mathcal{J}} \wedge \neg Cmd^B_{\mathcal{J}} \wedge \neg E_{\mathcal{J}} \wedge P_{\mathcal{J}} \wedge P^n_{\mathcal{J}}) \ \wedge \\
&(E \leftrightarrow A_{\mathcal{J}} \wedge B_{\mathcal{J}} \wedge C_{\mathcal{J}} \vee A_{\mathcal{J}} \wedge B_{\mathcal{J}} \vee A_{\mathcal{J}} \wedge C_{\mathcal{J}} \vee B_{\mathcal{J}} \wedge C_{\mathcal{J}}) \ \wedge \\
&(P \leftrightarrow (P_{\mathcal{J}} \vee P'_{\mathcal{J}}) \wedge I) \ \wedge \\
&(P^r \leftrightarrow \neg A_{\mathcal{J}} \wedge \neg B_{\mathcal{J}} \wedge \neg C_{\mathcal{J}} \wedge Cmd^r) \ \wedge \\
&(P^n \leftrightarrow \neg P^r) \ \wedge \\
&(A \leftrightarrow Cmd^A \wedge \neg Cmd^B \wedge \neg Cmd^C \wedge \neg E \wedge P \wedge P^n) \ \wedge \\
&(B \leftrightarrow Cmd^B \wedge \neg Cmd^A \wedge \neg Cmd^C \wedge \neg E \wedge P \wedge P^r) \ \wedge \\
&(C \leftrightarrow Cmd^C \wedge \neg Cmd^A \wedge \neg Cmd^B \wedge \neg E \wedge P \wedge P^n)) \\
&\rightarrow ((A \vee B \vee C) \rightarrow (I \wedge I_{\mathcal{J}}))
\end{aligned}$$

It is also not hard to convince oneself that χ_3 and χ_4 do not hold. In accordance with the claims in Section 4, the corresponding propositions are not tautologies. One can, in both cases, easily construct a mapping that renders the proposition false.

- $$((E_{\mathcal{J}} \leftrightarrow A_{\mathcal{J}\mathcal{J}} \wedge B_{\mathcal{J}\mathcal{J}} \wedge C_{\mathcal{J}\mathcal{J}} \vee A_{\mathcal{J}\mathcal{J}} \wedge B_{\mathcal{J}\mathcal{J}} \vee A_{\mathcal{J}\mathcal{J}} \wedge C_{\mathcal{J}\mathcal{J}} \vee B_{\mathcal{J}\mathcal{J}} \wedge C_{\mathcal{J}\mathcal{J}}) \wedge$$
  $$(P_{\mathcal{J}} \leftrightarrow (P_{\mathcal{J}\mathcal{J}} \vee P'_{\mathcal{J}\mathcal{J}}) \wedge I_{\mathcal{J}}) \wedge$$
  $$(P'_{\mathcal{J}} \leftrightarrow I_{\mathcal{J}}) \wedge$$
  $$(P^r_{\mathcal{J}} \leftrightarrow \neg A_{\mathcal{J}\mathcal{J}} \wedge \neg B_{\mathcal{J}\mathcal{J}} \wedge \neg C_{\mathcal{J}\mathcal{J}} \wedge Cmd^r_{\mathcal{J}}) \wedge$$
  $$(P^n_{\mathcal{J}} \leftrightarrow \neg P^r_{\mathcal{J}}) \wedge$$
  $$(A_{\mathcal{J}} \leftrightarrow Cmd^A_{\mathcal{J}} \wedge \neg Cmd^B_{\mathcal{J}} \wedge \neg Cmd^C_{\mathcal{J}} \wedge \neg E_{\mathcal{J}} \wedge P_{\mathcal{J}} \wedge P^n_{\mathcal{J}}) \wedge$$
  $$(B_{\mathcal{J}} \leftrightarrow Cmd^B_{\mathcal{J}} \wedge \neg Cmd^A_{\mathcal{J}} \wedge \neg Cmd^C_{\mathcal{J}} \wedge \neg E_{\mathcal{J}} \wedge P_{\mathcal{J}} \wedge P^r_{\mathcal{J}}) \wedge$$
  $$(C_{\mathcal{J}} \leftrightarrow Cmd^C_{\mathcal{J}} \wedge \neg Cmd^A_{\mathcal{J}} \wedge \neg Cmd^B_{\mathcal{J}} \wedge \neg E_{\mathcal{J}} \wedge P_{\mathcal{J}} \wedge P^n_{\mathcal{J}}) \wedge$$
  $$(E \leftrightarrow A_{\mathcal{J}} \wedge B_{\mathcal{J}} \wedge C_{\mathcal{J}} \vee A_{\mathcal{J}} \wedge B_{\mathcal{J}} \vee A_{\mathcal{J}} \wedge C_{\mathcal{J}} \vee B_{\mathcal{J}} \wedge C_{\mathcal{J}}) \wedge$$
  $$(P \leftrightarrow (P_{\mathcal{J}} \vee P'_{\mathcal{J}}) \wedge I) \wedge$$
  $$(P^r \leftrightarrow \neg A_{\mathcal{J}} \wedge \neg B_{\mathcal{J}} \wedge \neg C_{\mathcal{J}} \wedge Cmd^r) \wedge$$
  $$(P^n \leftrightarrow \neg P^r) \wedge$$
  $$(A \leftrightarrow Cmd^A \wedge \neg Cmd^B \wedge \neg Cmd^C \wedge \neg E \wedge P \wedge P^n))$$
  $$\rightarrow (A \rightarrow P^n_{\mathcal{J}})$$

- $$((E \leftrightarrow A_{\mathcal{J}} \wedge B_{\mathcal{J}} \wedge C_{\mathcal{J}} \vee A_{\mathcal{J}} \wedge B_{\mathcal{J}} \vee A_{\mathcal{J}} \wedge C_{\mathcal{J}} \vee B_{\mathcal{J}} \wedge C_{\mathcal{J}}) \wedge$$
  $$(P \leftrightarrow (P_{\mathcal{J}} \vee P'_{\mathcal{J}}) \wedge I) \wedge$$
  $$(P^r \leftrightarrow \neg A_{\mathcal{J}} \wedge \neg B_{\mathcal{J}} \wedge \neg C_{\mathcal{J}} \wedge Cmd^r) \wedge$$
  $$(P^n \leftrightarrow \neg P^r) \wedge$$
  $$(A \leftrightarrow Cmd^A \wedge \neg Cmd^B \wedge \neg Cmd^C \wedge \neg E \wedge P \wedge P^n) \wedge$$
  $$(B \leftrightarrow Cmd^B \wedge \neg Cmd^A \wedge \neg Cmd^C \wedge \neg E \wedge P \wedge P^r) \wedge$$
  $$(C \leftrightarrow Cmd^C \wedge \neg Cmd^A \wedge \neg Cmd^B \wedge \neg E \wedge P \wedge P^n) \wedge$$
  $$(P^r_{\mathcal{N}} \leftrightarrow \neg A \wedge \neg B \wedge \neg C \wedge Cmd^r_{\mathcal{N}}) \wedge$$
  $$(P^n_{\mathcal{N}} \leftrightarrow \neg P^r_{\mathcal{N}}))$$
  $$\rightarrow ((P^n \wedge E) \rightarrow P^n_{\mathcal{N}})$$

2 μCRL specification of the *VPI*

sort $Bool$
func $T, F :\rightarrow Bool$
$or, and : Bool\#Bool \rightarrow Bool$
$not : Bool \rightarrow Bool$
$if : Bool\#Bool\#Bool \rightarrow Bool$
var $x, y, z : Bool$
rew $not(T) = F$
$not(F) = T$
$or(T, x) = T$
$or(x, T) = T$
$or(F, x) = x$
$or(x, F) = x$
$and(x, y) = not(or(not(x), not(y)))$
$if(x, y, z) = or(and(x, y), and(not(x), z))$

sort $Natural$
func $0nat :\rightarrow Natural$
$S : Natural \rightarrow Natural$
$eq : Natural\#Natural \rightarrow Bool$
var $x, y : Natural$
rew $eq(0nat, 0nat) = T$
$eq(S(x), 0nat) = F$
$eq(0nat, S(x)) = F$
$eq(S(x), S(y)) = eq(x, y)$

sort $Integer$
func $0 :\rightarrow Integer$
$1 :\rightarrow Integer$
$int : Natural\#Natural \rightarrow Integer$
$eq : Integer\#Integer \rightarrow Bool$
$up : Integer \rightarrow Integer$
$Bool2Int : Bool \rightarrow Integer$
$Int2Bool : Integer \rightarrow Bool$
$if : Bool\#Integer\#Integer \rightarrow Integer$
var $x, y : Natural$
$i1, i2 : Integer$

rew $0 = int(0nat, 0nat)$
$1 = int(S(0nat), 0nat)$
$int(S(x), S(y)) = int(x, y)$
$eq(int(0nat, x), int(0nat, y)) = eq(x, y)$
$eq(int(x, 0nat), int(y, 0nat)) = eq(x, y)$
$eq(int(0nat, S(x)), int(y, 0nat)) = F$
$eq(int(S(x), 0nat), int(0nat, y)) = F$
$up(int(x, 0nat)) = int(S(0nat), 0nat)$
$up(int(0nat, S(x))) = int(0nat, x)$
$Bool2Int(T) = 1$
$Bool2Int(F) = 0$
$Int2Bool(int(S(x), 0nat)) = T$
$Int2Bool(int(0nat, x)) = F$
$if(T, i1, i2) = i1$
$if(F, i1, i2) = i2$

sort $Identifier$
func $id : Natural \rightarrow Identifier$
$eq : Identifier \# Identifier \rightarrow Bool$
var $n1, n2 : Natural$
rew $eq(id(n1), id(n2)) = eq(n1, n2)$

sort $Identifiers$
func $emptyids : \rightarrow Identifiers$
$add : Identifier \# Identifiers \rightarrow Identifiers$
$in : Identifier \# Identifiers \rightarrow Bool$
$in : Identifiers \# Identifiers \rightarrow Bool$
$union : Identifiers \# Identifiers \rightarrow Identifiers$
$rem : Identifier \# Identifiers \rightarrow Identifiers$
$rem : Identifiers \# Identifiers \rightarrow Identifiers$
$isect : Identifiers \# Identifiers \rightarrow Identifiers$
$eq : Identifiers \# Identifiers \rightarrow Bool$
$if : Bool \# Identifiers \# Identifiers \rightarrow Identifiers$
var $id, id1, id2 : Identifier$
$ids, ids1, ids2 : Identifiers$
rew $add(id, add(id, ids)) = add(id, ids)$
$add(id1, add(id2, ids)) = add(id2, add(id1, ids))$
$in(id, emptyids) = F$
$in(id1, add(id2, ids)) = or(eq(id1, id2), in(id1, ids))$
$in(emptyids, ids) = T$
$in(add(id, ids1), ids2) = and(in(id, ids2), in(ids1, ids2))$
$union(emptyids, ids) = ids$
$union(add(id, ids1), ids2) = add(id, union(ids1, ids2))$

$rem(id, emptyids) = emptyids$
$rem(id1, add(id2, ids)) = if(eq(id1, id2), rem(id1, ids), add(id2, rem(id1, ids)))$
$rem(emptyids, ids) = ids$
$rem(add(id, ids1), ids2) = rem(ids1, rem(id, ids2))$
$isect(emptyids, ids) = emptyids$
$isect(add(id, ids1), ids2) = if(in(id, ids2), add(id, isect(ids1, ids2)), isect(ids1, ids2))$
$eq(ids1, ids2) = and(in(ids1, ids2), in(ids2, ids1))$
$if(T, ids1, ids2) = ids1$
$if(F, ids1, ids2) = ids2$

sort *Expression*
func $e : Identifier \to Expression$
$and, or : Expression\#Expression \to Expression$
$not : Expression \to Expression$
$ids : Expression \to Identifiers$
var $id : Identifier$
$e, e1, e2, e3 : Expression$
rew $ids(ex(id)) = add(id, emptyids)$
$ids(not(e)) = ids(e)$
$ids(and(e1, e2)) = union(ids(e1), ids(e2))$
$ids(or(e1, e2)) = union(ids(e1), ids(e2))$

sort *Assignment*
func $a : Identifier\#Expression \to Assignment$
$ta : Natural\#Identifier\#Expression \to Assignment$
$ids : Assignment \to Identifiers$
var $n : Natural$
$id, id1, id2 : Identifier$
$e, e1, e2 : Expression$
rew $ids(a(id, e)) = add(id, ids(e))$
$ids(ta(n, id, e)) = add(id, ids(e))$

sort *Program*
func $emptyprogram : \to Program$
$add : Assignment\#Program \to Program$
$singleassignment : Program \to Bool$
$outputs : Program \to Identifiers$
$inputs : Program \to Identifiers$
$keep : Program \to Identifiers$
$ids : Program \to Identifiers$
var $p : Program$
$id : Identifier$
$n : Natural$

$e : Expression$
$a : Assignment$

rew $singleassignment(emptyprogram) = T$
$singleassignment(add(a(id, e), p)) =$
$\quad and(not(in(id, outputs(p))), singleassignment(p))$
$singleassignment(add(ta(n, id, e), p)) =$
$\quad and(not(in(id, outputs(p))), singleassignment(p))$
$outputs(emptyprogram) = emptyids$
$outputs(add(a(id, e), p)) = add(id, outputs(p))$
$outputs(add(ta(n, id, e), p)) = add(id, outputs(p))$
$inputs(p) = rem(outputs(p), ids(p))$
$keep(emptyprogram) = emptyids$
$keep(add(a(id, e), p)) = union(isect(ids(e), add(id, outputs(p))), keep(p))$
$keep(add(ta(S(n), id, e), p)) = add(id, union(isect(ids(e), outputs(p)), keep(p)))$
$ids(emptyprogram) = emptyids$
$ids(add(a, p)) = union(ids(a), ids(p))$
$ta(0nat, id, e) = a(id, e)$

sort $Valuation$

func $emptyvaluation :\to Valuation$
$add : Identifier\# Integer\# Valuation \to Valuation$
$add : Valuation\# Valuation \to Valuation$
$retrieve : Identifier\# Valuation \to Integer$
$select : Identifiers\# Valuation \to Valuation$
$evaluate : Program\# Valuation \to Valuation$
$evaluate : Assignment\# Valuation \to Valuation$
$evaluate : Expression\# Valuation \to Integer$
$normalize : Valuation \to Valuation$
$ids : Valuation \to Identifiers$
$if : Bool\# Valuation\# Valuation \to Valuation$

var $v, v1, v2 : Valuation$
$id, id1, id2 : Identifier$
$ids : Identifiers$
$n : Natural$
$i, i1, i2 : Integer$
$a : Assignment$
$p : Program$
$e, e1, e2 : Expression$

rew $add(id1, i1, add(id2, i2, v2)) =$
$\quad if(eq(id1, id2), add(id1, i1, v2), add(id2, i2, add(id1, i1, v2)))$
$add(emptyvaluation, v2) = v2$
$add(add(id, i, v1), v2) = add(id, i, add(v1, v2))$

$retrieve(id, emptyvaluation) = 0$
$retrieve(id1, add(id2, i2, v2)) = if(eq(id1, id2), i2, retrieve(id1, v2))$
$select(ids, emptyvaluation) = emptyvaluation$
$select(ids, add(id, i, v)) =$
$\quad if(in(id, ids), add(id, i, select(rem(id, ids), v)), select(ids, v))$
$evaluate(emptyprogram, v) = v$
$evaluate(add(a, p), v) = evaluate(p, evaluate(a, v))$
$evaluate(a(id, e), v) = add(id, evaluate(e, v), v)$
$evaluate(ta(n, id, e), v) =$
$\quad if(eq(evaluate(e, v), 1), add(id, up(retrieve(id, v)), v), add(id, int(0nat, n), v))$
$evaluate(ex(id), v) = retrieve(id, v)$
$evaluate(not(e), v) = Bool2Int(not(Int2Bool(evaluate(e, v))))$
$evaluate(or(e1, e2), v) =$
$\quad Bool2Int(or(Int2Bool(evaluate(e1, v)), Int2Bool(evaluate(e2, v))))$
$evaluate(and(e1, e2), v) =$
$\quad Bool2Int(and(Int2Bool(evaluate(e1, v)), Int2Bool(evaluate(e2, v))))$
$normalize(emptyvaluation) = emptyvaluation$
$normalize(add(id, i, v)) = add(id, Bool2Int(Int2Bool(i)), normalize(v))$
$ids(emptyvaluation) = emptyids$
$ids(add(id, i, v)) = add(id, ids(v))$
$if(T, v1, v2) = v1$
$if(F, v1, v2) = v2$

act $send : Valuation$
$read : Valuation$

proc $VPI(p : Program, show : Identifiers, v : Valuation) =$
$\sum_{inp \in Valuation}($
$read(inp).$
$send(normalize(select(show, evaluate(p, add(inp, v))))).$
$VPI(p, show, select(keep(p), evaluate(p, add(inp, v))))$
$\lhd\, and(singleassignment(p), and(eq(inputs(p), ids(inp)), eq(keep(p), ids(v))))\, \rhd$
$\delta)$

%Some additional functions for manipulation of programs.

func $replace : Identifier \# Identifier \# Expression \rightarrow Expression$
$replace : Identifier \# Expression \# Expression \rightarrow Expression$
$replace : Identifier \# Expression \# Assignment \rightarrow Assignment$
$disjunct : Identifiers \# Identifiers \rightarrow Bool$
$rem : Identifier \# Valuation \rightarrow Valuation$
$if : Bool \# Expression \# Expression \rightarrow Expression$

var $id, id1, id2 : Identifier$
$ids, ids1, ids2 : Identifiers$
$e, e1, e2, e3 : Expression$
$n : Natural$
$i : Integer$
$v : Valuation$

rew $replace(id1, id2, e) = replace(id1, ex(id2), e)$
$replace(id1, e1, ex(id2)) = if(eq(id1, id2), e1, ex(id2))$
$replace(id1, e1, not(e2)) = not(replace(id1, e1, e2))$
$replace(id1, e1, and(e2, e3)) = and(replace(id1, e1, e2), replace(id1, e1, e3))$
$replace(id1, e1, or(e2, e3)) = or(replace(id1, e1, e2), replace(id1, e1, e3))$
$replace(id1, e1, a(id2, e2)) = a(id2, replace(id1, e1, e2))$
$replace(id1, e1, ta(n, id2, e2)) = ta(n, id2, replace(id1, e1, e2))$
$disjunct(emptyids, ids) = T$
$disjunct(add(id, ids1), ids2) = and(not(in(id, ids2)), disjunct(ids1, ids2))$
$rem(id, emptyvaluation) = emptyvaluation$
$rem(id1, add(id2, i, v)) = if(eq(id1, id2), rem(id1, v), add(id2, i, rem(id1, v)))$
$if(T, e1, e2) = e1$
$if(F, e1, e2) = e2$

3 *SDF* specification of VLC^*

Module 'Integers', which specifies also the natural numbers, module 'Booleans' and module 'Layout' are not listed.

```
Vpi
imports  Integers
exports
  sorts  Program Identifier Assignment Expr Input-section
         Output-section Code-section Result-section Parameter-section
         Timer-section Equation-section Application Applications
         Appli-note Assignments
  lexical syntax
    [␣\t\n]             → LAYOUT
    "%"~[\n]*"\n"       → LAYOUT
    [a-zA-Z0-9\-/]+     → Identifier
    "="~[\n]*[\n]       → Appli-note
  context-free syntax
    Input-section Output-section Code-section Result-section
    Parameter-section Timer-section Equation-section                    → Program
    "DIRECT" "INPUT" "SECTION" Identifier*                              → Input-section
    "OUTPUT" "SECTION" Identifier*                                      → Output-section
    "CODE" "SYSTEM" "SECTION" Identifier*                               → Code-section
    "CURRENT" "RESULT" "SECTION" Identifier*                            → Result-section
    "SELF-LATCHED" "PARAMETER" "SECTION" Identifier*                    → Parameter-section
    "TIMER" "EXPRESSION" "RESULT" "SECTION" Identifier*                 → Timer-section
    "BOOLEAN" "EQUATION" "SECTION" Applications
    "END␣BOOLEAN␣EQUATION␣SECTION"                                      → Equation-section
    Assignment*                                                         → Assignments
    Application*                                                        → Applications
    "APPLICATION" Appli-note Assignments                                → Application
    "BOOL" Identifier "=" Expr                                          → Assignment
    "TIME" "DELAY" "=" NAT "SECONDS" "BOOL" Identifier "=" Expr         → Assignment

    "TRUE"                                                              → Expr
    "FALSE"                                                             → Expr
    Identifier                                                          → Expr
    ".N." Expr                                                          → Expr
    Expr "+" Expr                                                       → Expr    {assoc}
    Expr "*" Expr                                                       → Expr    {assoc}
    "(" Expr ")"                                                        → Expr    {bracket}
  priorities
    ".N."Expr → Expr > Expr "*"Expr → Expr > Expr "+"Expr → Expr
```