Traffic Flow by Data Flow

Jan Philipps Alexander Schmidt

Institut für Informatik
Technische Universität München
D-80290 München

FOCUS

E-mail: {philipps,schmiale}@informatik.tu-muenchen.de

Abstract

The traffic light controller of a simple two-way intersection is specified by data flow components. The focus of this work is on the use of readable description techniques, such as state transition diagrams and tables, to hide the mathematical background of the specifications. A second main point is the use of automatic formal verification tools for the proof of some simple behavioral properties of the specification to validate the specifications.

1 Introduction

During the last years, engineering techniques for embedded systems have become a subject of intensive research. Embedded systems contain both hardware and software parts; their design requires methods from control and software engineering. Since design errors can lead to damage or loss of the system, or even of human life, there are high demands on the system's correctness.

Design methods based on mathematical foundations promise to alleviate part of the problems in system design. Formal specifications can reduce misunderstandings between designers. Moreover, they open the door to formal verification of systems. Verification—especially at the early design steps—is much more trustworthy than other approaches to quality assurance, such as tests or reviews.

However, formal methods require experienced and mathematically well-educated designers. The specifications produced are often unreadable to the uninitiated, and in practice it is all but impossible to verify real-life systems by hand.

In this report we want to show how formal methods can receive higher acceptance by practitioners. We suggest two approaches:

- With suggestive notations like state transition diagrams and tables, the specifications become easier to write, to read, and to validate.
- Model checking tools give an almost fully automated approach to system verification. The properties to be verified are expressed in a temporal logic, and often quite difficult to formulate. However, we believe that it is feasible to formulate and verify simple behavioral properties to gain a second view of the system, orthogonal to the state machine or tabular system specifications.

We present these approaches through a case study. We specify a—very simple—traffic controller of a two-road intersection. As our semantic framework we use Focus [1, 2], which has been extended in [6, 3] by readable description techniques, such as state transition diagrams. We then translate the specification into the input language of the model checker SMV [16] and verify some behavioral properties.

The rest of this work is structured as follows. In the remainder of this section, we present the informal specification of the traffic light controller, as it would be given by a customer from industry. In Section 2, we introduce the description techniques used for our specification: state transition diagrams and tables. In Sections 4 and 5 the specification of the traffic control system is developed.

Section 6 demonstrates how model checkers can be used to validate the traffic controller specification. Section 7 summarizes our work.

Informal specification. We now give a summary of the first description of the case study [11]. We omitted some aspects of the original case study, such as a control panel that displays information about the current state of the controller; the addition of the necessary outputs would be straightforward. We added some details missing in the original description, such as timing information, initial states of the controller, fairness requirements, and the behavior of the controller when it switches between its operation modes.

The traffic light system is shown in Figure 1. It controls the signals of a two-road intersection. Each road has two lanes. We call the road that has the right-of-way when the traffic lights are switched off the "primary road"; the other road is the "secondary road".

For each lane there is a signal lamp, and two inductive loops. Each signal has the three lights red, yellow, and green. There are no special provisions for pedestrians or for left or right turns. One inductive loop is directly at the signal lamp and detects waiting cars. The second loop is 30 meters ahead of the intersection and is used to detect approaching cars.

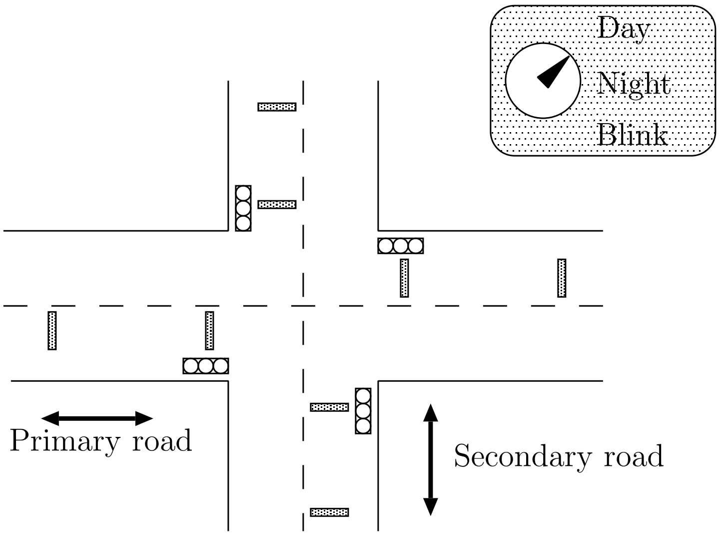

Figure 1: Traffic Intersection

Both signal lamps and inductive loops can fail. Upon failure of a signal lamp, the controller enters a special mode where the yellow lights on the secondary road are blinking. Failure of the loops cannot be detected by the controller. Instead, a loop is assumed to be defect when it signals a car continuously for 10 minutes.

The traffic light controller can operate in four modes:

Daytime mode: In this mode, a road where a waiting car is detected by an induction loop close to a signal lamp should enter its green phase after at most four minutes. Additionally, every green phase lasts at least 45 seconds.

Nighttime mode: During nighttime mode, each road is signaled "Red"by default. An approaching car detected by one of the loops ahead of the intersection will cause the controller to enter the green phase for that direction for 10 seconds. With every car that is detected during this green phase in the same direction the 10 seconds start over. The overall duration of a green phase must not exceed four minutes.

Constant-time mode: In this mode, each road enters alternatingly its green phase for three minutes. Constant-time mode is entered automatically when an inductive loop is assumed to be defect.

Blinking mode: Here, all signal lamps on the primary road are switched off, and the signal lamps on the secondary road blink with a frequency of 0.5 Hz.

Blinking mode is entered when either a lamp is defect, or when the external mode switch is put into the blinking position. When the blinking mode is entered, it must be ensured that all traffic lights switch to "Red" for two seconds before blinking begins.

Which mode the traffic controller is working in is determined by an external mode switch. The switch can be either in daytime, nighttime, or blinking position. No further assumptions about fairness or traffic optimization are made.

2 Specification Techniques

The semantics of our specifications is based on the methodology Focus [1, 2]. We assume the reader is familiar with the basic concepts of Focus; in this section we only give a short summary of those concepts that are relevant to this work.

In this section, we also describe our state machine model and introduce a tabular specification style that allows us to concisely express the conditions for a transition between two automaton states. Then, a simple approach for the specification of real-time properties is presented.

2.1 Streams and Behaviors

In Focus components communicate via streams of messages. In this section we give a brief introduction into the concept of streams as far as they are used here. Details can be found in [1], [2], or [3].

Let M be a set of messages. Then, we denote by M^∞ the set of infinite streams of elements of M and by M^* the set of finite streams over M. A complete communication history $M^\aleph$ is then defined as an infinite stream of finite streams over M:

$$M^\aleph =_{def} (M^*)^\infty$$

Thus, a communication history is intuitively an infinite sequence of time intervals; each interval contains a finite number of messages transmitted in the interval.

Frequently, not the full power of this stream model is needed. For our purposes, it is sufficient to restrict ourselves to *synchronous streams*, where each time interval contains at most one message. Thus, synchronous streams over M are equivalent to infinite streams over the message set $M \cup \{\perp\}$, where $\perp$ stands for the absence of a proper element of M in an interval. To simplify notation, we simply write M^∞ instead of $(M \cup \{\perp\})^\infty$ for synchronous streams.

Synchronous streams are often used for hardware specifications [5]. See [2] for a discussion of these and other kinds of streams used in Focus.

We need the following operations on synchronous streams:

$m\&x$	denotes the prefixing of an element m to a stream x
x_i	delivers the i-th element of x

For an overview of other operators used in FOCUS see [15, 2].

Let C be a set of channel identifiers. An element of the set $C \rightarrow M^\infty$ of functions from a channel identifier to a synchronous streams is called a *channel valuation*. For $I \subseteq C$ and $O \subseteq C$, a function

$$F : (I \rightarrow M^\infty) \rightarrow \mathcal{P}(O \rightarrow M^\infty)$$

is called a stream-processing function. Stream processing functions model the input/output-relation or the behavior of a system component. Note that for each input history, there may be several output histories. Thus, nondeterministic components can be modeled.

Again, we refer to [2] for a discussion of consistence and other properties of behaviors, of behavior composition, and of behavioral refinement.

2.2 State Transition Machines

The main component of the traffic light system is specified with a state transition machine or automaton, for short. Our state transition diagrams are derived from the work in [3], and also related to [7, 6].

Abstract syntax. A state transition machine $\mathcal{A}$ is a six-tuple $(I, O, S, V, V_0, \Delta)$, where

- I, O are sets of input and output channels. Every channel transmits a type of messages.
- $S \subseteq D_1 \times \ldots \times D_n$ is the set of states. It is composed by the system's attributes $s = s_1 : D_1, \ \ldots, \ s_n : D_n$ together with their types.
- V is a set of nodes we use to represent arbitrary equivalence classes of S.[1] If necessary, an additional attribute which holds a node number can be introduced to ensure definiteness. V_0 contains the initial nodes.
- $\Delta \subseteq V \times \mathcal{P}(P_\Delta) \times V$ is a set of transitions. A transition predicate P_Δ has the form

  $$\textit{pre};\ \textit{input}\ /\ \textit{output};\ \textit{post}$$

  Let x be the tuple of finite input and y be the tuple of finite output sequences. In addition, $s, s' \in S$ represent the state before and after the transition. Then

  - *pre* is a predicate over s indicating the precondition,
  - *input* is a predicate over s, x which denotes the required input stream tuple,
  - *output* is a predicate over s, x, y to define the output stream tuple,

[1]Some people call these equivalence classes *control states* and define the state space explicitely to be the product from *data states* and *control states*.

– *post* is a predicate over s, x, y, s' which indicates the state change by this transition.

A transition from node v to v' labeled with *pre*; *input* / *output*; *post* is allowed to fire if the *pre*condition is true and the input matches the pattern specified in *input*. Then, *output* is generated, and the state is changed to match *post* which implies v' to become the current node. Nonmatching input is ignored; the system stays in its current state and produces no output. The powerset $\mathcal{P}(P_\Delta)$ allows us to specify multiple predicates on a single transition.

The semantics of state transition systems can be defined by translating them to stream processing functions used by FOCUS. See [3, 6] for details.

2.3 Input and Output Expressions

There are special predicate forms dealing with input and output. To test whether a constant value c is available on channel i, we write $i?c$. We can also use input variables in input expressions as in $i?v$; all variables v that occur in input patterns are existentially quantified at the transition predicate level. To output a constant c over a channel o we write $o!c$.

In general, the input and output patterns will have the form

$$\begin{array}{rl} i_1?E_1, \ldots, i_{|I|}?E_{|I|} & \text{for input patterns, and} \\ o_1?B_1, \ldots, o_{|O|}?B_{|O|} & \text{for output patterns.} \end{array}$$

with E_p and B_q beeing expressions of the message type of i_p and o_q. $|M|$ simply denotes the cardinality of a set M.

As stated in Section 2.1 the type of message streams is lifted by a special symbol $\perp$, which is implicitly sent if there is no output command with a proper message.

2.4 Decision Tables

State transition machines are well suited for a graphical notation with *state transition diagrams* (STDs). The elements from V become the vertices and the edges get labeled by P_Δ. Frequently the transition predicates P_Δ become quite large. Writing them in the usual logical notation yields deeply nested formulas that are difficult to read and write. There has been some work to make formulas that occur in specification more accessible by writing them in tabular form, e.g. in [14, 17]; we use a similar but more compact approach.

We specify a transition with a *decision table* as follows:

$Condition_1(s,x)$	$C_{1,1}$	$\cdots$	$C_{1,m}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$Condition_k(s,x)$	$C_{k,1}$	$\cdots$	$C_{k,m}$

$Action_1(s,x,y,s')$	$A_{1,1}$	$\cdots$	$A_{1,m}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$Action_l(s,x,y,s')$	$A_{l,1}$	$\cdots$	$A_{l,m}$

The upper half of the decision table is called the *condition block*; the lower half the *action block*. The leftmost column contains conditions (preconditions and input patterns) or actions (postconditions and output patterns). In the condition block, the boxes in the other columns contain either a T (true, condition must be fulfilled), a F (false, condition must be false) or a '·' (don't care, the truth value of the condition is irrelevant). In the action block, the boxes contain either a X (the action must be executed) or are left empty (the action need not be executed). The compactness of the decision tables arises from writing a term once and reusing it in multiple occurrences in the transition predicate.

The idea is that for each column, when all conditions marked T are true, and all conditions marked F are false, all action expressions marked X must be true as well (P_{cause}). Moreover, all actions that are not forced to be true by any true column of the table, are false; this ist stated by P_{close}: Every action which is executed needs at least one X in a column which is true.

Formally, the semantics of a decision table D with m columns, k rows in the condition block and l rows in the action block is a transition predicate P_Δ given by:

$$P_\Delta(s,x,y,s') =_{def} P_{cause}(s,x,y,s') \wedge P_{close}(s,x,y,s')$$

$$P_{cause}(s,x,y,s') =_{def} \bigwedge_{1\le j\le m} \left(Col_j(s,x) \Rightarrow \bigwedge_{1\le i\le l} Q_{i,j}(s,x,y,s') \right)$$

$$P_{close}(s,x,y,s') =_{def} \bigwedge_{1\le i\le l} \left(Action_i(s,x,y,s') \Rightarrow \bigvee_{\{j \mid A_{i,j}=\mathsf{X}\}} Col_j(s,x) \right)$$

$$Col_j(s,x) = \bigwedge_{1\le i\le k} P_{i,j}(s,x)$$

Here the predicates $P_{i,j}$ and $Q_{i,j}$ distinguish between the different entries in the boxes:

- If $C_{i,j} = \mathsf{T}$, then $P_{i,j}(s,x) =_{def} Condition_i(s,x)$
- If $C_{i,j} = \mathsf{F}$, then $P_{i,j}(s,x) =_{def} \neg Condition_i(s,x)$
- If $C_{i,j} = \cdot$, then $P_{i,j}(s,x) =_{def}$ true

- If $A_{i,j} = \mathsf{X}$, then $Q_{i,j}(s, x, y, s') =_{def} Action_i(s, x, y, s')$
- If $A_{i,j}$ is empty, then $Q_{i,j}(s, x, y, s') =_{def}$ true

Note that P_{cause} uses no disjunction but a conjunction: *Every* action column whose condition column is true needs to be true, too. A simple syntactical criterium can be used to avoid possible contradicitions: Condition columns which can be true at the same time must be equal in those actions which have the same left side. Nondeterminisism can be introduced with two or more transition arrows between states, where each transition is labeled with its own decision table.

More conventionally, a decision table can be regarded as a number of if-then-else statements, one for each column, that are executed in parallel. For example, in the case that the action block consists just of two actions $v' = e_1$ and $o!e_2$, where v is a state attribute, o a channel identifier, and e_1, e_2 are expressions, the statement for the j-th column would read:

if $P_{1,j}(s, x) \wedge \ldots \wedge P_{k,j}(s, x)$
then $o!e_2; s' = s[e_1/v]$
else skip
endif

Of course, we have to ensure that the parallel execution of these statements causes no conflicts, and the syntactical criterium given above is basically a rephrasing of the Bernstein condition.

2.5 Time Properties

Focus takes the view that streams on all channels are divided into intervals of equal duration. For our synchronous streams, this can be achieved by having all communication channels run at the same clock rate.

In order to make specifications simpler, we do not count intervals to achieve timing information. Instead we use an external clock to provide the system with the current time. The clock itself can be specified as a Focus component with no inputs and a single output. Its output is required to be a strictly monotonically increasing sequence of real numbers, that contains for each interval exactly one time value. We assume that time starts as 0, although any other value would do.

This output is sent to all components of the traffic control system that need timing information. Automata can read input from the clock with an input pattern $Clock?T$, store T in attributes, or compare T with previously read time values.

Since most transitions of the traffic control automaton use time information, we will for reasons of brevity omit the line $Clock?T$ from the transition tables, and just access a time variable T. The semantics of the machine is still defined as if the line were present, however, reading a time value from our clock component.

3 Controller Structure and Data Types

Our specification follows the structure outlined in Figure 2. The eight inductive sensors are shown at the left. The loop controller combines signals from opposite sensors to a single signal. Furthermore, it checks whether a sensor indicates a vehicle for more than 10 minutes; if so, it assumes a failure in the sensor and notifies the traffic controller via the signal line LF. The signal controller on the right side splits a single signal sent by the traffic controller into separate signals for the traffic lights. It also checks the signals that indicate a damaged light bulb. The control panel at the bottom of the figure contains switches for choosing the operation mode manually.

The traffic controller at the center of the figure is the heart of the specification. It reads signals from the control panel, loop and signal controller, and outputs commands to the signal controller. It ensures the proper traffic flow in the various operation modes as required by the informal specification.

The clock is a virtual component: it sends a stream of time signals used by the traffic controller for the real time requirements. This component could be implemented as real time clock in a computer, but in our work we just regard it as a source of monotonically increasing time stamps.

Data types. The channels in Figure 2 are identified by a name together with a type describing the type of the messages transmitted via that channel. Similarly, attributes of the control automata are identified by their name and data type.

As data types we use the natural numbers $\mathbb{N}$, boolean values $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$, and for time values the real number $\mathbb{R}$ extended by the special value ∞, which is assumed to be larger than any value from $\mathbb{R}$. In addition, we introduce the following enumeration types.

When components signal the presence or absence of a value, we use the following type:

$$\mathbb{O} = \quad \{On, Off\}$$

For the two road names, the following type is introduced:

$$\mathbb{D} = \quad \{A, B, \circledcirc\}$$

Here A stands for the primary, B for the secondary road. The symbol $\circledcirc$ is an auxiliary symbol that will be used to ensure the traffic controller is fair to each direction. For each direction $d \in \mathbb{D}$, we write $d^\times$ for the crossing direction, i.e. $A^\times = B$, $B^\times = A$, $NoDir^\times = \circledcirc$.

The control panel sends messages of the type $\mathbb{M}$ to indicate the operation mode:

$$\mathbb{M} = \quad \{Day, Night, Blink\}$$

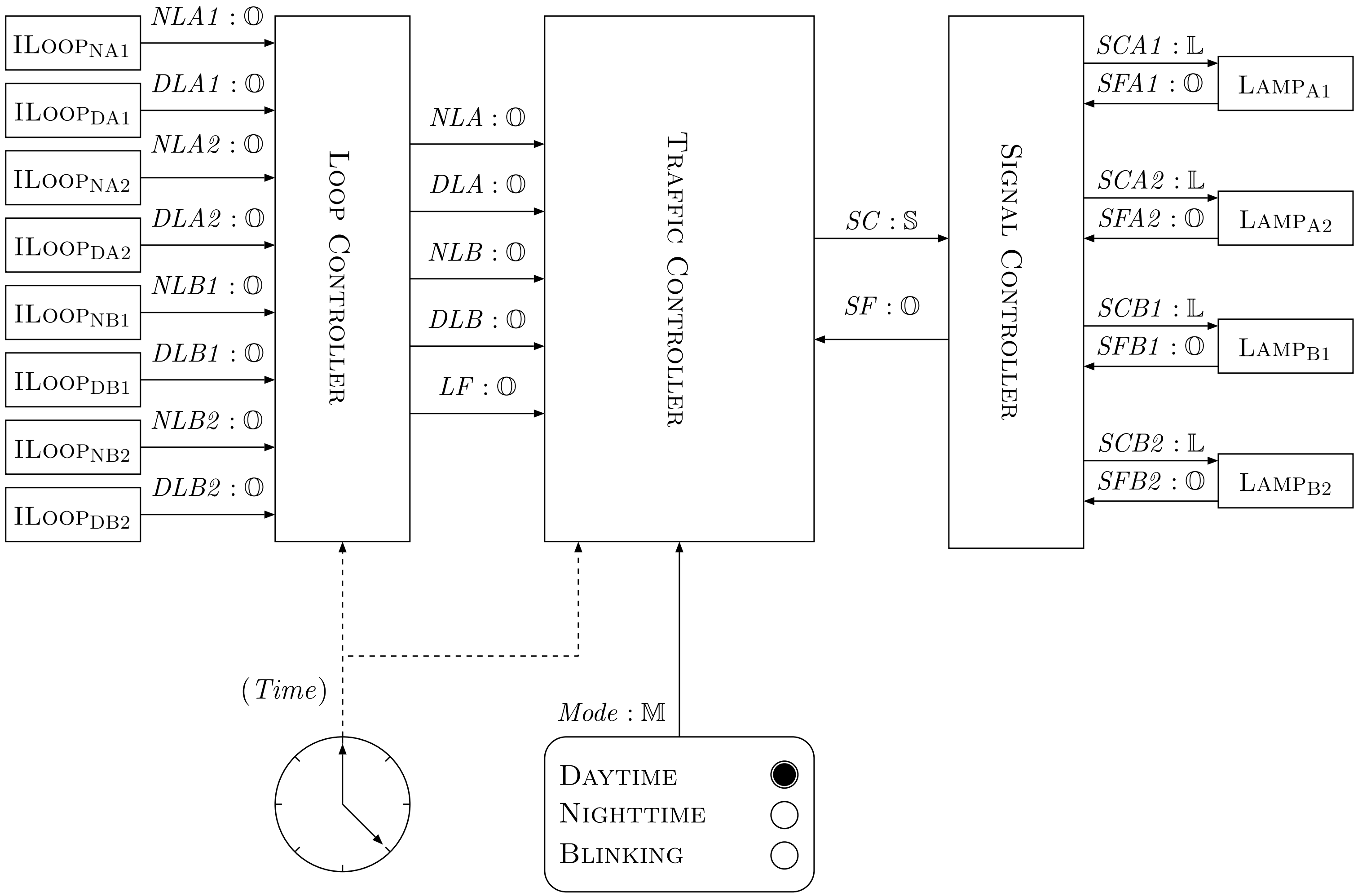

Figure 2: System layout

The traffic controller sends messages of type $\mathbb{S}$ to the signal controller:

$$\begin{aligned}\mathbb{S} = \quad & \{\mathit{AllOff}, \mathit{AllRed}, \mathit{BYel}\} \quad \cup \\ & \{\mathit{Red}(d), \mathit{RedYel}(d), \mathit{Green}(d), \mathit{Yellow}(d) \mid d \in \mathbb{D}\ \}\end{aligned}$$

Here *BYel* instructs the signal controller to switch the signals on the secondary road to yellow, and those on the crossing road off; *AllOff* and *AllRed* switch all four signals off or to red. *BYel* and *AllOff* are used to generate the blinking failure mode. The other messages switch the road denoted by the parameter to the corresponding color; the crossing road will be red. This means that there is some redundancy: $\mathit{Red}(d)$ and *AllRed* both instruct the signal controller to switch all signals to red, but it will make the specification a bit more readable.

From the messages in $\mathbb{S}$ the signal controller generates appropriate commands for the traffic lights themselves; this message set is denoted by $\mathbb{L}$.

$$\mathbb{L} = \quad \{\mathit{Red}, \mathit{RedYel}, \mathit{Green}, \mathit{Yellow}, \mathit{Off}\}$$

4 Traffic Controller

The structure of this section is as follows. First, we give a short overview over the operation of the traffic controller and explain in more detail the controller behavior in constant-time, daytime, and nighttime mode for a small subset of the green phase of road A: we examine the behavior for the transition from "road B is red, road A is red-yellow" to "road B is red, road A is yellow". The complete behavior, including the green phase for road B, is developed in Section 4.2. There we show how by parameterizing states and transitions with similar behavior, the number of states of the controller diagram can be reduced. Finally we assemble the results to the complete specification, and add the exception behavior for burnt-out lamps or the case that blinking mode is activated.

4.1 Green Phase

First, we model only part of the controller operation for each mode to demonstrate how the controller operates and how specifications are written in our framework. In order of increasing complexity of the specifications we examine first constant-time and daytime modes, finally the nighttime mode.

Constant-time mode. In this mode, the controller remains for one second in state RedYelA before it instructs the signal controller to switch the signals on road A to green, and moves to state GreenA. There, it waits for 180 seconds and then continues to YellowA; also, the signal controller must switch the signals on road A to yellow.

Figure 3 shows the corresponding automaton with transition names of the green phase for road A during constant-time mode; Figure 4 shows the decision tables for the two transitions.

The transition definitions make use of an automaton attribute $\mathcal{E}$; by convention, we use it to hold the time the current state was entered. As explained on page 8, the variable T contains the current output from the clock.

The idea is that $\mathcal{E}$ holds the time RedYelA has been entered; when the clock outputs a time value that is larger than $\mathcal{E} + 1$ (which might take several Focus intervals), the transition will be taken, *Green*(A) is sent to the signal controller, and $\mathcal{E}$ is set to the current time. Then, when the clock outputs a time value that has further increased by 180, the transition STOP will be taken.

Figure 3: Constant-time and daytime diagram

$T \geq \mathcal{E} + 1$	T
$SC!Green(A)$	X
$\mathcal{E}' = T$	X

(a) GO

$T \geq \mathcal{E} + 180$	T
$SC!Yellow(A)$	X
$\mathcal{E}' = T$	X

(b) STOP

Figure 4: Constant-time transitions

Daytime mode. The state diagram for the daytime is the same as for the constant-time mode in Figure 3. As in constant-time mode, the traffic controller waits in state RedYelA for one second. In daytime mode, however, GreenA can only be left when two conditions are fulfilled:

- The controller remained in the state for at least 45 seconds.
- A car is waiting at the crossing road B. This is signaled by the loop controller via input *NLB*.

In addition, GreenA must be left after at most four minutes after a car is detected on road B. This requirement is implicitly met if the implementation of the traffic controller is fast enough: GreenA is left as soon as possible.

Figure 5 shows the transition definitions for daytime mode.

$T \geq \mathcal{E} + 1$	T

$SC!Green(A)$	X
$\mathcal{E}' = T$	X

(a) GO

$T \geq \mathcal{E} + 45$	T
$NLB?On$	T

$SC!Yellow(A)$	X
$\mathcal{E}' = T$	X

(b) STOP

Figure 5: Daytime transitions

Nighttime mode. The controller behavior in nighttime mode is more complicated than in the other two modes. Figure 6 shows a first attempt to model the transition structure. As in constant-time and daytime mode, the state RedYelA is left after one second.

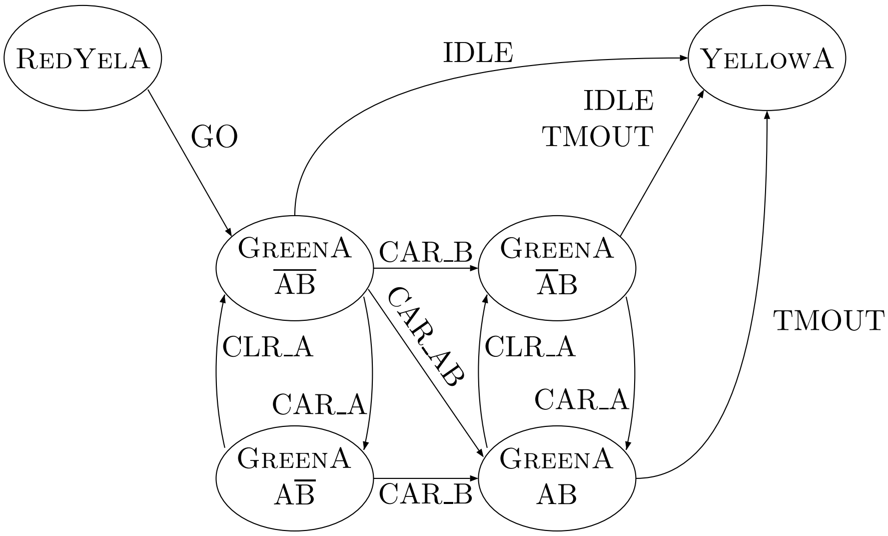

Figure 6: Nighttime diagram

However, while road A is in its green phase, the controller must detect the following events:

- The crossing of a car driving on road A. This is detected by the inductive loops near the signal lights: first, the loop signals a car, marking its arrival, and then it signals that the area around the lights is free again, marking the car's departure.
- The arrival of a car at road B. Since the signals on road B show red, the car has to wait; its presence is detected by the near loops on road B.

In Figure 6 the former state GreenA is therefore split in four states:

- In state $\overline{\mathrm{AB}}$, no car waits on road B, and no car has arrived on road A. This is the state that is entered from RedYelA. If, for example, a car has arrived on road B already, the controller will switch to one of the states below in the next cycle.
- In state $\mathrm{A}\overline{\mathrm{B}}$, a car has entered the crossing from road A, but no car is waiting on road B.
- In state $\overline{\mathrm{A}}\mathrm{B}$, a car is waiting on road B, but no car is currently passing via road A.
- Finally, in state AB, both a car has entered the crossing on road A, and a car is waiting on road B.

The transitions CAR_A, CLR_A and CAR_B handle arrival and departure of cars:

- Transition CAR_A is followed when a car is detected by the near loops on road A;
- CRL_A is followed when the car departed and according to the loops the intersection is cleared;
- CAR_B is followed when a car is detected waiting on road B.

The transition CAR_AB handles the case that cars are detected on both roads A and B; this transition is necessary because of the synchronous nature of our state machine model. Inputs that are not processed in one cycle are lost and could only in an explicit store be saved for the next cycle. A corresponding transition CLR_AB from state $\mathrm{A}\overline{\mathrm{B}}$ to state $\overline{\mathrm{A}}\mathrm{B}$ is not necessary.

Finally, transition IDLE is taken when no car has crossed via road A for ten seconds; transition TMOUT is taken when a car has waited on road B for more than 240 seconds. In both cases, the controller proceeds to state YellowA.

Note that there is no transition from state $\mathrm{A}\overline{\mathrm{B}}$ to YellowA. It might seem that a disgruntled employee could park his old car on road A instead of scrapping it, to ensure free passage on his way home from work. However, in this case, the loop controller will send a failure signal to the traffic controller; later we will modify the transitions so that they then switch to constant-time mode and thus proceed to YellowA anyway.

We skip the formal transition definitions for Figure 6; instead we work towards a simpler transition diagram, where the information about the presence of cars is encoded not in the vertices, but in the attributes of the control automaton.

First, we introduce the following attributes in addition to $\mathcal{E}$:

- A second time attribute $\mathcal{C}$; it holds the time a car arrived on road B. When the system time exceeds $\mathcal{C} + 240$, i.e. the car has been waiting for more than four minutes, the transition TMOUT is taken. Initially this variable is set to ∞, disabling transition TMOUT.

- A boolean attribute $\mathcal{B}$; it is true whenever the crossing is occupied with cars ("busy"). When a car arrives on road A, it is set to true, when the car leaves again it is set to false. Initially, it is set to false.

Figure 7 shows the new transition structure. It resembles that of constant-time and daytime mode, except for the new transition LOOP, which subsumes the transitions between the four AB states in Figure 6.

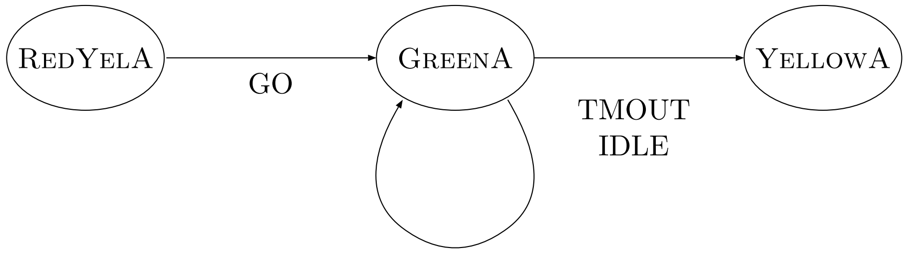

Figure 7: Nighttime diagram (folded)

The transition definitions are shown in Figure 8. Transition GO is similar to the other two modes, but initializes the new attributes. TMOUT and IDLE are straightforward; note that in TMOUT the system time is compared to $\mathcal{C}$, not $\mathcal{E}$. More interesting is transition LOOP. The first column corresponds to the CAR_A transitions. When the crossing is free, and a car is detected on road A, the crossing is marked as occupied. Conversely, in the second column, when the crossing is occupied, and no car is detected anymore, the crossing is marked as free again, and the time the car has left is noted in $\mathcal{E}$. The third column corresponds to CAR_B; when no car has yet been waiting on B, but a car arrived, the arrival time is noted in $\mathcal{C}$.

4.2 Introducing Symmetry

So far we examined only part of the controller's behavior, namely from the time the signals on road A shows red and yellow, via road A's green phase, to the time where the signals on road A show only yellow. To complete this green phase of road A, we also need the behavior until and from the time the signals of both roads are set to red.

In addition, there are similar transitions for the green phase of road B. Since the behavior for the two roads should be symmetrical, we parameterize the behavior with a new attribute, $\mathcal{D}$, that stores the name of the road that will be signaled green in the current cycle. We then arrive at the state transition diagram shown in Figure 9.

The decision which road will be signaled green has to made in the state BOTHRED, where all signals are set to red. In daytime and constant-time mode, the direction attribute is simply set to the crossing road:

$$\mathcal{D}' = \mathcal{D}^{\times}$$

$T \geq \mathcal{E} + 1$	T
$SC!Green(A)$	X
$\mathcal{E}' = T$	X
$\mathcal{C}' = \infty$	X
$\mathcal{B}' = \mathsf{F}$	X

(a) GO

$\mathcal{B}$	F	T	·
$\mathcal{C} = \infty$	·	·	T
$NLA?On$	T	F	·
$NLB?On$	·	·	T
$\mathcal{E}' = T$		X	
$\mathcal{B}' = \mathsf{F}$		X	
$\mathcal{B}' = \mathsf{T}$	X		
$\mathcal{C}' = T$			X

(b) LOOP

$T \geq \mathcal{C} + 240$	T
$SC!Yellow(A)$	X
$\mathcal{E}' = T$	X

(c) TMOUT

$\mathcal{B}$	F
$T \geq \mathcal{E} + 10$	T
$SC!Yellow(A)$	X
$\mathcal{E}' = T$	X

(d) IDLE

Figure 8: Nighttime transitions

In nighttime mode, the situation is more complicated, since here the controller has to wait for a car to arrive on either road A or B. A first attempt is shown in Figures 10 and 11. Let us assume that $\mathcal{D}$ can take a third value distinct from A and B; we will denote this value as $\circledcirc$ and read it as "unassigned". Upon entering state BothRed, the direction is set to this value, and the controller waits for a car to appear on either road. This road will determine the next green phase.

Unfortunately, the transition LOOP in Figure 11(c) is not consistent: when cars arrive on roads A and B at the same time, the transition can not be taken, since there is no state where $\mathcal{D} = A$ and also $\mathcal{D} = B$. We could solve this problem by splitting LOOP into four separate transitions, one for each column. Then either direction could be nondeterministically chosen.

Still, this solution is not satisfactory: the traffic controller could then choose road A for the next green phase, although a car is waiting on road B, and vice versa.

Our second solution introduces yet another attribute, $\mathcal{P}$, which contains the previous direction. In the situation that cars wait on both roads A and B, the controller will chose direction $\mathcal{P}^{\times}$ for the next green phase. Thus, the controller is fair in the sense that it alternates the green phases of the two roads, when on both roads cars arrive frequently enough.

Figure 12 shows the modified LOOP transition.

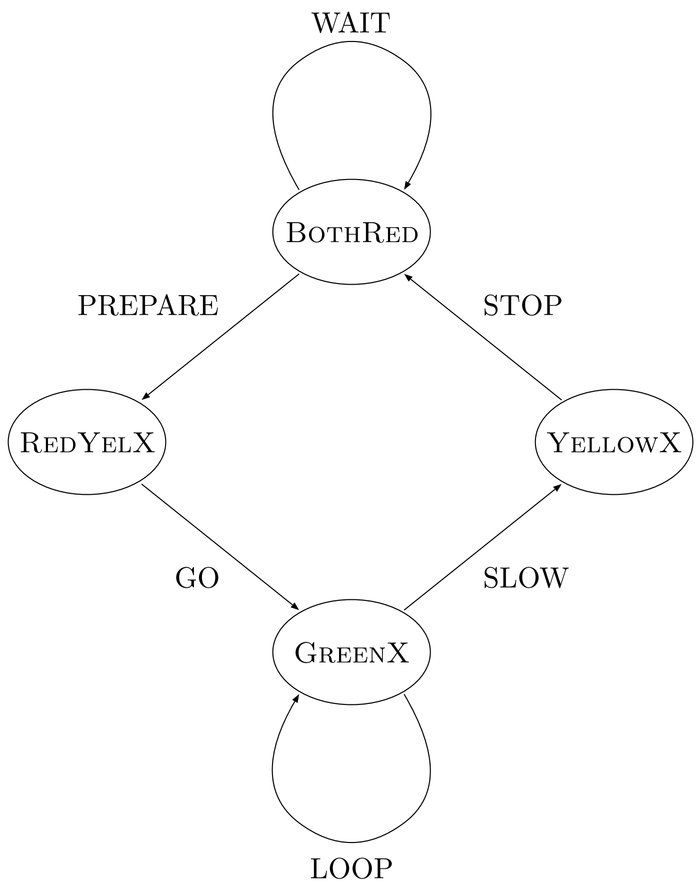

Figure 9: Transition diagram with symmetry

4.3 Complete Specification

Finally, we can specify the complete automaton. Figure 13 contains the transition structure; Figure 14 shows the transition definitions. Table 1 contains a list of the automaton's attributes. The transition tables are not minimal. In particular, the first two rows of Tables 14(a) and 14(d) could be omitted. Also, every row that consists only of don't care symbols can be removed.

The transition tables are assembled from the smaller tables presented in the previous paragraphs. In addition, conditions have been added as guards for normal operation: blinking mode must not be enabled ($Mode?Blink = \mathsf{F}$), and the signal controller must not send a failure signal ($SF?On = \mathsf{F}$). Finally, there are some conditions that determine whether daytime or nighttime mode is enabled.

Figure 15 shows the transitions that enter the exception behavior, when either blinking mode is enabled, or a lamp burned out. The transitions differ in how the signal controller should control the lamps when switching to failure mode: According to the informal specification, all lights should be switched to red for two seconds. That means:

- From state BOTHRED, no change is necessary.

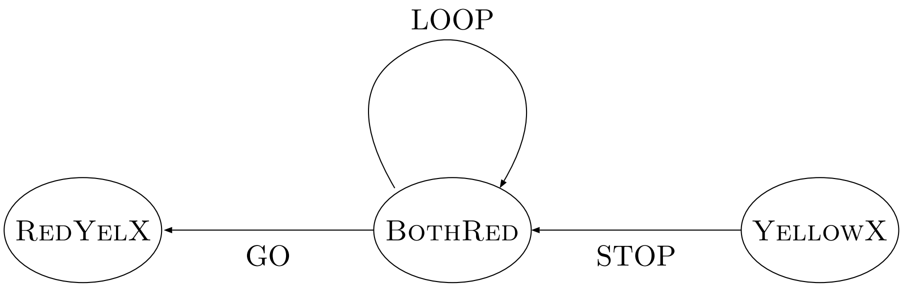

Figure 10: Nighttime diagram

$T \geq \mathcal{E} + 1$	T
$SC!Red(\mathcal{D})$	X
$\mathcal{D}' = \circledcirc$	X
$\mathcal{E}' = T$	X

(a) STOP

$\mathcal{D} \neq \circledcirc$	T
$SC!RedYel(\mathcal{D})$	X
$\mathcal{E}' = T$	X

(b) GO

$NLA?On$	T	·	·	·
$DLA?On$	·	T	·	·
$NLB?On$	·	·	T	·
$DLB?On$	·	·	·	T
$\mathcal{D}' = A$	X	X		
$\mathcal{D}' = B$			X	X

(c) LOOP

Figure 11: Inconsistent nighttime transitions

$NLA?On$	T	·	F	F	T	T	·	·
$DLA?On$	·	T	F	F	·	·	T	T
$NLB?On$	F	F	T	·	T	·	T	·
$DLB?On$	F	F	·	T	·	T	·	T
$\mathcal{D}' = A$	X	X						
$\mathcal{D}' = B$			X	X				
$\mathcal{D}' = \mathcal{P}^{\times}$					X	X	X	X

Figure 12: Fair nighttime transition "Stop"

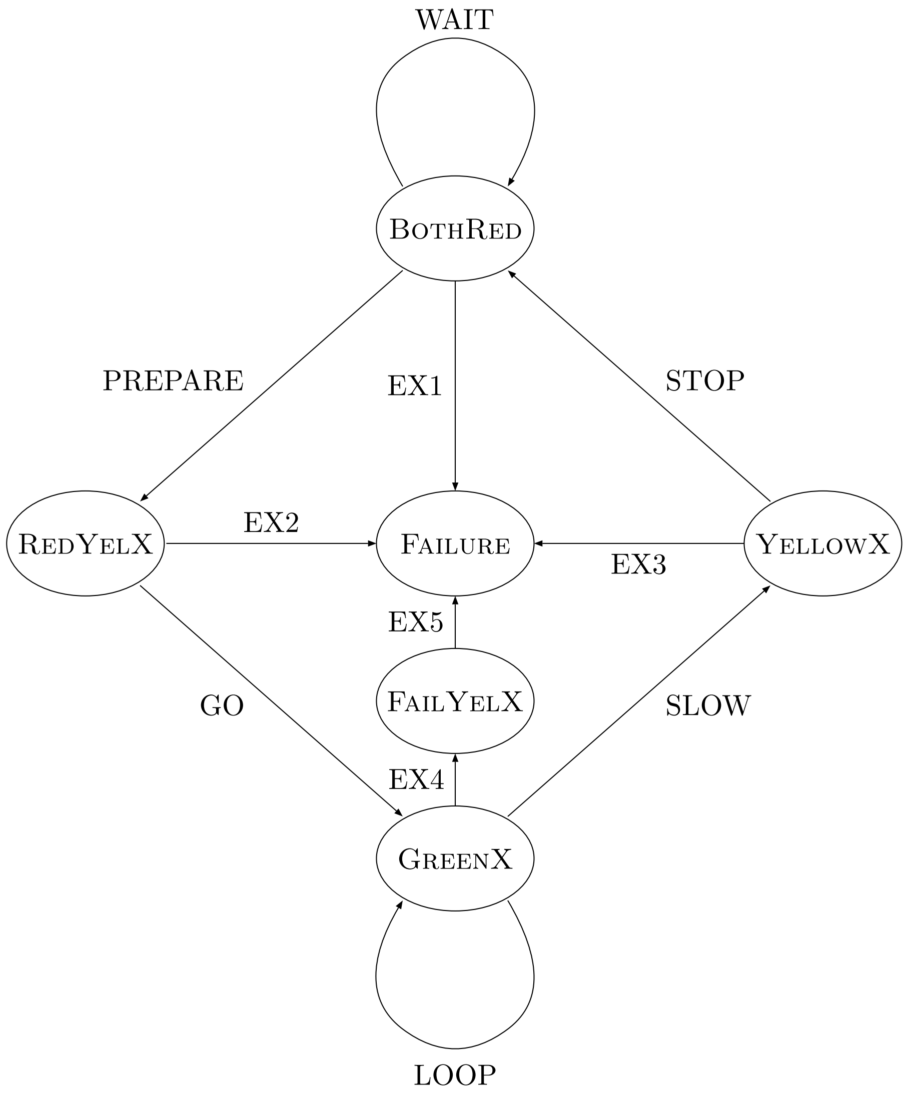

Figure 13: Control automaton

$Mode?Day$	T	·
$Mode?Night$	·	T
$Mode?Blink$	F	F
$LF?On$	·	·
$SF?On$	F	F
$T \geq \mathcal{E}+1$	T	T
$\mathcal{D} \neq \circledcirc$	T	T
$SC!RedYel(\mathcal{D})$	X	X
$\mathcal{E}' = T$	X	X

(a) PREPARE

$Mode?Night$	T	T	T	T	T	T	T	T
$LF?On$	F	F	F	F	F	F	F	F
$SF?On$	F	F	F	F	F	F	F	F
$\mathcal{D} = \circledcirc$	T	T	T	T	T	T	T	T
$NLA?On$	T	·	F	F	T	T	·	·
$DLA?On$	·	T	F	F	·	·	T	T
$NLB?On$	F	F	T	·	T	·	T	·
$DLB?On$	F	F	·	T	·	T	·	T
$\mathcal{D}' = A$	X	X						
$\mathcal{D}' = B$			X	X				
$\mathcal{D}' = \mathcal{P}^{\times}$					X	X	X	X

(b) WAIT

$Mode?Day$	T	·	·
$Mode?Night$	·	·	T
$Mode?Blink$	F	F	F
$LF?On$	·	T	F
$SF?On$	F	F	F
$T \geq \mathcal{E}+1$	T	T	T
$SC!Red(\mathcal{D})$	X	X	X
$\mathcal{E}' = T$	X	X	X
$\mathcal{D}' = \circledcirc$			X
$\mathcal{D}' = \mathcal{D}^{\times}$	X	X	
$\mathcal{P}' = \mathcal{D}$			X

(c) STOP

$Mode?Day$	T	·	·
$Mode?Night$	·	·	T
$Mode?Blink$	F	F	F
$LF?On$	·	T	F
$SF?On$	F	F	F
$T \geq \mathcal{E}+1$	T	T	T
$SC!Green(\mathcal{D})$	X	X	X
$\mathcal{E}' = T$	X	X	X
$\mathcal{C}' = \infty$			X
$\mathcal{B}' = \mathsf{F}$			X

(d) GO

$\mathcal{D} = A$	T	T	T	F	F	F
$Mode?Night$	T	T	T	T	T	T
$LF?On$	F	F	F	F	F	F
$SF?On$	F	F	F	F	F	F
$\mathcal{B}$	F	T	·	F	T	·
$\mathcal{C} = \infty$	·	·	T	·	·	T
$NLA?On$	T	F	·	·	·	T
$NLB?On$	·	·	T	T	F	·
$\mathcal{E}' = T$		X			X	
$\mathcal{B}' = \mathsf{F}$		X			X	
$\mathcal{B}' = \mathsf{T}$	X			X		
$\mathcal{C}' = T$			X			X

(e) LOOP

$Mode?Day$	T	·	·	·
$Mode?Night$	·	·	T	T
$Mode?Blink$	F	F	F	F
$LF?On$	·	T	F	F
$SF?On$	F	F	F	F
$\mathcal{B}$	·	·	F	F
$NLB?On$	T	·	·	·
$T \geq \mathcal{E}+45$	T	·	·	·
$T \geq \mathcal{E}+180$	·	T	·	·
$T \geq \mathcal{E}+10$	·	·	T	·
$T \geq \mathcal{C}+240$	·	·	·	T
$SC!Yellow(\mathcal{D})$	X	X	X	X
$\mathcal{E}' = T$	X	X	X	X

(f) SLOW

Figure 14: Control automaton transitions

	Type	Purpose
$\mathcal{E}$	$\mathbb{R}$	Time of state entry
$\mathcal{C}$	$\mathbb{R}$	Time of arrival of car on road $\mathcal{D}^\times$ (used in nighttime mode only)
$\mathcal{B}$	$\mathbb{B}$	Intersection is occupied by cars from road $\mathcal{D}$ (used in nighttime mode only)
$\mathcal{D}$	$\mathbb{D}$	Name of road for green phase
$\mathcal{P}$	$\mathbb{D} \setminus \{\circledcirc\}$	Name of road of previous green phase (used in nighttime mode only)

Table 1: Summary of control automaton attributes

- From RedYelX and YellowX, all lights are switched to red again.
- From state GreenX, the signal switches first to yellow to give the drivers on road $\mathcal{D}$ some time to slow down and then via FailYelX to red.

The exception behavior itself is specified in Figures 16 and 17. In state Failure, the controller waits for two seconds before it switches all signals off; then the yellow lights on road B start blinking.

When the blinking mode is deactivated (i.e. the mode is daytime or nighttime) and the signal controller does not send a failure signal, the failure mode is left again via transitions RESUME1 or RESUME2. State BothRed is entered again, and the variables $\mathcal{D}$ and $\mathcal{P}$ are set to the proper values for the operation mode.

5 Loop and Signal Controller

Besides the traffic controller itself, the traffic control system consists of a controller for the inductive loops, and a controller for the traffic signals. These two controllers form the control automaton's interface to the physical world. Compared to the traffic control automaton, their behavior is quite simple.

5.1 Loop Controller

The task of the loop controller is twofold:

- It checks the status of the inductive loops, groups the signals for opposing lanes together, and forwards the result to the control automaton.
- It checks whether any loop has been signaling for more than 10 minutes; if so, an error signal is forwarded to the control automaton.

$Mode?Blink$	T	·
$SF?On$	·	T
$\mathcal{E}' = T$	X	X

(a) EX1

$Mode?Blink$	T	·
$SF?On$	·	T
$SC!Red(\mathcal{D})$	X	X
$\mathcal{E}' = T$	X	X

(b) EX2 and EX3

$Mode?Blink$	T	·
$SF?On$	·	T
$SC!Yellow(\mathcal{D})$	X	X
$\mathcal{E}' = T$	X	X

(c) EX4

$T \geq \mathcal{E} + 1$	T
$SC!Red(\mathcal{D})$	X
$\mathcal{E}' = T$	X

(d) EX5

Figure 15: Entering failure mode

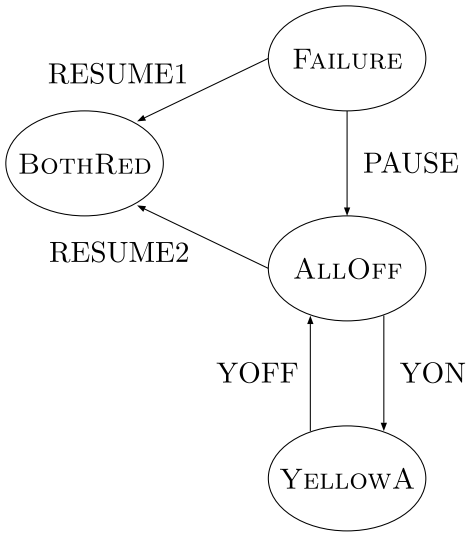

Figure 16: Failure states

$Mode?Blink$	T	·
$SF?On$	·	T
$T \geq \mathcal{E} + 2$	T	T
$\mathcal{E}' = T$	X	X
$SC!AllOff$	X	X

(a) PAUSE

$Mode?Day$	T	·
$Mode?Night$	·	T
$SF?On$	F	F
$SC!AllRed$	X	X
$\mathcal{E}' = T$	X	X
$\mathcal{D}' = \mathcal{D}^{\times}$	X	
$\mathcal{D}' = \circledcirc$		X
$\mathcal{P}' = \mathcal{D}$		X

(b) RESUME1 and RESUME2

$Mode?Blink$	T	·
$SF?On$	·	T
$T \geq \mathcal{E} + 1$	T	T
$SC!BYel$	X	X
$\mathcal{E}' = T$	X	X

(c) YON

$Mode?Blink$	T	·
$SF?On$	·	T
$T \geq \mathcal{E} + 1$	T	T
$SC!AllOff$	X	X
$\mathcal{E}' = T$	X	X

(d) YOFF

Figure 17: Failure mode

We specify the loop controller with a predicate that directly characterizes its input/output behavior. Let $LOOPS = \{NLA1, NLA2, DLA1, DLA2, NLB1, NLB2, DLB1, DLB2\}$ be the set of all channels from the inductive loops to the loop controller. Then the loop controller is specified as follows:

component LoopCtrl

inputs
$NLA1, NLA2, DLA1, DLA2, NLB1, NLB2, DLB1, DLB2, Time$
outputs
NLA, DLA, NLB, DLB, LF
spec

$$\begin{aligned}
NLA_i = On &\Leftrightarrow NLA1_i = On \vee NLA2_i = On \\
NLB_i = On &\Leftrightarrow NLB1_i = On \vee NLB2_i = On \\
DLA_i = On &\Leftrightarrow DLA1_i = On \vee DLA2_i = On \\
DLB_i = On &\Leftrightarrow DLB1_i = On \vee DLB2_i = On \\
LF_i = On &\Leftrightarrow (Time_i \geq 600) \wedge \\
&\qquad \bigvee_{c \in LOOPS} \quad (\forall j : Time_i - 600 \leq Time_j \leq Time_i \Rightarrow c_i = On)
\end{aligned}$$

The first lines of the specification define the interface of the loop controller; it is taken from Figure 2. The remaining lines define its behavior. The i-th output on channel NLA is equal to On, if at least one of the two sensors $NLA1$ or $NLA2$ detects a car at time i. The remaining sensor outputs are similar.

More interesting is formula that detects failures in the inductive loops. An error signal is supposed to be output in the i-th interval of stream LF, iff at least one of the eight sensors has been *On* since 10 minutes before the time in interval i. Our model cannot capture this requirement precisely. Instead, we demand that there is a sequence of intervals before the i-th interval, such that one of the sensors is *On* in each interval, and the duration of the sequence is at least 10 minutes. This corresponds to a sampling of the inputs from the sensors and consequently it is possible that a failure will be erroneously assumed when the sensors are *Off* between two readings.

A proper formalization of the interface to this kind of sensors would require continuous streams, defined over a dense time model. In [13], it has been demonstrated how Focus can be extended to continuous streams.

5.2 Signal Controller

The signal controller distributes the commands from the control automaton to the four traffic signals. Moreover, if any lamp of any signal is defect, an error signal is forwarded to the traffic control automaton.

The signal controller can be decomposed into five components working in parallel as shown in Figure 18. Four of the components control the four traffic lights, the fifth component detects signal lamp failures.

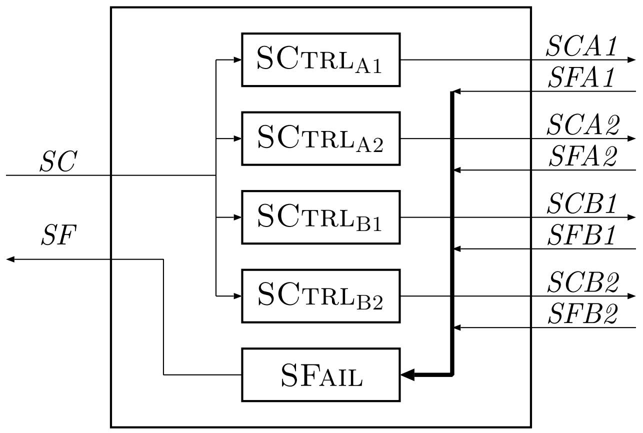

Figure 18: Decomposition of the signal controller

We specify these components as simple functional programs. We only give the specification for the first lamp controller $\text{SC}\textsc{trl}_{\text{A1}}$ and the failure detector; the other three lamp controllers are similar.

component $\text{SC}\textsc{trl}_{\text{A1}}$

inputs
  SC
outputs
  $SCA1$
spec
  $\mathit{SCA1} = f(SC)$ **where**

$$\begin{array}{rcl} f(\mathit{AllOff} \,\&\, r) & = & \mathit{Off} \,\&\, f(r) \\ f(\mathit{RedYel}(A) \,\&\, r) & = & \mathit{RedYel} \,\&\, f(r) \\ f(\mathit{Yellow}(A) \,\&\, r) & = & \mathit{Yellow} \,\&\, f(r) \\ f(\mathit{Green}(A)) \,\&\, r) & = & \mathit{Green} \,\&\, f(r) \\ f(a \,\&\, r) & = & \mathit{Red} \,\&\, f(r) \\ & & \textbf{if}\, a \notin \{\mathit{AllOff}, \mathit{RedYel}(A), \mathit{Yellow}(A), \mathit{Green}(A)\} \end{array}$$

component SFAIL

inputs
  $SFA1, SFA2, SFB1, SFB2$
outputs
  SF
spec
  $SF = f_{(\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F})}(SFA1, SFA2, SFB1, SFB2)$ **where**

$$f_{(b_1,b_2,b_3,b_4)}(a_1 \,\&\, r_1,\, a_2 \,\&\, r_2,\, a_3 \,\&\, r_3,\, a_4 \,\&\, r_4) = (b'_1 \vee b'_2 \vee b'_3 \vee b'_4) \,\&\, f_{(b'_1,b'_2,b'_3,b'_4)}(r_1, r_2, r_3, r_4)$$

where

$$b'_1 = b_1 \vee a_1,\ b'_2 = b_2 \vee a_2$$
$$b'_3 = b_3 \vee a_3,\ b'_4 = b_4 \vee a_4$$

The complete specification for the signal controller is then the network of the lamp control and failure components:

component SIGNALCTRL

inputs
  $SC, SFA1, SFA2, SFB1, SFB2$
outputs
  $SCA1, SCA2, SCB1, SCB2, SF$
spec
  $\mathrm{SCTRL_{A1}}(SC, SCA1)$
  $\mathrm{SCTRL_{A2}}(SC, SCA2)$
  $\mathrm{SCTRL_{B1}}(SC, SCB1)$
  $\mathrm{SCTRL_{B2}}(SC, SCB2)$
  $\mathrm{SFAIL}(SFA1, SFA2, SFB1, SFB2, SF)$

6 Model Checking

In the previous sections, we developed a formal specification for the traffic light system starting from an informal, textual description. It is not obvious, however, whether these specifications indeed describe the intended system. There are several reasons why our specification could be flawed:

- The specification could itself be wrong, for instance because of typing errors.
- The specification, while syntactically correct, might not match the informal requirements, because we misunderstood them, or because they themselves are contradictory or incomplete.

In this section, we demonstrate how automatic verification based on a model checking tool can be used to validate specifications. In model checking, it is verified whether a property—usually formulated in a temporal logic—holds for a given model. The model is usually defined as a state transition system.

Here we employ model checkers to show that the traffic light system fulfills certain standard poperties that are expected of a traffic light. Such properties are for example:

- that at no time signals of both roads will be set to green;
- that, when a signal is set to yellow, the controller will subsequently switch the signal to red.
- that, when a car waits long enough, it will be allowed to cross the intersection.

Proving these properties will give us more confidence in our controller. When failing to prove a property, the model checker will return a trace of an example execution that violates the property. Of course, we need to make sure that both our translation into the input language of the model checker and the formalization of the property in temporal logic is plausible.

In the rest of this section, we explain how the traffic light system is translated into the model checker's input language, and how properties such as those mentioned above can be formalized. Since most of the complexity of the traffic light system is within the traffic control automaton, we omit the loop and lamp controllers, and only attempt to validate the traffic controller's behavior at its interfaces to the other components.

We used the model checker SMV [16], since it is freely available, reasonably efficient, and has a rather simple input language.

6.1 Specification Translation

Since SMV, like most model checkers, can only verify closed systems, the SMV model consists of two components:

- The traffic controler itself; its state space consists of the control state (the vertices of Figure 13, and the attributes of Table 1. In addition, we introduce a state attribute `trans` that contains the name of the transition to be followed; this attribute serves only for more readable execution traces.
- The environment; its state space consists of the current mode (day, night, constant-time), the current values of the inductive loop sensors and the current time.

SMV does not directly support communication over channels. Because of our synchronous execution model, however, we can simply use shared variables for communication.

```
  next(trans) = mcprepare &

-- State change:
  ctrl = BothRed & next(ctrl) = RedYelX &

-- Enabledness:
  ( dir=DirA & mode=Day & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) |
    dir=DirA & mode=Nite & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) |
    !(dir=DirA) & mode=Day & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) |
    !(dir=DirA) & mode=Nite & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) ) &

-- Actions:
  (dir=DirA & mode=Day & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) ->
    ( next(sc) = RedYelA )) &
  (dir=DirA & mode=Nite & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) ->
    ( next(sc) = RedYelA )) &
  (!(dir=DirA) & mode=Day & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) ->
    ( next(sc) = RedYelB )) &
  (!(dir=DirA) & mode=Nite & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) ->
    ( next(sc) = RedYelB )) &

  (!( dir=DirA & mode=Day & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) |
      dir=DirA & mode=Nite & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) |
      !(dir=DirA) & mode=Day & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot) |
      !(dir=DirA) & mode=Nite & !(mode=Blink) & !(sf) & d >= 1 & !(dir=Bot)
   ) -> next(sc) = sc)
  &
  next(busy) = busy &
  next(car) = car &
  next(waiting) = waiting &
  next(dir) = dir &
  next(prev) = prev
```

Figure 19: Translation example

The translation of the transition diagrams and tables is then straightforward. As an example, Figure 19 shows the translation of the transition PREPARE (see Figure 14).

For each transition, the translation essentially consists of four parts:

- The first line sets the state attribute `trans` to the transition name;
- The second line changes the control state;
- The next lines check whether at least one condition column is true;
- The fourth part contains the effect of the actions. In addition, it also leaves unmodified all attributes not explicitly modified by an action.

Because the input language of SMV does not allow functions to be defined, the translation contains explicitly distinguishes the case that $\mathcal{D} = A$ and $\mathcal{D} \neq A$. Thus, there are twice as many entries in the translation as columns in the original tables.

The translation of the real-time aspects of the specification is somewhat problematic. Since SMV only supports finite subsets of the natural numbers, we could not make use of a monotonically increasing time variable T. Instead, the translation uses time values that are relative to the time a state was entered. The time difference between the entering of a state and the transition is in the translation denoted by the attribute `d`. Consequently, comparisons of the form $T' \geq \mathcal{E} + k$ are now translated into formulas of the form `d >= k`. Since the traffic controller has a single thread of control, this modification does not change the controller's behavior. However, there is now the additional assumption that the system has a minimum speed, since the values of `d` and the constants are bounded.

6.2 Property Formalization

The model checker SMV uses the branching-time temporal logic CTL (Computation Tree Logic) for the specification of properties. An introduction to this logic can be found in [4]; here we just give a few examples. We assume that ϕ, ψ are a propositional expression over the variables that define the system's state space. Then frequently used CTL idioms are:

- $AG\phi$ is true, iff ϕ holds invariantly for all executions of the system. Similarly, $EG\phi$ holds, iff there is one execution where ϕ is invariantly true.

  For example, we will expect that our traffic light has an execution, where all signals are forever switched to red: This situation occurs in nighttime mode, when no car ever arrives on either road. However, this will not be the case for all executions. Thus, we expect $AG(\mathit{ca.ctrl} = \mathit{Red})$ to be false, but $EG(\mathit{ca.ctrl} = \mathit{Red})$ to be true for our system.

```
(nla & !(ctrl = GreenX & dir = DirA) -> next(nla) = 1) &
(nlb & !(ctrl = GreenX & dir = DirB) -> next(nlb) = 1) &
next(d) > 0 & next(d) < 256
next(m) = m & next(lf) = 0 & next(sf) = 0  &
```

Figure 20: Environment assumptions

- $AF\phi$ is true, when in all executions after a finite number of transitions a state is reached where ϕ is true. Similarly, $EF\phi$ means that this is true for at least one execution. The formulas $AX\phi$ and $EX\phi$ express the stronger property that system states where ϕ hold must be reached after exactly one transition.

  Invariants and reachability can be combined. For example, $AGAF\phi$ states that ϕ is true infinitely often.

- $AG(\phi \Rightarrow AF\psi)$ is an idiom for reactivity properties. It means that whenever ϕ holds in a system's execution, after a finite number of transitions a state will be reached where ψ holds. Again, variations of this property can be formed by writing EG and EF.

Some of the properties we are interested in only hold when the sytem's environment—the car sensors and failure detectors—behaves in a certain way. Assumptions on the environment, however, can in general not be expressed in CTL. Therefore we model the environment itself with a restricted transition relation. Our environment transitions are as follows:

- Whenever a car arrives at the intersection, it remains there until the traffic light turns green.
- Neither the inductive loops nor the signal lamps fail.
- The time progress between transitions is bounded.

In addition, for some properties we restrict changes of the operation mode. Figure 20 shows how these assumptions are encoded in SMV.

Simple behavior properties. We verified some small formulas to ensure that our specification indeed behaves in a way expected of traffic light controllers. The formulas describe the cyclic behavior of the controller. Here are some examples:

```
EG (EF (ctrl = RedYelX & dir = DirA))

EG (EF (ctrl = RedYelX & dir = DirB))
```

```
AG (EF (ctrl = RedYelX & dir = DirA))

AG (EF (ctrl = RedYelX & dir = DirB))

AG (AF (ctrl = RedYelX & dir = DirA))

AG (AF (ctrl = RedYelX & dir = DirB))
```

The first two properties show that there is at least one execution of the system, where the controller can always start a green phase for each road. The next two properties show that this holds for all executions—consequently, there is no execution where the controller deadlocks. The last two formulas show that there are infinitely many green phases for each road.

The first four properties hold for both daytime and nighttime mode. This is not true for the last two properties: They are invalid if no car ever arrives on road A or road B, respectively.

Another set of formulas can be used to show that also the sequential operation of our controller is correct. For example, the property

```
AG (ca.ctrl = RedYelX -> AX (ca.ctrl = GreenX))
```

can be verified to show that when starting a green phase and the signal lamp shows red and yellow, the next state must be such that the signal lamp switches to green.

Utility property. A simple utility property that the traffic light is expected to fulfill is that when a car arrives at the intersection, it may cross it when it waits long enough. Above, we have seen that this property already holds for daytime mode, since there are infinitely many green phases for each road. For nighttime mode, the situation is more complex.

We formulate the property as follows:

```
AG (nlb -> AF (ca.ctrl = GreenX & ca.dir = DirB))
```

Whenever a car is detected by the near induction loop on road B, after some finite but unbounded time the controller will signal green to road B.

Interestingly, this property does not hold when for the original informal specification. There it is required that in night mode, only the distant loops are checked for arriving cars. The specification presented on page 14, however, checks both the near and the distant inductive loops. With this modification, the verification is successful.

6.3 Remarks

The translation of the specification results in an SMV program of slightly more than 800 lines. In order to avoid typing errors in the translation, we generated most of the file automatically with a small PERL [18] script from a log file generated by compiling the LaTeX source of this report. Only the declarations of the state attributes, and the restrictions on the environment transitions had to be added by hand.

Because of the size of the transition relation, most of the time used for verification is for building the transition relation. The verification of properties such as the ones presented above only takes a few seconds.

A typical verification takes about seven to eight minutes on a Sparc Ultra 1 with 64 MB main memory, of which six minutes are used to build the transition relation.

7 Summary

In this report, we specified a simple traffic control system with three components: An interface component for the inductive loops which detects arriving and waiting cars, an interface component for the four signals, and a central component which controls traffic flow and handles exceptional situations caused by failures in the inductive loops and signal lamps.

Because of the various operation modes of the system, the central control component turned out to be rather complex. Therefore, we specified this part of the controller not directly by logical formulas like the two interface components, but modeled it as a state transition system. The transitions are pairs of conditions and actions and were specified in a tabular style for readability. Our approach to tables is inspired by the decision tables used in RSML [12]. For the complex transition conditions in the traffic controller, these tables are more concise than other table representations [14, 17].

However, while the transition definitions for normal operation of the traffic light fit on a single page (see Figure 14), it is not obvious whether they correctly implement the given informal requirements.

Therefore, we translated our specification to the input language of the model checker SMV; we could then use the model checker to prove several theorems that describe the correct traffic light behavior. We believe that this use of automated verification to show the consistency of two views of a system specification—the state transition system and the temporal logic formulas—is a promising extension to conventional review processes.

We used an ad hoc translation of the specification into SMV and did not give a formal proof of correctness. The synchronous stream model and the state transition semantics we used is, however, similar to the approach used in the prototypical CASE tool AutoFocus [10]; this tool could be used for rapid prototyping [8], and for a more formal approach to verification [9].

Acknowledgements. This work is based on a case study provided by Prof. Dr. Georg Färber and his colleagues at the institute for process control of the TU München. We thank Thomas Kolloch and Katharina Spies for many stimulating discussions, and Peter Scholz for his helpful comments on this work.

References

[1] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus—Revised Version. Technical Report TUM-I9202-2, Institut für Informatik, Technische Universität München, 1993.

[2] M. Broy and K. Stølen. Focus on system development. Book manuscript, 1997.

[3] Manfred Broy. The specification of system components by state transition diagrams. Technical Report TUM-I9729, Institut für Informatik, Technische Universität München, 1997.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J.Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Computer Science*, pages 428–439, 1990.

[5] Max Fuchs and Ketil Stølen. A formal method for hardware/software co-design. Technical Report TUM-I9517, Institut für Informatik, Technische Universität München, 1995.

[6] R. Grosu, C. Klein, B. Rumpe, and M. Broy. State transition diagrams. Technical Report TUM-I9630, Institut für Informatik, Technische Universität München, 1996.

[7] R. Grosu and B. Rumpe. Concurrent timed port automata. Technical Report TUM-I9533, Institut für Informatik, Technische Universität München, 1995.

[8] F. Huber and B. Schätz. Rapid prototyping with autofocus. In A. Rennoch A. Wolisz, I. Schieferdecker, editor, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch*, pages 343–352, 1997.

[9] F. Huber, B. Schätz, and G. Einert. Consistent graphical specification of distributed systems. In *FME'97*, Lecture Notes in Computer Science. Springer, 1997. To appear.

[10] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus—A Tool for Distributed Systems Specification . In Bengt Jonsson and Joachim Parrow, editors, *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*. Springer, 1996.

[11] Technische Universität München, Lehrstuhl für Prozeßrechner. Anforderungsspezifikation eines Ampel-Controllers, 1994.

[12] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[13] O. Müller and P. Scholz. Functional specification of real-time and hybrid systems. In *Proc. Hybrid and Real-Time Systems, Grenoble*, number 1201 in Lecture Notes in Computer Science. Springer, 1997.

[14] D. L. Parnas. Tabular representation of relations. Technical Report 260, CRL, October 1992.

[15] B. Schätz and K. Spies. Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik. Technical Report TUM-I9529, Institut für Informatik, Technische Universität München, 1995.

[16] SMV, 1992. Availabe at `http://www.cs.cmu.edu/~modelcheck/`.

[17] K. Spies. Funktionale Spezifikation eines Kommunikationsprotokolls. Technical Report TUM-I9414, Institut für Informatik, Technische Universität München, 1994.

[18] L. Wall and R. L. Schwartz. *Programming in in perl.* O'Reilly & Associates, 1990.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

Liste aller erschienenen Berichte von 1990-1994
auf besondere Anforderung

342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids
342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law
342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space
342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes
342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems
342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter
342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology
342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies
342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness
342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design
342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System
342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm
342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming
342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks

Reihe A

342/15/95 A	Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs
342/16/95 A	Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik
342/17/95 A	Georg Stellner: Using CoCheck on a Network of Workstations
342/18/95 A	Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications
342/19/95 A	Thomas Schnekenburger: Integration of Load Distribution into ParMod-C
342/20/95 A	Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication
342/21/95 A	Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control
342/22/95 A	Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes
342/23/95 A	Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation
342/24/95 A	M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems
342/01/96 A	Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries
342/02/96 A	Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs
342/03/96 A	Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes
342/04/96 A	Thomas Huckle: Efficient Computation of Sparse Approximate Inverses
342/05/96 A	Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification
342/06/96 A	Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components
342/07/96 A	Richard Mayr: Some Results on Basic Parallel Processes
342/08/96 A	Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht
342/09/96 A	P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme
342/10/96 A	Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFSLib – A File System for Parallel Programming Environments

Reihe A

342/11/96 A Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions

342/12/96 A Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete

342/13/96 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks

342/14/96 A Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project

342/15/96 A Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus

342/16/96 A Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen

342/17/96 A Richard Mayr: Model Checking PA-Processes

342/18/96 A Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States

342/01/97 A Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsf"ahigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken

342/02/97 A Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations

342/03/97 A Bernhard Mitschang: Technologie f"ur Parallele Datenbanken - Bericht zum Workshop

342/04/97 A nicht erschienen

342/05/97 A Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik

342/06/97 A Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump

342/07/97 A Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming

342/08/97 A Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement

342/09/97 A Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic

342/10/97 A Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus

342/11/97 A Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity

342/12/97 A Christian B. Czech: Architektur und Konzept des Dycos-Kerns

Reihe A

342/13/97 A	Jan Philipps, Alexander Schmidt: Traffic Flow by Data Flow
342/14/97 A	Norbert Fröhlich, Rolf Schlagenhaft, Josef Fleischmann: Partitioning VLSI-Circuits for Parallel Simulation on Transistor Level
342/15/97 A	Frank Weimer: DaViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL-Programmen
342/16/97 A	Niels Reimer, Jürgen Rudolph, Katharina Spies: Von FOCUS nach INSEL - Eine Aufzugssteuerung
342/17/97 A	Radu Grosu, Ketil Stølen, Manfred Broy: A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B	Wolfgang Reisig: Petri Nets and Algebraic Specifications
342/2/90 B	Jörg Desel: On Abstraction of Nets
342/3/90 B	Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems
342/4/90 B	Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme
342/1/91 B	Barbara Paech1: Concurrency as a Modality
342/2/91 B	Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox -Anwenderbeschreibung
342/3/91 B	Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen
342/4/91 B	Werner Pohlmann: A Limitation of Distributed Simulation Methods
342/5/91 B	Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis
342/6/91 B	Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme
342/7/91 B	W. Reisig: Concurrent Temporal Logic
342/1/92 B	Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support
342/2/92 B	Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen
342/1/93 B	Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung
342/2/93 B	Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick
342/1/94 B	Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS