ELSEVIER Science of Computer Programming 29 (1997) 235–258

Science of Computer Programming

The I²C-bus in discrete-time process algebra

S.H.J. Bos, M.A. Reniers*

Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, Netherlands

Abstract

Using discrete time process algebra with relative timing, a model for the I²C-bus is designed. The model of the I²C-bus is divided into three parts: a model for the bus lines, a model for the master interfaces and a model for the slave interfaces. The model of the bus lines is based on a model for a wired-AND. For the models of the interfaces, the approach is to start from a high level bus protocol and refine it step by step. First, a single master without timing constraints is considered. Then the model is adapted to deal with the timing constraints. Then also the restriction to a single master is relaxed. It turns out that the model for the slave interfaces can be based on the model for the master interfaces. The use of the model obtained is discussed and illustrated. © 1997 Published by Elsevier Science B.V.

Keywords: I²C-bus; Master; Slave; Interface; Serial data line; Serial clock line; Timing requirements; Discrete-time process algebra; Relative timing

1. Introduction

The *I²C-bus* is a bidirectional bus for *Inter-IC* control [9]. Devices on the I²C-bus can be microprocessors, RAM, LCD-drivers, I/O-ports, etc. All I²C-bus compatible devices incorporate an on-chip interface which allows them to communicate with each other via the I²C-bus without additional external interfacing.

For devices communicating with each other on the bus, a communication protocol is defined which avoids all possibilities of confusion, data loss and blockage of information. The timing aspects form an important part of the communication protocol of the bus.

For the exchange of information between the devices connected to the I²C-bus it has two bidirectional wires, called *serial data* (*SDA*) and *serial clock* (*SCL*). The

*Corresponding author.

PII S0167-6423(96)00036-6

devices can be considered as masters or slaves when performing data transfers. During a data transfer there is exactly one master and one slave; a master is the device which initiates a data transfer and generates the clock signals which permit that transfer.

The I^2C-bus is widely used in mass electronics products because of several advantages it offers. Firstly, it can be implemented in a very cheap way; the wires can be made by exploiting certain properties of the electronic circuitry, not demanding additional logic. Secondly, it serves as a standard for specification and design of a wide variety of ICs such as microprocessors, RAMs, LCD-drivers, I/O-ports, etc. Moreover, the fact that it is a two-wire bus instead of 8, 16 or 32, makes it very attractive since the layout and routing of such wide busses is much harder. Finally, there are ongoing developments of new protocols, notably ACCESS.bus [1], which extend I^2C-bus.

The interoperability of the many versions and types of I^2C-bus components from different vendors is an important issue. Much attention has already been paid to the electrical aspects of the interoperability problem (value of pull-up resistors, etc.). But the timing aspects are more difficult to describe rigorously and therefore this paper presents an explicit rigorous model of the I^2C-bus and its interface with the devices connected to it. The availability of such a model is important for checking conformance of new implementations with respect to the standard. This model can be used for checking properties such as freedom of deadlocks of implementations.

To model the I^2C-bus and its interfaces, we will discuss a discrete time process algebra [4, 3] with relative timing with some additional features. Since the timing aspects in the protocol for the I^2C-bus is the main concern within the specification and verification of the processes to be designed, the choice for a timed process algebra is straightforward. We choose for a discrete time version rather than a real time one, because all specified delays imposed by the timing constraints are multiples of a certain unit. Furthermore, the motivation for considering relative timing instead of absolute timing is based on the fact that all timing constraints involve pairs of transitions; each second transition of such a pair is allowed to occur a specified amount of time after the first transition has taken place.

Using discrete time process algebra, we will design processes that model the behaviour of the bus itself, and of its interfaces with the devices connected to the bus. Since the task of the interface is to translate instructions from the device to bus transitions and vice versa, it must communicate with the device on the one hand and with the bus on the other. Communications with the bus are needed to accomplish the clamping, releasing and polling of the bus lines. Communications with the device contain the instructions from the device. We make a distinction between interfaces for master devices and for slave devices, since their behaviour is different. The design of the master and slave devices is considered beyond the scope of this paper, because of the enormous variety of applications that can be implemented by master and slave devices.

The model given in this paper can be used when designing devices to be connected to the I^2C-bus. It is of uttermost importance that the interface between a device and the bus is well-understood. It is our firm belief that a formal specification of such an interface improves this understanding considerably. Also, if a formal specification of

the device is provided, there are possibilities to reason formally about the device being connected to the bus, i.e., their interaction can be considered.

Another, minor, reason for this research is that, with the recent appearance of [3–5], discrete-time process algebra seems to have reached a decent state of maturity. This application can therefore also be seen as a case study in the practical application of discrete-time process algebra.

Structure of the paper: The paper is structured as follows. The Sections 2 and 3 provide a short introduction to the I^2C-bus and the relative discrete time process algebra, respectively. In Section 4, a model for the bus lines is given. Then, in Section 5, an interface is designed for a master device in a single-master environment. The approach is to take a high level bus protocol and refine this protocol step by step until an interface is obtained. In Section 6, the interface for a single master is extended to function in a multi-master environment. After having designed those master interfaces, we give an interface for a slave device in Section 7. This concludes the construction of a model for the I^2C-bus in relative discrete time process algebra. The following section deals with the usage of the model presented in this paper. Some conclusions and suggestions for further research are given in Section 9.

2. The I^2C-bus

In this section we will give a concise introduction to the I^2C-bus; for a more thorough description and explanation we refer to [7–9].

The I^2C-bus is used to interconnect integrated circuits. Devices communicating with each other on the bus must have some form of protocol which avoids all possibilities of confusion, data loss and blockage of information. Fast devices must be able to communicate with slow devices. The system must not depend on the devices connected to it. A procedure has to be devised to decide which device will be in control of the bus and when.

For the exchange of information between the devices connected to it, the I^2C-bus has two wires along which information is carried between the devices connected to the bus: the *SDA* line, that carries the data signal and the *SCL* line along which clock pulses are transmitted.

Each device is recognized by a unique address and can operate as either a transmitter or a receiver, depending on the function of the device. Devices can also be considered as masters and slaves when performing data transfers. A master is the device which initiates a data transfer on the bus and generates the clock signals that permit that transfer.

The I^2C-bus is a multi-master bus: more than one device capable of controlling the bus can be connected to it. As a consequence more than one master could try to initiate a data transfer at the same time. To avoid the chaos that might ensue from such an event, an arbitration procedure has been developed. For a description of this procedure we refer to Section 6.4.

Both *SDA* and *SCL* are bidirectional lines. When the bus is free, both lines are high. Devices are capable of clamping and releasing the wires. If one of the devices connected to the bus clamps a line, this line will become low and only if all devices have released a line, it will become high again. A line behaving in such a way is called a wired-AND. A wired-AND is an electronic building block with N inputs and a single output that computes the logical N-ary AND function. The low-to-high and high-to-low transitions of the wire are not instantaneous; they require the passage of a certain rise and fall time respectively. During such a transition the voltage level on the wire is not stable. Apart from rise and fall times, which are related to the physical properties of the bus wires, the specification for the bus will contain no timing constraints. Indeed all other timing constraints have to do with the protocol that prescribes how the bus wires should be manipulated and are therefore the responsibility of the interfaces that connect the devices to the I^2C-bus. The behaviour of the wires consists solely of alternating high-to-low and low-to-high transitions on both lines. These transitions are always responses to the clamping and releasing actions initiated by the devices connected to the bus. The duration of the high and low level periods of the wires are determined by the devices that clamp and release the bus lines and the timing constraints imposed by the bus protocol.

Within the procedure of the I^2C-bus, unique states arise which are defined as *start* and *stop* conditions. A high-to-low transition on the *SDA* line while the *SCL* line is high indicates a start condition. A low-to-high transition on the *SDA* line while the *SCL* line is high defines a stop condition. Start and stop conditions are always generated by a master. A bus session starts with a start condition and ends with a stop condition. The bus is considered to be free when there is no bus session. In between the start and stop condition the following events take place: a slave is addressed, the addressed slave acknowledges, data bytes are transmitted in both directions and acknowledged byte-wise. The sending (receiving) of a byte is achieved by sending (receiving) eight bits serially. Obviously this complicates the bus protocol considerably. In order to restrict the size of the problem, we will have the device take care of addressing and byte-wise data transfer. Thereby we shift some complexity from the interface to the devices, which are beyond the scope of this paper. With these restrictions a data transfer only consists of sending and receiving bits.

One clock pulse is generated for each data bit transferred. The data on the *SDA* line must be stable during the high period of this clock pulse. The sending of a bit with value 0 is now achieved by clamping (if necessary) the *SDA* line and generating a clock pulse. Likewise, a bit with value 1 is sent by releasing (if necessary) the *SDA* line and then generating a clock pulse. Receiving a bit is achieved by generating a clock pulse and then polling the *SDA* line to determine whether a 0 or a 1 has been received.

The timing constraints involved in the I^2C-bus are discussed next. There are only two timing constraints dealing with a maximum delay, i.e., they express that an event takes at most a certain amount of time. These are the *rise time tR* and *fall time tF* of the *SCL* line. These express the maximum time needed for the *SCL* line to perform a low-to-high and a high-to-low transition respectively. All other timing constraints are

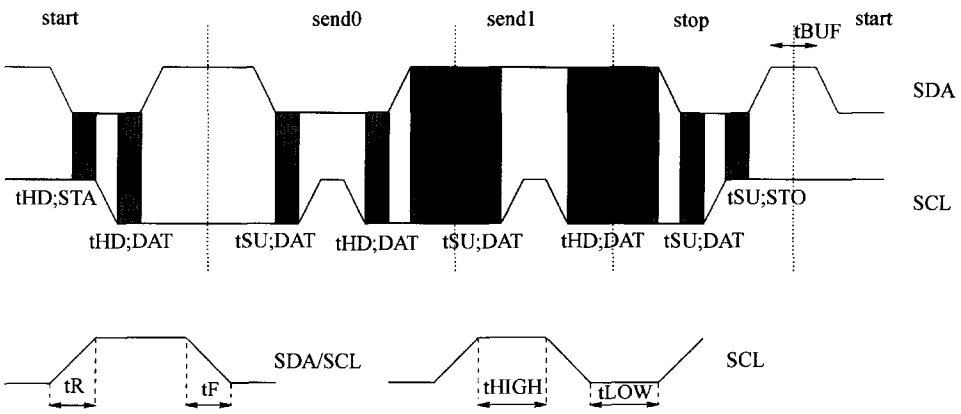

Fig. 1. Definition of timing constraints.

dealing with minimum delays, i.e., they express that an event takes at least a certain amount of time. There is only one minimum timing constraint which is related solely to the *SDA* line: *tBUF* denotes the *bus free time*, i.e., the minimal time needed for the *SDA* line to be high between a *stop* and a *start* condition. The timing constraints *tLOW* and *tHIGH* denote the *clock low* and *clock high* period, respectively. These are dealing solely with the *SCL* line. The remaining timing constraints deal with both the *SDA* and *SCL* line. There are three timing constraints for set-up times: the *data set-up time* (*tSU;DAT*), the *set-up time for a start condition* (*tSU;STA*), and the *set-up time for a stop condition* (*tSU;STO*). Besides these set-up times there are also two hold times: the *data hold time* (*tHD;DAT*) and the *hold time for a start condition* (*tHD;STA*). In Fig. 1 the timing constraints are depicted in relation to both the *SDA* and *SCL* lines. The names of the timing constraints are taken from [9].

In the previous part of this section the protocol on the I^2C-bus is described. In this protocol the communication primitives have been discussed. We described how these primitives are used in relation to each other to build a complete bus session. Apart from this protocol for communications on the bus, there should also exist another protocol between masters and slaves, which we will call the "meta-protocol". This meta-protocol describes the communication behaviour between masters and slaves. The meta-protocol states that arbitration is not allowed between a start condition and a data bit, a stop condition and a data bit, and a start condition and a stop condition [9]. For a more thorough explanation of the meta-protocol we refer to [7]. For our model, it is important that *start* and *stop* signals never will be involved in the arbitration procedure.

3. Discrete time process algebra with relative timing

In this section, we will introduce the process algebra used for modelling the I^2C-bus in this paper. Such an algebraic theory is given by a signature, i.e., a set of constant

and function symbols, and a set of equations (often called axioms). The process algebra is parameterized by a set A of constants and a partial, commutative and associative function γ. We will instantiate the set A of constants and the communication function γ later on. Using the constant and function symbols, terms can be constructed. These terms represent processes. The axioms express which processes are to be considered equal. The axioms for this process algebra are given in Appendix A. In the axioms also auxiliary operators ν_{rel}, $\lfloor\ \rfloor^{\omega}$, $\between$, and $|$ appear. The axioms can also be found in [2–4, 7]. Also for other issues concerning discrete time process algebra we refer to these papers.

We only list the operators that play a role in the main text of this paper. The signature consists, among others, of the following constant and function symbols:

1. the special constants $\underline{\delta}$, $\underline{\tau}$, δ and τ;
2. for $a \in A$, the constants $\underline{a}$ and a;
3. the binary operators $+$, $\cdot$, and $\between$;
4. the unary operators σ_{rel}, ρ_f, ∂_H, τ_I, π_{tf}, and Θ_C for $f: A \to A$, and $H, I, C \subseteq A$;

The special constants $\underline{\delta}$ and $\underline{\tau}$ denote deadlock and the silent step in the current time slice, respectively. The constants δ and τ are the delayable versions thereof; they can occur in any time slice.

For our application, we assume that the set A contains actions $s_{ch}(v)$, $r_{ch}(v)$, and $c_{ch}(v)$ for a channel ch and a datum v. These actions denote the sending, reception, and communication of datum v along channel ch. For $a \in A$, the action $\underline{a}$ denotes the execution of a in the current time slice, and the action a denotes the execution of a in any time slice. The execution of atomic actions and inaction is considered to be instantaneous.

The binary operators $+$ and $\cdot$ denote *alternative* and *sequential composition*, respectively. The alternative composition of the processes x and y is the process that executes process x or process y, but not both. We have the time factorization assumption, i.e., passage of time as such cannot determine a choice. The sequential composition of the processes x and y is the process that first executes x, and upon termination thereof starts with the execution of process y.

With the *unit delay* operator processes can be delayed one time slice: the process $\sigma_{\mathrm{rel}}(x)$ denotes the process x starting in the next time slice. In order to express that a process x has to be delayed for $n \geqslant 0$ time slices, we introduce the shorthand $\sigma^n_{\mathrm{rel}}(x)$ for the n times repeated application of σ_{rel}.

Two processes can be composed in parallel with the *parallel composition* operator $\between$. The process $x \between y$ interleaves the behaviours of x and y and tries to communicate send actions with receive actions of the same datum along the same channel, but is forced to synchronize on time steps, i.e., a clock tick is executed by both processes at the same time or, if this is not possible, the process deadlocks. In our application the communication function γ is defined as follows $\gamma(s_{ch}(v), r_{ch}(v)) = c_{ch}(v)$ and $\gamma(r_{ch}(v), s_{ch}(v)) = c_{ch}(v)$. For all other cases γ is undefined.

The *renaming* operator ρ_f is used to rename atomic actions into atomic actions according to the mapping $f: A \to A$. The *encapsulation* operator is a special renaming

operator where all actions given by the set H ($H \subseteq A$) are renamed into δ. This operator is used to block the execution of unwanted actions. The *abstraction* operator τ_I is also a special renaming operator; in this case all atomic actions given by the set I ($I \subseteq A$) are renamed into the atomic action τ, while all others remain unchanged. This operator, as may be clear from its name, is used to abstract from actions which are considered to be unobservable. These actions are given by the parameter I. The *time abstraction* operator π_{tI} is used to abstract from timing by embedding processes in the time-free theory, i.e., all atomic actions are made delayable and all time steps are removed. The *priority* operator Θ_C is used to achieve minimum delays (or maximal progress) for atomic actions from the set C ($C \subseteq A$).

4. A model for the bus lines

In this section we will design a process that models the behaviour of an individual wire of the I^2C-bus. This is necessary in order to validate the design for the interface of the next section.

Since the behaviour of one line does not depend on the behaviour of the other and since both lines are copies of the same piece of hardware, viz. a wired-AND, it suffices to design a process that describes a wired-AND. Hence, a model for the I^2C-bus consists of the parallel composition of the behaviour of two processes describing a wired-AND. Since there is no physical interaction between the two wires, the processes that model the wires will not communicate, with each other, when composed in parallel.

First we consider the communication between the wired-AND and its environment (see Fig. 2), the processes connected to it. Suppose the number of devices is N. The actions clamping and releasing by device i are represented as communications between the wired-AND and the device's interface over a channel $ch(i)$. Clamping the wire by device i will be denoted by sending the value dn along channel $ch(i)$ and releasing the wire is denoted by sending the value up. We also introduce the channels $stat(i)$, one for each device connected to the wired-AND. This channel is used to signal the current voltage level of the wire (high or low) to the devices communicating with it. Each process connected to the wire is then able to poll the wire in order to obtain the current voltage level.

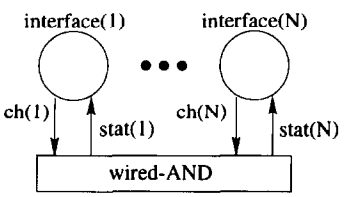

Fig. 2. The channels between the interfaces and the wired-AND.

Table 1
Equations describing a wired-AND: $2 \leqslant k < N$

WA	$= WA(0)$
$WA(0)$	$= \sum_{1 \leqslant i \leqslant N} (r_{ch(i)}(dn) \cdot WA(0 \to 1) + s_{stat(i)}(High) \cdot WA(0))$
$WA(1)$	$= \sum_{1 \leqslant i \leqslant N} (r_{ch(i)}(dn) \cdot WA(2) + r_{ch(i)}(up) \cdot WA(1 \to 0) + s_{stat(i)}(Low) \cdot WA(1))$
$WA(k)$	$= \sum_{1 \leqslant i \leqslant N} (r_{ch(i)}(dn) \cdot WA(k+1) + r_{ch(i)}(up) \cdot WA(k-1) + s_{stat(i)}(Low) \cdot WA(k))$
$WA(N)$	$= \sum_{1 \leqslant i \leqslant N} (r_{ch(i)}(up) \cdot WA(N-1) + s_{stat(i)}(Low) \cdot WA(N))$
$WA(0 \to 1)$	$= \sum_{0 \leqslant m \leqslant tF} \underline{\tau} \cdot \sigma^m_{rel}(WA(1))$
$WA(1 \to 0)$	$= \sum_{0 \leqslant m \leqslant tR} \underline{\tau} \cdot \sigma^m_{rel}(WA(0))$

Now we are ready to describe the process representing the wired-AND formally. This process is denoted by WA. We will use auxiliary processes $WA(k)$, for $0 \leqslant k \leqslant N$, that describe the situation where k devices clamp the wire. Initially, there is no device that clamps the wire, so $WA = WA(0)$. The specifications of these processes can be found in Table 1.

Each of the processes $WA(k)$, $0 \leqslant k < N$, is at any time willing to receive the *dn* value from a device. Then the parameter is increased by one in order to keep track of the number of clamping devices. In the situation where no devices are clamping the wire ($k = 0$), and a device clamps the wire, the voltage level of the wired-AND has to be changed to *Low*. This is achieved by the process $WA(0 \to 1)$. Process $WA(N)$ is not capable of receiving a *dn* value since all devices are clamping the wire.

Process $WA(k)$, $0 < k \leqslant N$, is at any time willing to receive an *up* value. In that case, the parameter is decreased by one. If the number of devices clamping the wire changes from 1 to 0, the voltage level of the wired-AND is changed to *High*. This is achieved by process $WA(1 \to 0)$. Process $WA(0)$ is not capable of receiving an *up* value since no devices are clamping the wire.

The processes $WA(0 \to 1)$ and $WA(1 \to 0)$ incorporate the maximum delay periods fall time tF and rise time tR respectively. In order to make the choice of delay time uncontrollable by the environment of the wire, we use the silent action $\underline{\tau}$. The $\underline{\tau}$ can be considered to hide a delay time determining action which is internal to the bus and hence invisible for the environment.

All processes $WA(k)$ are capable of signalling the current value of the wire. If there are no devices clamping the wire, the current voltage level is *High* and in all other cases, except when the wire is not stable, the voltage level is *Low*. The wire is not stable during the transition from high-to-low ($WA(1 \to 0)$) and from low-to-high ($WA(0 \to 1)$); in these cases, the wire cannot signal its current value.

The process descriptions for the *SDA* and *SCL* lines can now be obtained from process WA by renaming the channels according to the renaming functions fd and fc which are given in Table 2.

Using these renamings, the specifications of the lines are given by $SDA = \rho_{fd}(WA)$ and $SCL = \rho_{fc}(WA)$. The bus is then specified as the parallel composition of the two lines: $Bus = SDA \parallel SCL$.

Table 2
The renaming functions fd and fc ($1 \leqslant i \leqslant N$, $v \in \{up, dn\}$, $w \in \{High, Low\}$)

$fd(r_{ch(i)}(v))$	$= r_{d(i)}(v)$	$fc(r_{ch(i)}(v))$	$= r_{c(i)}(v)$
$fd(s_{stat(i)}(w))$	$= s_{t(i)}(w)$	$fc(s_{stat(i)}(w))$	$= s_{s(i)}(w)$

5. Designing an interface for a single master

5.1. Introduction

In this section, we design a process which acts as an interface between the I²C-bus and the devices connected to the bus. This interface translates the bus protocol in terms of the actions *start*, *stop*, *zero*, *one*, and *recbit* into clamping, releasing and polling the bus wires and vice versa. Furthermore, the interface will see to it that the timing constraints concerning the transitions are being met. In this way the master (slave) does not have to deal with the timing aspects of the protocol. Initially, we reduce the complexity of the design stage by simplifying the problem by imposing the following restrictions:

- we base the design on the behaviour of a pure master, i.e, a device which only acts as a master and not as a slave device;
- we limit the number of masters connected to the bus to one;
- we neglect the timing constraints.

In Section 5.4, an interface for a single master with timing constraints is designed based on the interface for the single master without timing constraints and in Section 6, we extend the interface to function in a multi-master environment. The first restriction will be maintained, we do not consider a device which can act as both a master and a slave. Before we design the interface, we describe the bus protocol in terms of the actions *start*, *stop*, *zero*, *one*, and *recbit* in Section 5.2.

5.2. The bus protocol

In this section, we will explain the communication protocol on the bus and how this protocol is used in the design of the interface between the bus and a device. The communication on the I²C-bus can be described as the behaviour of the following process, with initial state I:

$$I = start \cdot N$$

$$N = send0 \cdot N + send1 \cdot N + recbit \cdot N + stop \cdot I$$

The communication actions *start* and *stop* denote bus transitions which stand for the transmissions of *start* and *stop* signals and are always initiated by a master device. Communication actions *send*0, *send*1 and *recbit* indicate the transmission of a single bit with value zero or one, and the reception of a single bit, respectively.

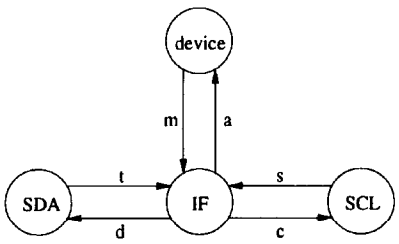

Fig. 3. The channels between the processes.

Process IF_i which models the interface for device i is connected to the processes *SDA*, *SCL* and the device as indicated in Fig. 3. Channel c_i is used for communications *up* and *dn* that cause transitions on the *SCL* line, channel s_i is used to signal the current state to the interface, channel d_i is used for communications *up* and *dn* that cause transitions on the *SDA* line, channel t_i is used to pass on the current state to the interface, channel m_i is used by the device to issue the commands to the process IF_i, and channel a_i is used to acknowledge to the device when an action is performed. Since the process specification for the interface is the same for each device, we will omit the subscript i when it is clear from the context which device is meant.

5.3. *Interface for a single master*

In this section, we construct an interface module for a pure master in a single master environment. In order to achieve this, we extend the bus protocol from the previous section. Since we are designing an interface that does not have to deal with any timing constraints at this point, we consider the rise and fall times to be zero: $tR = tF = 0$ or equivalently we consider $\pi_{tf}(Bus)$ instead of *Bus*.

The bus protocol is described in the previous section in terms of the primitives *start*, *stop*, *zero*, *one*, and *recbit*. These are all initiated by the master device. The matching signals that should occur on the bus are generated by *IF*'s subprocesses *T*, *P*, *S*0, *S*1 and *R*, respectively. These will also send matching acknowledgements to the device. Equations for process *IF* and its subprocesses can be found in Table 3.

The initial state is associated with process *I*. This state represents the situation that the master is not involved in a bus session. Since we have assumed that there is only one master connected to the bus, it follows that the bus is free, so both the *SDA* and *SCL* line are high. This means that when the interface returns to the initial state after a bus session, i.e., after the occurrence of a stop condition, both lines must be released.

As mentioned before, a session consists of communication actions. In between consecutive actions, the *SCL* line has to be clamped by the master. This situation is represented by process *N*. Although the I^2C-bus-specification [9] does not restrict the value of the *SDA* line, while the *SCL* line is low, we will demand that the master releases the *SDA* line. With this assumption a slave which is waiting to manipulate the data line is able to do so as soon as possible. We try to realize minimum delays. This

Table 3
Equations for the single master interface without timing constraints

$$
\begin{array}{lll}
IF = I & & T = DD \cdot CD \cdot (s_a(ready) \| DU) \cdot N \\
I = r_m(start) \cdot T & & P = DD \cdot CU \cdot DU \cdot s_a(ready) \cdot I \\
N = r_m(stop) \cdot P & & S0 = DD \cdot CU \cdot CD \cdot (s_a(ready) \| DU) \cdot N \\
\quad + r_m(send0) \cdot S0 & & S1 = CU \cdot CD \cdot s_a(success) \cdot N \\
\quad + r_m(send1) \cdot S1 & & R = CU \cdot (r_t(High) \cdot CD \cdot s_a(one) \\
\quad + r_m(recbit) \cdot R & & \quad + r_t(Low) \cdot CD \cdot s_a(zero)) \cdot N \\
CU = s_c(up) & & DU = s_d(up) \\
CD = s_c(dn) & & DD = s_d(dn)
\end{array}
$$

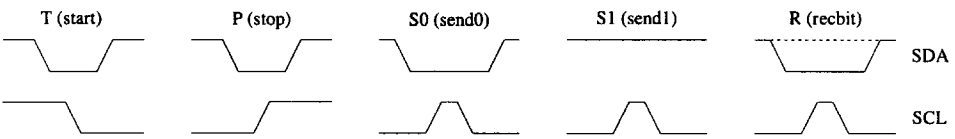

Fig. 4. The actions in terms of bus transitions.

decision is also convenient when a slave-transmitter is trying to manipulate the data line.

In Fig. 4 the bus transitions associated to the communication primitives are depicted. The equations from Table 3 correspond closely to these diagrams. Additionally, acknowledgements are sent via channel a to the device.

In order to send a *one*, the *SDA* line must be high during the clock pulse. Because we restricted the number of masters to one, and the *SDA* line had been released already, we do not have to check this voltage level. To read a single bit from the bus, the data line value must be checked during the high period of the clock, since only then the value of the *SDA* line can be considered stable. During the clock high period the voltage level of the *SDA* line is polled and its value is passed on to the device along channel a. The master must have released the *SDA* line to enable the slave-transmitter to put data on the bus.

5.4. Timing

From now on, we will no longer demand that the rise and fall times are equal to zero, as we did in Section 5.3. The new process that models the behaviour of the interface, within which timing constraints have to be met, will be called *TIF*. The connection with the bus and the device remains the same, however. In order to have process *TIF* deal with rise and fall times correctly, we will add some synchronization statements. Our construction of the bus with the transitional states $SCL(0 \to 1)$ and $SCL(1 \to 0)$ and the signalling of the current state provides a mechanism to solve this rather elegantly: after a releasing or clamping action the process that caused the transition, is simply delayed until the desired value is signalled by the bus lines. We

can now describe the up-going and down-going flanks of both the *SDA* and *SCL* line quite simply. Process *CD* (clock down) describes the clamping of the clock line and the subsequent waiting for the line to become low. Process *CU* (clock up) describes the releasing of the clock line and the delay that is caused by counting off the rise time. Processes *DD* (data down) and *DU* (data up) behave alike with regard to the *SDA* line. See Table 4 for their specifications. Note that the send actions of these processes are non-delayable because of the following assumption: *A device will not delay an action unnecessarily.*

It is assumed that all devices connected to the bus have a local internal clock mechanism. This implies that each device has a local clock high and clock low period. For device i these will be denoted by $THigh(i)$ and $TLow(i)$ respectively. If no confusion can arise, we omit the i. Furthermore, it is demanded that these local clock high and clock low periods are related to the "global" clock high and clock low period as follows: for all i, $THigh(i) \geqslant tHIGH$ and $TLow(i) \geqslant tLOW$.

We have to keep in mind that the timing of some actions does not depend on the execution of the previous action, as is usual with relative timing, but on the execution of some earlier action. We cannot express this naturally using the given algebraic operators.

This problem can be solved by introducing a timer process *Timer* (see Table 5), which is placed in parallel with process *TI*, i.e., $TIF = TI \parallel Timer$, that counts time slices that passed since a clock low period has started. As soon as *TLow* time slices have passed, $s_c(up)$ is allowed to be executed. In a similar way the timer process is used to count off the clock high period as well. Of course this timer process has to be instructed when it has to start counting and it must be able to signal to its environment when the counting has been completed. We need channels v and w along which there is communication between the timer and the interface process *TIF* (see Fig. 5). Along channel v the value *finished* is sent by the timer if the counting is completed and along channel w process *TIF* can send the number of time slices that have to be counted off.

Table 4
Equations for upgoing and downgoing flanks

$CU = \underline{\underline{s_c(up)}} \cdot r_s(High)$	$DU = \underline{\underline{s_d(up)}} \cdot r_t(High)$
$CD = \underline{\underline{s_c(dn)}} \cdot r_s(Low)$	$DD = \underline{\underline{s_d(dn)}} \cdot r_t(Low)$

Table 5
Equations for process *Timer*: $1 \leqslant n$

$$Timer = Timer(0)$$
$$Timer(0) = s_v(finished) \cdot Timer(0) + \sum_{0 \leqslant n} r_w(n) \cdot Timer(n)$$
$$Timer(n) = \sigma_{rel}(Timer(n-1)) + \sum_{0 \leqslant k} r_w(k) \cdot Timer(k)$$

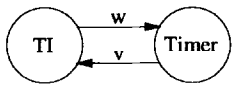

Fig. 5. Connecting process *Timer*.

Table 6
Equation for upgoing and downgoing clock flanks

$CU = r_v(\mathit{finished}) \cdot \underline{\underline{s_c(up)}} \cdot r_s(High) \cdot \underline{\underline{s_w(THigh)}}$
$CD = r_v(\mathit{finished}) \cdot \underline{\underline{s_c(dn)}} \cdot r_s(Low) \cdot \underline{\underline{s_w(TLow)}}$

In parameter n we store the number of time slices that still have to be counted off. Initially, the value of n equals zero. Process $Timer(0)$ represents the state in which any previous counting process is completed and therefore it is willing to signal this to process TI along channel v by means of the communication actions $s_v(\mathit{finished})$. This action must be delayable due to the fact that process TI will not always be ready to receive this value. Therefore, process *Timer* realizes only minimum delays. During execution of any process $Timer(n)$, $n > 0$, n is decreased by one after each σ-step. All processes $Timer(n)$, for $0 \leqslant n$, have the possibility to restart the counting at any time.

With the assumption that the beginning of the counting off of the clock high and low periods should be based on transitions of the *SCL* line, we can redefine the processes *CU* and *CD* (see Table 6). Process *CU* now first checks if process *Timer* has finished counting and hence if the clock line is allowed to become high, by $r_v(\mathit{finished})$. When this is the case, the clock line is released followed by the delay caused by counting off the rise time by process *SCL*. Finally, process *Timer* is started to count off the clock high period. Process *CD* behaves similarly to start the counting off of the clock low period. Note that a single-master situation is still being regarded and consequently, the clock high period cannot be ended prematurely by another device with a shorter clock high period.

Now we are equipped with the building blocks to add the timing constraints to our interface. Under the assumption that actions will not be delayed unnecessarily, set-up and hold times will not be increased unnecessarily. In Fig. 1 the bus transitions associated to the communication primitives are depicted together with the setup and hold times. The equations from Table 7 correspond closely to those diagrams.

The bus free time between a stop and a start condition is guaranteed by process *TP*. The rise and fall times and the clock high and low periods are maintained by the processes *CU*, *CD*, *DU*, and *DD*. When receiving a bit, the setup and hold time for data are enforced by the device transmitting the data bit (see Section 7).

Table 7
Equations for a single master interface with timing constraints

$$
\begin{aligned}
TIF &= TI \,\|\, Timer \\
TI &= r_m(start)\cdot TT \\
TN &= r_m(stop)\cdot TP + r_m(send0)\cdot TS0 + r_m(send1)\cdot TS1 + r_m(recbit)\cdot TR \\
TT &= DD\cdot \sigma_{rel}^{tHD;STA}(CD)\cdot \sigma_{rel}^{tHD;DAT}(\underline{\underline{s_a(ready)}} \,\|\, \underline{\underline{s_d(up)}})\cdot TN \\
TP &= DD\cdot \sigma_{rel}^{tSU;DAT}(CU)\cdot \sigma_{rel}^{tSU;STO}(DU)\cdot \sigma_{rel}^{tBUF}(\underline{\underline{s_a(ready)}})\cdot TI \\
TS0 &= DD\cdot \sigma_{rel}^{tSU;DAT}(CU)\cdot CD\cdot \sigma_{rel}^{tHD;DAT}(\underline{\underline{s_a(ready)}} \,\|\, \underline{\underline{s_d(up)}})\cdot TN \\
TS1 &= r_t(High)\cdot \sigma_{rel}^{tSU;DAT}(CU)\cdot CD\cdot \sigma_{rel}^{tHD;DAT}(\underline{\underline{s_a(success)}})\cdot TN \\
TR &= CU\cdot(\underline{\underline{r_t(High)}}\cdot CD\cdot \underline{\underline{s_a(one)}} + \underline{\underline{r_t(Low)}}\cdot CD\cdot \underline{\underline{s_a(zero)}})\cdot TN
\end{aligned}
$$

6. Designing an interface for a multi-master environment

6.1. Introduction

When we allow more than one master to control the bus, we will encounter four kinds of difficulties:

- Since a session may only be started when the bus is free, it is not allowed, as in the single master model, to start a session by simply commanding the interface to transmit a *start* signal. Another master could be using the bus. So, first a check has to be made whether the bus is free. One way to handle this is to detect and keep track of the *start* and *stop* signals on the bus. This is the issue of Section 6.3.
- When two or more masters try to put data onto the bus, an arbitration procedure is needed to avoid the chaos that might ensue from such an event. In Section 6.4 we will design a process specification for this arbitration procedure.
- Another aspect of the multi-master control is that of synchronizing the clock signals generated by the various master devices. In Section 6.4, we will explain how clock synchronization can be described.
- Since processes *SDA* and *SCL* are deaf during their transitional states, all clamping actions of the *SDA* line and the *SCL* line have to become delayable in the multi-master case to prevent the presence of deadlock. Note that all releasing actions can remain non-delayable since only the last interface to release a line will start the low-to-high transition. The altered versions of the processes describing the downgoing flanks of both the *SDA* and *SCL* line will be given in Section 6.2.

6.2. The interface in the multi-master environment

The process which models the interface between bus and master device in the multi-master case will be called *MIF*. The connecting channels between *MIF* and the bus and the device are as defined before. The names of the subprocesses of *MIF* will start with *M*.

As we have mentioned earlier all clamping actions of the interface have to become delayable due to the deaf state of the processes modelling the bus lines. This restriction simply means that the processes *CD* and *DD* from the previous section have to be altered in this respect that the actions $s_c(dn)$ and $s_d(dn)$ become delayable. In Table 8 the specifications for *MCD* and *MDD* are given.

6.3. Session recording

We introduce a separate process *MStatus*, composed in parallel with the processes *MI* and *Timer*, connected to processes *SCL* and *SDA* by the channels *s* and *t*, respectively and to the device by the new channel *ms*. Communication between process *MStatus* and process *MI* takes place along the new channel *b* (see also Fig. 6).

Process *MStatus* is introduced in order to describe a mechanism to monitor the bus lines. During monitoring the transmissions of *start* and *stop* signals can be recognized and from the order of their occurrence process *MStatus* is able to determine whether the bus is busy or free. Also, process *MStatus* is used to signal the outcome to either the master device or the interface component *MI*.

Initially, the bus is free. Both the *SDA* line and the *SCL* line are high. This state is described by process *MStatus*(*free*). *MStatus*(*free*) is capable of detecting a *start* signal that occurs on the bus (see Table 9). Recall that a *start* is a high-to-low transition on the *SDA*, while the *SCL* line is high. Such transitions are detected by "polling" the bus lines at well chosen moments in time. In our construction of the

Table 8
Equations for the processes *MCD* and *MDD*

$MCD = r_v(finished) \cdot s_c(dn) \cdot r_s(Low) \cdot s_w(TLow)$
$MDD = s_d(dn) \cdot r_t(Low)$

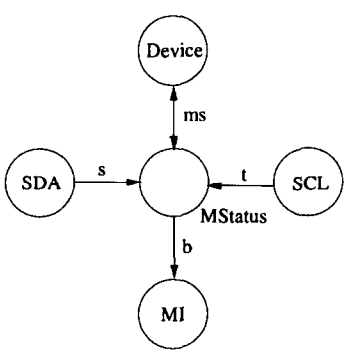

Fig. 6. Connecting process *MStatus*.

Table 9
Equations for the process *MStatus*

$$
\begin{aligned}
MStatus &= MStatus(free)\\
MStatus(free) &= r_t(Low)\cdot r_s(Low)\cdot MStatus(busy)\\
&\quad + r_{ms}(start)\cdot s_b(startsession)\cdot MStatus(free)\\
MStatus(busy) &= r_s(High)\cdot(r_t(Low)\cdot(r_t(High)\cdot\sigma_{rel}^{tBUF}(MStatus(free))\\
&\quad + r_s(Low)\cdot MStatus(busy))\\
&\quad + r_t(High)\cdot r_s(Low)\cdot MStatus(busy))\\
&\quad + r_{ms}(start)\cdot s_{ms}(busy)\cdot r_s(Low)\cdot MStatus(busy)
\end{aligned}
$$

bus model, where the *SDA* and *SCL* components are able to signal their status to process *MIF* in every time slice, if the status can be defined, polling can be accomplished by checking whether a communication is possible on the "status" channels s of the *SCL* line and t of the *SDA* line. Process *MStatus*(*free*) initiates the communication on channel t, connected to process *SDA* by the delayable atomic action $r_t(Low)$. Because of the delayability of this receive action, the process keeps checking whether the *SDA* line is low until it is. The other value, *High*, will be ignored when it is sent by process *SDA*. As soon as the low value on the data line is received, the process waits until the clock line has been pulled down in a similar manner, viz., by $r_s(Low)$. Only then, we can be certain that a *start* has occurred and the bus is now busy. We assume that each device sticks to the protocol, so no undefined transitions are to be expected such as releasing the *SDA* line before the clock line is pulled down.

Process *MStatus*(*busy*) (see Table 9) behaves similarly to detect a *stop* conditon. After a *stop* signal has been recognized the bus status is free. A *stop* is generated by producing a low-to-high transition on the *SDA* line while the *SCL* line is high. Since the data on the *SDA* line can only be considered valid during a clock high period, inspections of the *SDA* line are only useful after the clock line has been found high. Note that the clock line is always low when process *MStatus*(*busy*) is started. Since we are only interested in the case in which the clock line is high, the process is delayed until a low-to-high transition on the clock line takes place ($r_s(High)$). Then, in the same time slice, the value of the *SDA* line is checked. The non-delayability of this receive action will not cause deadlock since both the set-up stop (*tSU;STO*) and the set-up start (*tSU;STA*) timing constraints see to it, that the value of the data line is at least stable during the first time slice after the high clock period has started.

If the value of the *SDA* line is found to be high in the beginning of the clock high period, the only signal that can occur is the transmission of a *one*. Hence, a *stop* did not occur and the status remains busy. A new attempt to detect a *stop* is made, starting in the next clock low period. This is expressed by the alternative $r_t(High)\cdot r_s(Low)\cdot MStatus(busy)$.

If, on the other hand, the value of the *SDA* line is low, two events may occur: the actual *stop* signal or a high-to-low transition on the clock line, resulting in the

transmission of a zero. The latter event can be detected by $r_s(Low)$. Also in this case the status remains to be busy.

The first event, in which the actual transmission of the *stop* signal is completed, indicates that the low-to-high transition of the *SDA* line will occur before the high period of the clock is ended. Since the end of the clock high period will be noticed immediately by the above alternative, we do not have to add an explicit time bound to the action that detects the transition on the data line ($r_t(High)$). After the high value is received and subsequently the bus free time is counted off, the status of the bus will be free. Summarizing, the both events, that may happen when the data line was found low immediately after the clock high period started, can be detected by $\underline{\underline{r_t(Low)}} \cdot (r_t(High) \cdot \sigma_{rel}^{tBUF}(MStatus(free)) + r_s(Low) \cdot MStatus(busy))$.

After explaining the detection of *start* and *stop* conditions on the bus, the one matter left to discuss is which process MI, or the master, will communicate directly with *MStatus*. We choose to let the master signal to process *MStatus* that it wants to start a session. Process *MStatus* responds by sending the current state, either to the master, when the bus is busy, or to MI, when the bus is free and so a *start* condition can be generated. When the bus is busy, the master itself should decide if a new attempt will be made.

In order to accomplish the communication between processes *MStatus* and MI a channel b has been introduced. The only value that will be sent and received along this channel is *startsession*, which indicates to process MI that a session can be started. To handle the communication with the master we introduced a channel *ms* between *MStatus* and the master device. The process descriptions for process *MStatus* can be found in Table 9.

As a consequence, process MI needs some adjustment as well (see Table 10); $MI = r_b(startsession) \cdot MT$. After receiving the *startsession* signal, now being generated by process *MStatus*, subprocess MT is started to produce a *start* condition on the bus.

Table 10
Equations for the interface of a pure master in a multi-master environment

$$
\begin{aligned}
MIF &= MI \,\|\, Timer \,\|\, MStatus \\
MI &= r_b(startsession) \cdot MT \\
MN &= r_m(stop) \cdot MP + r_m(send0) \cdot MS0 + r_m(send1) \cdot MS1 + r_m(recbit) \cdot MR \\
MT &= MDD \cdot \sigma_{rel}^{tHD:STA}(ECD) \cdot \sigma_{rel}^{tHD:DAT}(\underline{\underline{s_a(ready)}} \,\|\, \underline{\underline{s_d(up)}}) \cdot MN \\
MP &= MDD \cdot \sigma_{rel}^{tSU:DAT}(CU) \cdot \sigma_{rel}^{tSU:STO}(DU) \cdot \sigma_{rel}^{tBUF}(\underline{\underline{s_a(ready)}}) \cdot MI \\
MS0 &= MDD \cdot \sigma_{rel}^{tSU:DAT}(CU) \cdot ECD \cdot \sigma_{rel}^{tHD:DAT}(\underline{\underline{s_a(ready)}} \,\|\, \underline{\underline{s_d(up)}}) \cdot MN \\
MS1 &= \sigma_{rel}^{tSU:DAT}(CU) \cdot (\underline{\underline{r_t(High)}} \cdot ECD \cdot \sigma_{rel}^{tHD:DAT}(\underline{\underline{s_a(success)}}) \cdot MN + \underline{\underline{r_t(Low)}} \cdot \underline{\underline{s_a(fail)}} \cdot MI) \\
MR &= CU \cdot (\underline{\underline{r_t(High)}} \cdot ECD \cdot \underline{\underline{s_a(one)}} + \underline{\underline{r_t(Low)}} \cdot ECD \cdot \underline{\underline{s_a(zero)}}) \cdot MN \\
ECD &= MCD + r_s(Low) \cdot \underline{\underline{s_c(dn)}} \cdot s_w(TLow)
\end{aligned}
$$

6.4. Clock synchronization and arbitration

All masters generate their own clock on the *SCL* line to transfer data on the I^2C-bus. Data is only valid during the high period of the clock. A defined clock is therefore needed for the bit-by-bit arbitration procedure to take place.

Clock synchronization is performed using the wired-AND connection of the interfaces to the *SCL* line. A high-to-low transition on the *SCL* line will cause all devices concerned to start counting off their low period. When all devices have counted off their low period, the *SCL* line will be released and go high. Next, all the devices will start counting their high periods. The first device to complete its high period will pull the *SCL* line down again. In this way, a synchronized clock is generated with its low period determined by the device with the longest clock low period, and its high period determined by the one with the shortest clock high period.

In the single master case, we have used processes *CD* and *CU* to model the counting off of the clock low period and clock high period, respectively. Since only the counting off of the clock high period can be interrupted and hence shortened by another device, we can leave process *CU* unchanged and restrict ourselves to extending our description of *MCD* to *ECD* (see Table 10). The process *ECD* (extended clock down) must not only be able to detect whether its own local clock high period is finished, it also has to detect a high-to-low transition on the *SCL* line caused by another master. In that case the protocol describes that the clock line must be clamped immediately and the clock low period must be counted off. All subprocesses during which a clock releasing action (i.e. *MCD* occurs) have to be adapted (see Table 10).

Another difficulty that arises in the multi-master case is the arbitration procedure. Two or more masters may generate a start condition within the hold time for a start condition. Arbitration takes place on the *SDA* line while the *SCL* line is high, in such a way that the master which transmits a high level on the *SDA* line while another master is transmitting a low level will lose arbitration. This procedure will not affect the data transfer initiated by the winning master. So, a master can only lose arbitration when it is transmitting a *one*, while another master is transmitting a *zero*. Whenever a master loses arbitration it must wait for the bus to become free.

The arbitration procedure can be modelled by polling the value of the *SDA* line during the high clock period when a *one* is being sent. If the value of the *SDA* line is high the session is continued, if the value is low arbitration is lost. In that case the master is notified that the transmission failed (see process *MS*1 in Table 10).

7. The slave interface

So far, the interface only incorporates the functionality of a master device. In this section we will design an interface for a slave device based on the design of the interface for a master device from the previous section. A slave incorporates functions to detect a *start* signal on the bus and must be able to send and receive bits on clock

Table 11
Equations for the slave interface

$$
\begin{aligned}
SIF &= SI \,\|\, SStatus \\
SI &= r_m(send0)\cdot SS0 + r_m(send1)\cdot SS1 + r_m(recbit)\cdot SR + r_m(mute)\cdot SI \\
SS0 &= r_s(Low)\cdot \underline{\underline{s_c(dn)}}\cdot SDD\cdot \sigma_{rel}^{tSU:DAT}(\underline{\underline{s_c(up)}})\cdot r_s(High)\cdot r_s(Low)\cdot \\
&\quad \sigma_{rel}^{tHD:DAT}(\underline{\underline{s_a(ready)}} \,\|\, \underline{\underline{s_d(up)}})\cdot SI \\
SS1 &= r_s(Low)\cdot \underline{\underline{s_c(dn)}}\cdot r_t(High)\cdot \sigma_{rel}^{tSU:DAT}(\underline{\underline{s_c(up)}})\cdot r_s(High)\cdot r_s(Low) \\
SDD &= s_d(dn)\cdot r_t(Low) \\
SR &= r_s(Low)\cdot r_s(High)\cdot (\underline{\underline{r_t(Low)}}\cdot \underline{\underline{s_a(zero)}} + \underline{\underline{r_t(High)}}\cdot \underline{\underline{s_a(one)}})\cdot SI \\
SStatus &= SStatus(free) \\
SStatus(free) &= r_t(Low)\cdot r_s(Low)\cdot \underline{\underline{s_{ms}(start)}}\cdot SStatus(busy) \\
SStatus(busy) &= r_s(High)\cdot (\underline{\underline{r_t(Low)}}\cdot (r_t(High)\cdot SStatus(free) \\
&\qquad + r_s(Low)\cdot SStatus(busy)) \\
&\quad + \underline{\underline{r_t(High)}}\cdot r_s(Low)\cdot SStatus(busy))
\end{aligned}
$$

pulses generated by a master. Furthermore, the slave interface must be able to mute when the slave device instructs it to do so. This suggests that the interface for a slave can be realized by a part of the interface for a master.

The slave interface, from now on denoted by *SIF* (see Table 11), will be connected to process *SDA* via channels *d* and *t* and to *SCL* via channels *c* and *s*. It communicates with the slave device along channels *m* and *a*. These connections are similar to those used to connect a master interface to its environment (see Fig. 3).

The slave interface is a parallel composition of two components *SI* and *SStatus*. Process *SStatus* behaves similar to process *MStatus* from the master variant. *SStatus* is connected to process *SCL* via a channel *s* and to process *SDA* via *t*. Furthermore, *SStatus* is connected to the slave device via channel *ms*. Unlike the situation concerning process *MStatus*, there is no connection necessary between *SStatus* and *SI*.

Detecting *start* and *stop* signals on the one hand and sending and receiving bits on the other can take place completely independent. We will first construct the part of *SIF* which takes care of sending and receiving bits. Afterwards, we will use process *MStatus* to specify a process which will have the same functionality as we want *SStatus* to have.

Subprocesses *SS*0, *SS*1, and *SR* differ only from *MS*0, *MS*1, and *MR* in this respect that no clock pulse has to be generated during execution and that initially the clock line has to be polled to sense the high and low periods of the clock pulse generated by the master. Exchanging the value *mute* enables the slave device to instruct the interface to enter the initial state. For example, when the slave, after decoding the address sent by a master, discovers that it is not addressed and therefore no longer has to be part of the current session, the *mute* value will be sent to the slave's interface.

Transmitting a *zero* now consists of waiting for the clock line to become low, clamping the clock line so the master cannot end the low period before the data line is

put to the low level and clamping the *SDA* line. After the value of the *SDA* line has become low, the set-up data time has to be counted off and afterwards the process releases the clock line, enabling the master to start counting off the clock high period. As soon as the low value is detected on the clock line, the data hold time is counted off, the data line is released, and slave device is notified that the transmission is completed.

Process *SS*1 is less complex, since the *SDA* line was already released. Furthermore, a slave cannot lose arbitration. Receipt of a bit can be expressed in a way similar to the master variant. Only the actions related to the clock line are absent. Note that, at the beginning of the execution of process *SI*, nothing can be said about the state of the wires, this in contrast with process *N* from the master's interface.

As mentioned earlier, process *MStatus* will be used to express the desired functionality of process *SStatus*. The only feature we want process *SStatus* to have is notifying the master that a *start* signal has been detected. When the high value on the *SDA* line is detected, we know from Section 6 that the only signal that can be sent during the current clock pulse is a *one*. In order to inform the slave that a *start* is detected, we simply add the send action $s_{ms}(start)$, immediately before the status turns busy. The slave will initially wait for the communication of the value *start* along channel *ms* and then it will send its instructions to the interface along channel *m*.

8. Usefulness of the model

In this section, we will show how a system consisting of the I^{2}C-bus with M master devices and S slave devices connected to it can be described in the model. Then we indicate for what purposes the model can be used.

Suppose that M master devices and S slave devices are connected to the I^{2}C-bus and that process algebra specifications $Master_i$, and $Slave_j$ are given for those. So, $M + S$ devices are connected to the bus. In Fig. 7 this system is depicted.

In Section 4, a model for the bus with N devices connected is given. Hence, the model for the bus is parameterized with the number of devices connected to it. We will

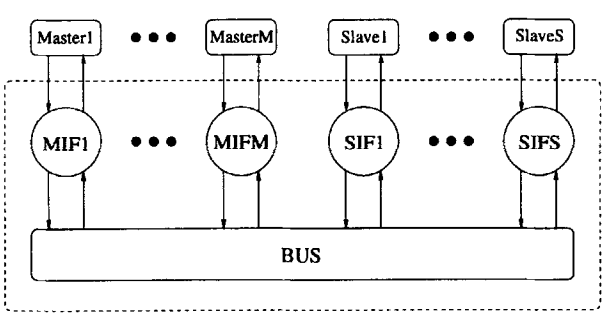

Fig. 7. The complete system.

simply write *Bus* instead of Bus_{M+S} or $Bus(M+S)$. Then the complete system can be described by the process

$$\Theta_C \circ \partial_H\left(\left(\mathop{\|}_{1 \leq i \leq M} (Master_i \| MIF_i)\right) \| \left(\mathop{\|}_{1 \leq j \leq S} (Slave_j \| SIF_j)\right) \| Bus\right)$$

where the set H consists of all send and receive actions, and the set C consists of all successful communication actions. The application of the encapsulation operator ∂_H removes all send and receive actions which, due to the interleaving, do not result in successful communications. The priority operator Θ_C ensures that successful communications are not delayed unnecessarily.

Next, we discuss some ways in which this model can be used. Firstly, since we explicitly modelled the timing constraints, it is possible to reason formally about the timing aspects involved in the I^2C-bus. Secondly, the model can be used when designing devices to be connected to the I^2C-bus. It is of uttermost importance that the interface between a device to and the bus is well-understood. It is our firm belief that a formal specification of such an interface improves this understanding considerably. Also, if a formal specification of the device is provided, there are possibilities to reason formally about the device being connected to the bus, i.e, their interaction can be considered. Thirdly, in situations where a number of devices have to communicate via the I^2C-bus it is useful to establish that there is absence of deadlock. Given specifications of those devices (in process algebra), one can easily check whether the complete system has absence of deadlock (see [7] for some examples). The main technique involved is that of "normalizing" a process term. Such a normal form only contains atomic actions, the constant deadlock and the operators $\cdot$ and $+$. For more information about normalization and related issues we refer to [5, 6]. From a normal form one also easily inspects the actual transitions on the bus lines, which might be useful to compare with the bus transitions that were expected. Fourthly, another important use of the model is that of testing conformance of new implementations of the I^2C-bus with the standard. Finally, the model of the I^2C-bus allows for abstraction, i.e., it can be looked at from different levels of abstraction. For example, if one is only interested to observe the order in which the communication primitives and the acknowledgements can be executed this can be achieved by applying the abstraction operators τ_I and π_{tf} to the full description (S), e.g., $\pi_{tf} \circ \tau_I(S)$, where I consists of the atomic actions from which we want to abstract. In this example we would abstract from successful communications along the channels b, c, d, ms, s, t, v, w.

With the abstraction operators, encapsulation and renaming we can also reason formally about the models that are obtained in the various design phases. For example, we can establish that the bus protocol from Section 5.2 and the system consisting of a time-free bus and a single master interface are in accordance as follows: $I = \rho_f \circ \tau_I \circ \partial_H(\pi_{tf}(Bus) \| IF)$, where $f(r_m(p)) = p$, I consists of all successful communications along the channels c, d, s, t, and the sending of the acknowledgements to the device, and H consists of the unsuccessful communication actions along c, d, s, t.

9. Conclusions

We introduced a special module which provided us with an interface between the bus and the devices. Such an interface simplifies the modelling of the problem considerably, since it is able to translate transitions on the bus into I^2C-bus primitives and vice versa. Regarding the first module, the behaviour of both *SDA* and *SCL* line relies on the behaviour of a wired-AND and hence, it sufficed to specify a process which behaves as a wired-AND in order to specify a model for the bus. As for the latter module, we made a distinction between an interface for a master device on the one hand and an interface for a slave device on the other. Starting from a high level specification of the data sessions that take place on the bus, we were able to derive a specification for the interface for a master device in a systematic and incremental manner. Specifications for the interface of a slave device could be derived from the interface for a master device quite easily, since the behaviour of a slave differs only slightly from the behaviour of a master device. We tried to keep the interface for both master and slave devices as simple as possible by shifting some tasks to the devices using the interface. For instance, the conversion of bits to bytes and vice versa and the decoding of the address byte have become tasks of the device.

Because of the large interest of the timing aspects imposed by the bus protocol, a discrete time process algebra with relative timing appeared to be the most suitable algebra to express the system of bus and devices connected to the bus. We can conclude that the discrete time process algebra with relative timing is expressive enough for the specification of a communication protocol which is restricted by timing constraints. We only needed one separate timer process to meet some of the timing constraints. However, due to the relative character of the algebra, most of the timing constraints could be met by simply defining the correct order in which the bus transitions should occur. Some steps in the design path were very straightforward or were the logical consequence of previous steps. Others were more complicated.

With the model for the bus lines and the interfaces with slave and master device as presented in this paper it is possible to

- reason formally about the timing aspects involved in the I^2C-bus;
- check freedom of deadlocks for a given description (in process algebra) of a device;
- test conformance of new implementations of the I^2C-bus with the standard.

In our design, the interface monitors the bus lines in addition to generating the transitions of the lines. Subject of further research could be the uncoupling of these two functions. In this case, the interface must be split into two modules, one for the monitor function and one for the generation of the transitions. We expect that the synchronization of such two modules will be very complex. Still, it would be an interesting starting-point. Another option for further research would be to design an interface, which is solely engaged in transmitting the several bus signals, without having to deal with the timing constraints at all. Timing would be the full responsibility of the device. We chose that the translation performed by the interface occurs on the level of the bus primitives. It would be interesting to investigate other possibilities

for the level in which the translation occurs. One could think of a shift towards the I^2C-bus.

Acknowledgements

The first author would like to thank Jos Baeten and Rudolf Mak (both of Eindhoven University of Technology) for their valuable remarks on preliminary (and extended) versions of this document. The second author would also like to thank Jos Baeten for setting-up the collaboration between the authors. We also would like to acknowledge Twan Basten (Eindhoven University of Technology), Loe Feijs (Eindhoven University of Technology and Philips Research Laboratory Eindhoven), and the anonymous references for their valuable comments.

Appendix A. Axioms

In this appendix the axioms of the discrete time process algebra with relative timing are given, for x, y, and z variables, $a \in A \cup \{\tau, \delta\}$, $H, I, C \subseteq A, f: A \to A$, and $n \geqslant 0$. The set A of atomic actions and the partial, commutative and associative communication function $\gamma: A \times A \to A$ are considered parameters of the process algebra.

$$x + y = y + x \qquad \sigma_{rel}(x) + \sigma_{rel}(y) = \sigma_{rel}(x + y)$$
$$(x + y) + z = x + (y + z) \qquad \sigma_{rel}(x) \cdot y = \sigma_{rel}(x \cdot y)$$
$$x + x = x \qquad x + \underline{\delta} = x$$
$$(x + y) \cdot z = x \cdot z + y \cdot z \qquad \underline{\delta} \cdot x = \underline{\delta}$$
$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \qquad \sigma^0_{rel}(x) = x$$
$$v_{rel}(\underline{a}) = \underline{a} \qquad \sigma^{n+1}_{rel}(x) = \sigma_{rel}(\sigma^n_{rel}(x))$$
$$v_{rel}(x + y) = v_{rel}(x) + v_{rel}(y) \qquad \lfloor x \rfloor^\omega = v_{rel}(x) + \sigma_{rel}(\lfloor x \rfloor^\omega)$$
$$v_{rel}(x \cdot y) = v_{rel}(x) \cdot y \qquad a = \lfloor \underline{a} \rfloor^\omega$$
$$v_{rel}(\sigma_{rel}(x)) = \underline{\delta} \qquad y = v_{rel}(x) + \sigma_{rel}(y) \Rightarrow y = \lfloor x \rfloor^\omega$$
$$x \cdot (\underline{\tau} \cdot (v_{rel}(y) + z) + v_{rel}(y)) = x \cdot (v_{rel}(y) + z)$$
$$x \cdot (\underline{\tau} \cdot (y + v_{rel}(z)) + y) = x \cdot (y + v_{rel}(z))$$
$$\underline{a} \cdot x = \underline{a} \cdot y \Rightarrow \underline{a} \cdot (\sigma_{rel}(x) + v_{rel}(z)) = \underline{a} \cdot (\sigma_{rel}(y) + v_{rel}(z))$$
$$x \cdot (\tau \cdot \lfloor y + z \rfloor^\omega + \lfloor y \rfloor^\omega) = x \cdot \lfloor y + z \rfloor^\omega$$

$$\pi_{tf}(\underline{a}) = a$$
$$\pi_{tf}(\sigma_{rel}(x)) = \pi_{tf}(x)$$
$$\pi_{tf}(x \cdot y) = \pi_{tf}(x) \cdot \pi_{tf}(y)$$
$$\pi_{tf}(x + y) = \pi_{tf}(x) + \pi_{tf}(y)$$
$$\pi_{tf}(\lfloor x \rfloor^\omega) = \pi_{tf}(x)$$

$$x \| y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x | y \qquad \underline{a} \cdot x | \underline{b} = (\underline{a} | \underline{b}) \cdot x$$
$$\underline{a} \lfloor\!\lfloor x = \underline{a} \cdot x \qquad \underline{a} | \underline{b} \cdot x = (\underline{a} | \underline{b}) \cdot x$$

$$\underline{a}\cdot x \mathbin{\underline{\|}} y = \underline{a}\cdot(x \| y) \qquad \underline{a}\cdot x|\underline{b}\cdot y = (\underline{a}|\underline{b})\cdot(x\|y)$$
$$(x+y)\mathbin{\underline{\|}} z = x\mathbin{\underline{\|}} z + y\mathbin{\underline{\|}} z \qquad (x+y)|z = x|z + y|z$$
$$\sigma_{rel}(x)\mathbin{\underline{\|}} v_{rel}(y) = \underline{\delta} \qquad x|(y+z) = x|y + x|z$$
$$\sigma_{rel}(x)\mathbin{\underline{\|}}(v_{rel}(y)+\sigma_{rel}(z)) = \sigma_{rel}(x\mathbin{\underline{\|}} z) \qquad \sigma_{rel}(x)|v_{rel}(y) = \underline{\delta}$$
$$\underline{a}|\underline{b} = \underline{\underline{\gamma(a,b)}} \text{ if } \gamma \text{ defined} \qquad v_{rel}(x)|\sigma_{rel}(y) = \underline{\delta}$$
$$\underline{a}|\underline{b} = \underline{\delta} \text{ otherwise} \qquad \sigma_{rel}(x)|\sigma_{rel}(y) = \sigma_{rel}(x|y)$$

$$\rho_f(\underline{a}) = \underline{\underline{f(a)}} \text{ if } a \in A \qquad \rho_f(x\cdot y) = \rho_f(x)\cdot\rho_f(y)$$
$$\rho_f(\underline{\delta}) = \underline{\delta} \qquad \rho_f(x+y) = \rho_f(x) + \rho_f(y)$$
$$\rho_f(\underline{\tau}) = \underline{\tau} \qquad \rho_f(\sigma_{rel}(x)) = \sigma_{rel}(\rho_f(x))$$
$$\partial_H(\underline{a}) = \underline{a} \quad \text{if } a \notin H \qquad \tau_I(\underline{a}) = \underline{a} \quad \text{if } a \notin I$$
$$\partial_H(\underline{a}) = \underline{\delta} \quad \text{if } a \in H \qquad \tau_I(\underline{a}) = \underline{\tau} \quad \text{if } a \in I$$
$$\partial_H(x+y) = \partial_H(x) + \partial_H(y) \qquad \tau_I(x\cdot y) = \tau_I(x)\cdot\tau_I(y)$$
$$\partial_H(x\cdot y) = \partial_H(x)\cdot\partial_H(y) \qquad \tau_I(x+y) = \tau_I(x) + \tau_I(y)$$
$$\partial_H(\sigma_{rel}(x)) = \sigma_{rel}(\partial_H(x)) \qquad \tau_I(\sigma_{rel}(x)) = \sigma_{rel}(\tau_I(x))$$

$$\Theta_C(\underline{a}) = \underline{a}$$
$$\Theta_C(\sigma_{rel}(x)) = \sigma_{rel}(\Theta_C(x))$$
$$\Theta_C(x\cdot y) = \Theta_C(x)\cdot\Theta_C(y)$$
$$\Theta_C(\underline{a}+x) = \underline{a} + \Theta_C(x) \quad \text{if } a \notin C$$
$$\Theta_C(\underline{a}\cdot x + y) = \underline{a}\cdot\Theta_C(x) + \Theta_C(y) \quad \text{if } a \notin C$$
$$\Theta_C(\underline{a} + v_{rel}(x) + \sigma_{rel}(y)) = \underline{a} + \Theta_C(v_{rel}(x)) \quad \text{if } a \in C$$
$$\Theta_C(\underline{a}\cdot x + v_{rel}(y) + \sigma_{rel}(z)) = \underline{a}\cdot\Theta_C(x) + \Theta_C(v_{rel}(y)) \quad \text{if } a \in C$$

References

[1] ACCESS.bus Industry Group, Sunnyvale, California, *ACCESS.bus Specifications – Version 2.1* (1993).

[2] J.C.M. Baeten and J.A. Bergstra, Discrete time process algebra (extended abstract), in: W.R. Cleaveland, ed., *CONCUR '92, 3rd Internat. Conf. on Concurrency Theorey,* Stony Brook, 1992, Lecture Notes in Computer Science, Vol. 630 (Springer, Berlin, 1992) 401–420.

[3] J.C.M. Baeten and J.A. Bergstra, Discrete time process algebra with abstraction, in: H. Reichel, ed., *FCT '95, Internat. Conf. on Fundamentals of Computation Theory,* Dresden, 1995, Lecture Notes in Computer Science, Vol. 965 (Springer, Berlin, 1995) 1–15.

[4] J.C.M. Baeten and J.A. Bergstra, Discrete time process algebra, *Formal Aspects Comput.* **8**(2) (1996) 188–208.

[5] J.C.M. Baeten and C. Verhoef, Concrete process algebra, in: S. Abramsky, D.M. Gabbay and T.S.E. Maibaum, eds., *Semantic Modelling, Handbook of Logic in Computer Science,* Vol. 4 (Oxford Univ. Press, Oxford, 1995) 149–268.

[6] J.C.M. Baeten and W.P. Weijland, *Process Algebra, Cambridge Tracts in Theoretical Computer Science,* Vol. 18 (Cambridge Univ. Press, Cambridge 1990).

[7] S.H.J. Bos, The I^2C-bus in discrete time process algebra, Master's Thesis, Eindhoven University of Technology (1995).

[8] A. Hanse, Connecting asynchronous devices to the I^2C-bus, Master's Thesis, Eindhoven Univerisity of Technology (1993).

[9] Philips, I^2C-bus compatible ICs, *Philips Data Handbook: Integrated Circuits,* Vol. IC 12 (Philips, Eindhoven, 1989).