Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

Asynchronous Readers and Writers

Antoon H. Boode [a,b,1], and Jan F. Broenink [a]
[a] *Robotics and Mechatronics,*
Faculty of Electrical Engineering, Mathematics and Computer Science,
University of Twente, the Netherlands
[b] *InHolland University of Applied Science, the Netherlands*

Abstract. Reading and writing is modelled in CSP using actions containing the symbols **?** and **!**. These reading and writing actions are synchronous and there is a one-to-one relationship between occurrences of pairs of these actions. It is cumbersome to ease the restriction of synchronous execution of the read and write actions. For this reason we introduce the half-asynchronous parallel operator that acts on actions containing the symbols **¿** and **¡** and study the impact on a Vertex Removing Synchronised Product.

Keywords. CSP, Half-Synchronous Alphabetised Parallel Operator, Asynchronous Write and Read Actions, Vertex Removing Synchronised Product

Introduction

Periodic hard real-time robotic applications can be modelled using formal methods like process algebras. These models describe the behaviour that the application has to exhibit. This is well-known, for example by using a graphical tool like TERRA [1].

To implement the models, a transformation can be made to graphs[1]. The graphs are then, as a data-structure, the controlling mechanism in the processes that execute on some hard real-time operating system. An architecture of such a system is given in Boode and Broenink [2]. This architecture is implemented by de Boer [3] on an embedded processor running the QNX® Neutrino® Real-Time Operating System (RTOS).

The performance of the architecture proposed in [2] has serious drawbacks due to the large amount of synchronisation messages by which the Synchronisation Server controls the synchronous execution of actions, as if these actions are executed atomically by the involved processes. For this reason we have defined a Vertex Removing Synchronised Product [4,5], denoted as $\boxtimes$, that, while multiplying graphs, reduces the longest path in the set of graphs representing the processes of the periodic hard real-time application. This longest path determines the worst case execution time within a period of the application. Although this improves the performance of the application, it does not help the designer in the modelling process.

One of the problems that a designer may encounter, is the situation where a process has to communicate a certain value with one or more processes. If this has to be executed synchronously, formal languages like Communicating Sequential Processes (CSP) [6] supply such a mechanism inherently. But if the actions of writing and reading are asynchronous,

[1]Corresponding Author: *Ton Boode, Robotics and Mechatronics, CTIT Institute, Faculty EEMCS, University of Twente, P.O. Box 217 7500 AE Enschede, The Netherlands*. Tel.: +31 631 006 734; E-mail: `a.h.boode@utwente.nl`.

[1]We only consider finite directed acyclic labelled multi-graphs.

languages like CSP have no operator that support this. Therefore an arguably complex design has to be made to enforce asynchronous writing and reading.

A mechanism by which the writer and the readers have an optional communication is describer by Gruner et al. [7], called the *optional parallel operator*, denoted as $\Uparrow$. This mechanism is still synchronous in the sense that during the communication the writer and only those readers that are able to recieve that data are engaging in the data transfer. All other processes that could receive the data will not, because they are not in the appropriate state yet. In this manner the characteristics of synchronous interaction are relaxed to a subset of the reading processes.

Another approach is given by Marwedel [8] who describes an extended rendezvous, by which the acknowledgement from the receiver to the sender is delayed, such that the receiver can perform checks or calculations on the received data.

In this paper we propose a *half-synchronous action* which allows a process to write a value x over a channel c, without the requirement that the reading processes must be in a state where they can read the value x over a channel c [2]. The writing action is denoted as ¡ (c ¡ x : T) and the reading action is denoted as ¿ (c ¿ x : T). This means that we adjust the alphabetised parallel operator, $_X||_Y$, in a similar fashion as [7] and introduce the *half-synchronous alphabetised parallel operator* $_X\Downarrow_Y$.

For simplicity we require that the reading processes execute their c ¿ x : T synchronously[3]. Of course this requirement can be relaxed to a definition of the half-synchronous action, where the reading processes are divided into sets which are set-wise asynchronous, but intra-set-wise synchronous, giving full flexibility to the asynchronous write and reads.
The advantages of the $_X\Downarrow_Y$ operator is three-fold;

- it eases the complexity of the design eliminating arguably complex process specifications:
  - it is not necessary to use a buffer process in the model to achieve asynchronous writing and reading,
  - the writes (¡) and reads (¿) are asynchronous, which makes it possible to have an order of writes and reads that, if synchronous (**!**, **?**), would lead to a deadlock,
- by reducing the number of actions involved in this asynchronous writing and reading of the processes, improves the performance of the periodic hard real-time application,
- in a distributed computing system, for example a processor-coprocessor combination, the waiting time of the processor or coprocessor can be reduced.

Our interest is of a graph theoretical nature and we will show an adaptation of the Vertex-Removing Synchronised Product (VRSP) which supports the half-synchronous actions and the $_X\Downarrow_Y$ operator. The adjusted version of the VRSP is called the Dot Vertex-Removing Synchronised Product (DVRSP) and denoted as $\dot{\boxtimes}$.

In Section 1 we give the semantics for the $_X\Downarrow_Y$ operator. In Section 2 we adjust the VRSP so that this adjusted version of VRSP can produce graphs that enforce the semantic rules of the $_X\Downarrow_Y$ operator. In Section 3 we give an example of a distributed application with a processor-coprocessor combination. This example shows that, while the coprocessor is executing, the processor can execute actions that would otherwise be delayed. A significant performance gain is achieved by reducing the waiting time of processes running on the processor, while waiting for the coprocessor to finish. In Section 4 we discuss the advantages and

[2] In CSP this modelling is restricted to two processes interacting synchronously via an action containing the **!** and the **?**.

[3] Like all synchronous actions, this is handled by the Synchronisation Software as described in [2].

disadvantages of the ${}_X\Downarrow_Y$ operator and give the conclusions of our research. We finish with a view on our future work with respect to the ${}_X\Downarrow_Y$ operator and the DVRSP in Section 5.

1. Semantics of the Half-Synchronous Operator

In, for example, CSP [6] one has the possibility to let a process write a value via a variable that will be read by another process using channels. Schneider [9] describes the communication over a channel as 'If c is a channel name of type T, and v is a particular value of type T, then the CSP expression $c\,!v \rightarrow P$ describes a process which is initially willing to output v along channel c, and subsequently behave as P' and 'If processes $P(x)$ are defined for each $x \in T$ then the CSP input expression $c\,?x : T \rightarrow P(x)$ describes a process which is initially ready to accept any value x of type T along channel c'. But this is still synchronous.

According to Hoare [6] $c\,!v \rightarrow P_1$ can be written as $c.v \rightarrow P_1$ and $c\,?v \rightarrow P_2$ can be written as $c.v \rightarrow P_2$ where $c.v$ is just an action over which the processes P_1 and P_2 will synchronise. Hoare [6, page 134] observes 'the convention that channels are used for communication in only one direction and between only two processes'.

For our purpose this communication restricted to two processes in a synchronous manner is too restrictive. Often there is the need for one writer and n readers, for example, in the situation where a process wants to multicast a message to several other processes. It is well known that a designer using, for example, CSP or the Calculus of Communicating Systems (CCS) has sufficient operators to describe any problem at hand [10]. But such a description may become quite complicated, as an example if a designer wants to model the observer design pattern [11]. To ease the design of concurrent systems an operator supporting such patterns would be convenient from a pragmatic point of view.

Remark 1. *The concept of reading and writing to a buffer is not a rendezvous. An undefined time may elapse between the writing to and the reading from the buffer. Although in a rendezvous there is communication, possibly passing of data between the participating processes, in process algebra a rendezvous is just a synchronising action. If data is passed from one process to another during a rendezvous this is atomic; there is no time elapse between the writing and the reading of the processes.*

Writing to and reading from a buffer lies in between synchronous and asynchronous communication in the sense that the writer does not have to wait for the reader to do the write action, but the readers will read synchronously.

Communication via a buffer can be modelled using a synchronising action, which separates the writing of x and the reading of x in time. By this abstraction, the buffer, which is used on the implementation level, is not visible in the model.

As a simple CSP example in Listing 1, the processes A, B synchronise over a *sync* action which separate the *write.x* and *read.x* in time. The alphabet of A is X and the alphabet of B is Y.

$$\begin{array}{lcl} A & = & write.x \rightarrow sync \rightarrow \text{SKIP} \\ B & = & sync \rightarrow read.x \rightarrow \text{SKIP} \\ AB & = & A_X||_Y B \end{array}$$

Listing 1: Reading from and writing to a buffer.

The graphs $G_1, G_2, G_1 \boxtimes G_2$ representing the processes A, B and AB are given in Figure 1.

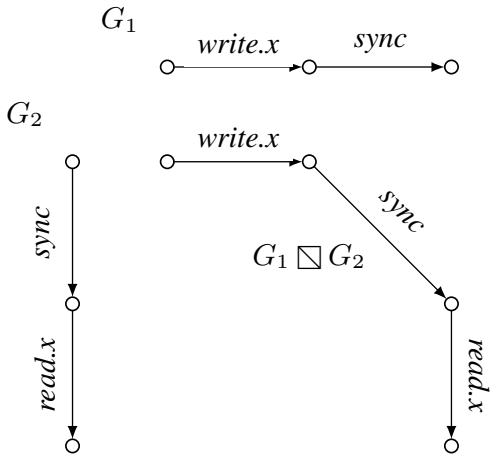

Figure 1. G_1, G_2 and $G_1 \boxtimes G_2$

Note that Roscoe [12] gives a more eloquent description of a buffer, which we give in Listing 2.

$$
\begin{aligned}
Buff^{\,N}_{\langle\rangle} &= left?x : T \rightarrow Buff^{\,N}_{\langle x\rangle} \\
Buff^{\,N}_{s^\frown\langle y\rangle} &= \#s < N - 1 \;\&\; (STOP \sqcap left?x : T \rightarrow Buff^{\,N}_{\langle x\rangle^\frown s^\frown\langle y\rangle}) \\
&\quad \Box\, right!y \rightarrow Buff^{\,N}_{s}
\end{aligned}
$$

Listing 2: Reading from and writing to a buffer [12].

The optional parallel operator $\Uparrow$, described by Gruner et al. [7], requires that "*any one or more* of these processes may synchronise with their environment." It is up to the process whether it will engage in this synchronisation.

Using this operator the designer cannot model a system where the writer process does not have to wait for a reading process which will synchronise with the writer process. At least one reading process must synchronously communicate with the writing process. Because we want to separate the writing action and the reading actions in time, we will not use this free choice of synchronisation. Instead we introduce an operator that disconnects the synchronisation of the writer process and the reader processes. We call this operator the *half-synchronous parallel alphabetised operator* denoted by ${}_X\Downarrow_Y$.

As symbols for the half-synchronous actions we use for reading ¿ and for writing ¡. We denote an action that contains the ¡ as *¡-action* and an action that contains the ¿ as *¿-action*. The semantics of ${}_X\Downarrow_Y$ is that

- the *¡-action* is asynchronous and unique with respect to the *¡-actions* of other processes and
- the *¿-action* is enabled if the related *¡-action* (see Definition 1) has been executed.

Whenever there is more than one process containing related *¿-actions*, these actions are synchronous.

The rationale is that we want to be able to model one writer and n readers where the waiting-time of the readers is, although timely in a real-time fashion, undefined. In this manner the writer can continue its task without being delayed by the readers. The readers will read atomically as if in one action. This is where VRSP shows its strength; the length[4] of the

[4] A directed path in a graph G is a sequence of distinct vertices $v_1 v_2 \ldots v_k$ of $V(G)$ such that $v_j v_{j+1} \in A(G)$ for $j = 1, \ldots, k-1$. The length of a path $v_1 v_2 \ldots v_k$ is defined as $\sum_{i=1}^{k-1} t(v_i v_{i+1})$. See Remark 2 for the definition of $t(v_i v_{i+1})$.

graph is reduced if the processes have the reading of a value on all of their longest paths. The behaviour is closely related to the observer pattern [11][5].

Nakata and Uustala [13] describe four co-inductive operational semantics, where co-inductivity is used for defining and proving properties of systems of concurrent interacting objects using the

- *small-step relational* semantics,
- *big-step relational* semantics,
- *small-step functional* semantics and
- *big-step functional* semantics.

For our half-synchronous operator the big-step relational semantics is important. We have to separate the writing and the reading in time. Therefore in any execution of the system a trace τ that contains a read, must also contain a write before the read, therefore $c ¡ x : T, c ¿ x : T \in \tau \Rightarrow c ¡ x : T < c ¿ x : T$[6].

Following [13], the proposition $(s, \sigma) \rightarrow (s', \sigma')$ states that in state σ the statement *s* one-step reduces to *s'* with the next state being σ'. These are exactly the same as one would use for an inductive semantics. The normalization relation is the terminal many-step reduction relation, defined co-inductively to allow for the possibility of infinitely many steps. The proposition $(s, \sigma) \rightsquigarrow \tau$ expresses that running s from σ results in the trace τ.

Here we deviate from [13]. Let $\overset{a}{\rightsquigarrow}$ denote a trace which contains a as an action. Let $\alpha(\rightsquigarrow)$ denote the alphabet containing the actions in $\rightsquigarrow$. Furthermore the CSP semantics of an action apply. Then Figure 2 gives the relational semantics of the ${}_X\Downarrow_Y$ operator.

$$\frac{P \overset{c¡x:T}{\rightsquigarrow} P',\ Q_1 \overset{c¿x:T}{\rightarrow} Q'_1, \cdots, Q_n \overset{c¿x:T}{\rightarrow} Q'_n}{P \Downarrow Q_1 \Downarrow \cdots \Downarrow Q_n \overset{c¡x:T}{\rightsquigarrow} P' \Downarrow Q_1 \Downarrow \cdots \Downarrow Q_n \overset{c¿x:T}{\rightarrow} P' \Downarrow Q'_1 \Downarrow \cdots \Downarrow Q'_n}, c ¿ x : T \notin (X, Z)$$

$$\frac{Q_i \overset{c¿x:T}{\rightarrow} Q'_i,\ Q_j \overset{y}{\rightarrow} Q'_j}{Q_i \Downarrow Q_j \overset{y}{\rightarrow} Q_i \Downarrow Q'_j}, y \neq c ¿ x : T, c ¿ x : T \in (Y_i \cdot Y_j), y \notin (X, Y_{k=1,\cdots n, j \neq k}, Z)$$

$$\frac{P \rightsquigarrow P',\ Q_i \overset{c¿x:T}{\rightarrow} Q'_i}{P \rightsquigarrow P'}, (\alpha(\rightsquigarrow) \cdot (Y_1, \cdots, Y_n, Z)) = \varnothing$$

$$\frac{Q_i \overset{c¡x:T}{\rightarrow} Q'_i,\ Q_j \overset{c¡x:T}{\rightarrow} Q'_j}{SKIP}, i \neq j$$

Figure 2. Relational semantics of the half-synchronous operator.

In Figure 2 the alphabets of $P, Q_1, \cdots, Q_n, R$ are denoted as $X, Y_1 \cdots, Y_n, Z$. Furthermore we define $X \bigcap Y_i = (X \cdot Y_i)$ and $X \bigcup Y_i = (X, Y_i)$.

For ease of reading we omit in Figure 2 for the parallel operator the alphabets, therefore $Q_i \,{}_{Y_i}\Downarrow_{Y_j}\, Q_j$ is denoted as $Q_i \Downarrow Q_j$.

[5]The observer pattern describes the behaviour of objects, where one object informs other objects of the occurrence of some event, for example a state change. The half-synchronous operator is a part of the description of the behaviour of processes. Arguably one might say that within the design cycle the half-synchronous operator acts on a more abstract level than the observer pattern.

[6]The order of two arcs v_1v_2, w_1w_2 is denoted by $v_1v_2 < w_1w_2$ if there exist a path from v_2 to w_1.

From a graph theoretical point of view this gives a restriction on the parallel actions, because a *¿-action* has to wait for the related *¡-action*.

Definition 1. *Two actions are related if and only if*

- *one action contains the ¡ precisely once and does not contain the ¿, and the other action contains the ¿ precisely once and does not contain the ¡,*
- *the prefix of the labels of both actions with respect to the ¡ and ¿ is identical and*
- *the postfix of the labels of both actions with respect to the ¡ and ¿ is identical.*

2. Impact on the Vertex Removing Synchronised Product

Of course the $\Downarrow$ operator leads to an adjustment of the definition of the VRSP ($\boxbslash$) and its intermediate stage ($\boxtimes$) into the Dot Vertex-Removing Synchronised Product (DVRSP) ($\dot{\boxbslash}$) and its dot intermediate stage ($\dot{\boxtimes}$).

As an example in Figure 4 we show the graph representing the case where n values are written by process P_1 and all or none are read by process P_2[7]. The processes P_1, P_2 are represented by graphs G_1, G_2 in Figure 3 and Figure 4. Because the DVRSP is defined in two stages, we give the dot intermediate stage of G_1, G_2, $G_1 \dot{\boxtimes} G_2$, in Figure 3 and the DVRSP of G_1, G_2, $G_1 \dot{\boxbslash} G_2$, in Figure 4. Note that $G_1 \dot{\boxtimes} G_2$ consists of three components and $G_1 \dot{\boxbslash} G_2$ consists of one component. Two components are removed in the second stage of DVRSP, because the level of the sources of these components are zero, whereas the level of these vertices in the Cartesian product of $G_1, G_2, G_1 \square G_2$ are greater than zero.

Remark 2. *The definition of a label has to be augmented. Boode et al. [4] have given as a definition for a label 'For each arc $a \in A$, $\lambda(a) \in L$ consists of a pair $(l(a), t(a))$, where $l(a)$ is a string representing an action and $t(a)$ is a positive real number representing the worst-case execution time of the action represented by $l(a)$'. $l(a)$ is augmented by the restriction that whenever ¿ and ¡ are in $l(a)$, the arc with label $\lambda(a)$ is either representing a reading or writing action.*

Remark 3. *Let the processes P_1, P_2, represented by graphs G_1, G_2 respectively, half-synchronise over some writing action $c ¡ x_i : T$ of P_1 and some reading action $c ¿ x_i : T$ of P_2 on some channel c. Then an arc representing a reading action $c ¿ x_i : T$ on some channel c, only makes sense in the product $G_1 \dot{\boxtimes} G_2$ if every path from the source of $G_1 \dot{\boxtimes} G_2$ to the arc representing the reading action $c ¿ x_i : T$ contains an arc representing a related writing action $c ¡ x_i : T$. Therefore, for an arc $v_2 w_2 \in A(G_2)$ with $l(v_2 w_2) = c ¿ x : T$, in every path from the source of $G_1 \dot{\boxtimes} G_2$ to $v_2 x_2, x_2 \in V(G_1), v_1 x_2 \in A(G_1 \dot{\boxtimes} G_2)$, there must be an arc labelled $c ¡ x_i$.Whenever this is not the case, the related reading process may encounter a deadlock. The opposite, if a writing action $c ¡ x_i : T$ is not followed by a related reading action $c ¿ x : T$ is not prohibited. Although useless, we do not prohibit the writing of values without reading.*

For two graphs G_i, G_j, an arc $v_i w_i \in V_i$ is related to an arc $v_j w_j \in V_j$ if in the processes represented by G_i, G_j, the actions they represent are related.

[7]The *waitForNextPeriod* action in Figure 4 is defined as a method in the class *RealtimeThread* in the Real-Time Specification for Java [14,15].

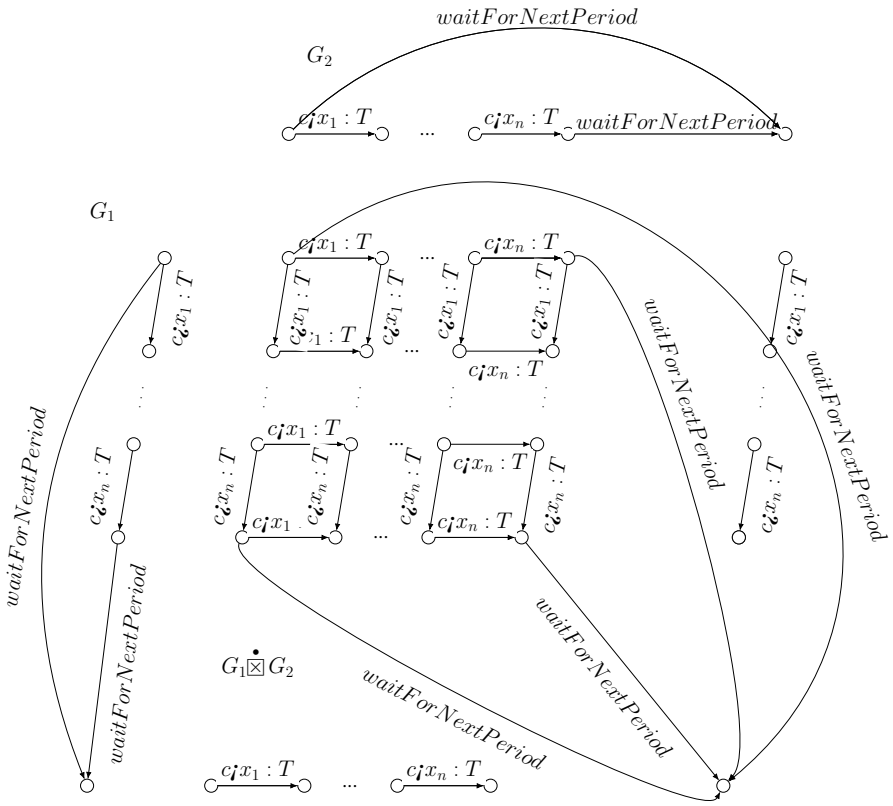

Figure 3. The intermediate stage of DVRSP for $G_1, G_2, G_1 \dot{\boxtimes} G_2$.

The DVRSP of G_i and G_j, $G_i \dot{\boxtimes} G_j$ is constructed in two stages.

Firstly, the dot intermediate stage, denoted as $G_i \dot{\boxtimes} G_j$ of G_i and G_j, is defined as the graph on vertex set $V_{i,j} = V_i \times V_j$ with three types of arcs:

- Arcs of type 0 are between pairs $(v_i, v_j) \in V_{i,j}$ and $(w_i, w_j) \in V_{i,j}$ with $v_i w_i \in A_i$, $v_j = w_j$, $\lambda(v_i w_i) \notin L_j$, and either ¡ $\in l(v_i w_i)$ or ¿ $\in l(v_i w_i)$ (with $v_j w_j \in A_j$, $v_i = w_i$, $\lambda(v_j w_j) \notin L_i$, and either ¡ $\in l(v_j w_j)$ or ¿ $\in l(v_j w_j)$). These arcs of $G_i \dot{\boxtimes} G_j$ are called half-synchronous arcs, and the set of these arcs is denoted as $A^h_{i,j}$. Thus, $A^h_{i,j} = \{(v_i, v_j)(w_i, w_j) | v_i, w_i \in V_i, v_j, w_j \in V_j, v_i w_i \in A_i, v_j = w_j, \lambda(v_i w_i) \notin L_j$ and either ¡ $\in l(v_i w_i)$ or ¿ $\in l(v_i w_i)$, or $v_j w_j \in A_j, v_i = w_i, \lambda(v_j w_j) \notin L_i$ and either ¡ $\in l(v_j w_j)$ or ¿ $\in l(v_j w_j)\}$
- Arcs of type 1 are between pairs $(v_i, v_j) \in V_{i,j}$ and $(w_i, w_j) \in V_{i,j}$ with $v_i w_i \in A_i$, $v_j = w_j$, $\lambda(v_i w_i) \notin L_j$, ¡ $\notin l(v_i w_i)$ and ¿ $\notin l(v_i w_i)$ (with $v_i = w_i$ and $v_j w_j \in A_j$, $\lambda(v_j w_j) \notin L_i$, ¡ $\notin l(v_j w_j)$ and ¿ $\notin l(v_j w_j)$). These arcs of $G_i \dot{\boxtimes} G_j$ are called asynchronous arcs, and the set of these arcs is denoted as $A^a_{i,j}$. Thus, $A^a_{i,j} = \{(v_i, v_j)(w_i, w_j) | v_i, w_i \in V_i, v_j, w_j \in V_j$ with $v_i w_i \in A_i, v_j = w_j$ and $\lambda(v_i w_i) \notin L_j$, or $v_j w_j \in A_j, v_i = w_i$ and $\lambda(v_j w_j) \notin L_i\}$
- Arcs of type 2 are between pairs $(v_i, v_j) \in V_{i,j}$ and $(w_i, w_j) \in V_{i,j}$ with $v_i w_i \in A_i$, $v_j w_j \in A_j$, $\lambda(v_i w_i) = \lambda(v_j w_j)$ and ¡ $\notin l(v_i w_i)$. These arcs of $G_i \dot{\boxtimes} G_j$ are called synchronous arcs, and the set of these arcs is denoted as $A^s_{i,j}$. Thus, $A^s_{i,j} = \{(v_i, v_j)(w_i, w_j) | v_i, w_i \in V_i, v_j, w_j \in V_j$ with $v_i w_i \in A_i, v_j w_j \in A_j, \lambda(v_i w_i) = \lambda(v_j w_j)$ and ¡ $\notin l(v_i w_i)\}$ and $A_{i,j} = A^h_{i,j} \bigcup A^a_{i,j} \bigcup A^s_{i,j}$.

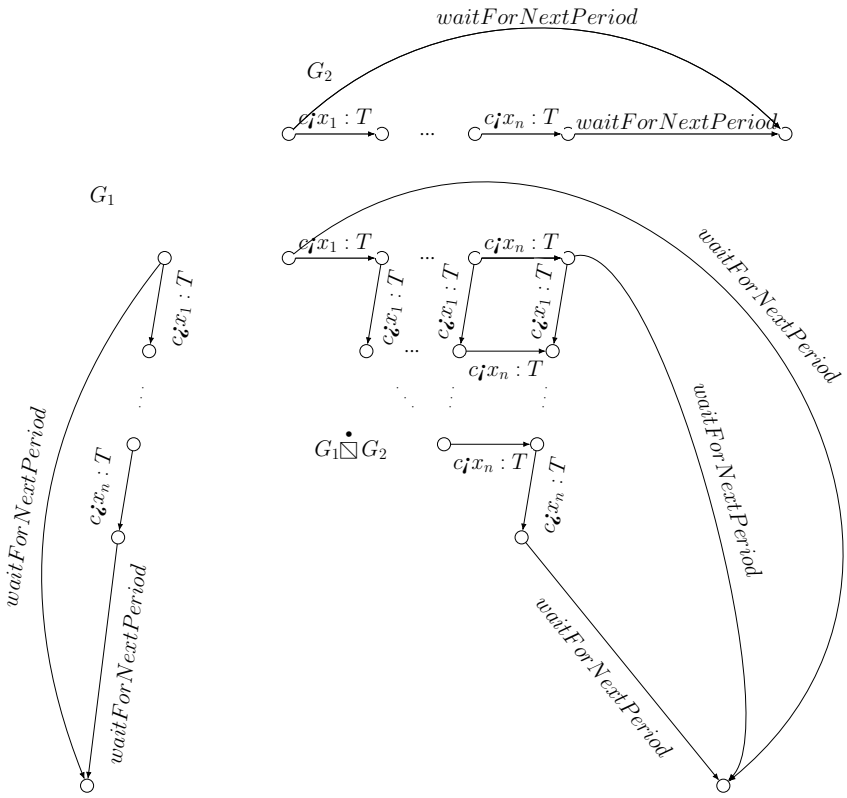

Figure 4. Half-synchronous writing and reading of $G_1 \dot{\boxslash} G_2$ and its factors G_1, G_2.

Secondly,

1. all arcs $v_x w_x \in A_{i,j}$ for which there exists a related arc $v_y w_y \in A_{i,j}$, with operator ¿ $\in \lambda(v_x w_x)$ for which there does not exist a related arc $v_y w_y$ with operator ¡ $\in \lambda(v_y w_y)$ with $v_y w_y < v_x w_x$ are removed,
2. all vertices at $level\,0$ in the intermediate stage that are at $level > 0$ in $G_i \square G_j$ are removed, together with all the arcs that have one of these vertices as a tail. This is then repeated in the newly obtained graph, and so on, until there are no more vertices at $level\,0$ in the current graph that are at $level > 0$ in $G_i \square G_j$.

The resulting graph is called the *Dot Vertex-Removing Synchronised Product (DVRSP)* of G_i and G_j, denoted as $G_i \dot{\boxslash} G_j$. For $k \geqslant 3$, the DVRSP $G_1 \dot{\boxslash} G_2 \dot{\boxslash} \ldots \dot{\boxslash} G_k$ is defined recursively as $((G_1 \dot{\boxslash} G_2) \dot{\boxslash} \ldots) \dot{\boxslash} G_k$.

Remark 4. *DVRSP does not allow two processes to write a value on the same channel.*

Without consistency[8] of the graphs, deadlocks with respect to the $\Downarrow$ operator are possible in the processes represented by these graphs. Only a read from x - read from y combination is prone to deadlocks, because a read from x (or a read from y) is a synchronous action. So if two processes, both reading from x and reading y in series, have their reads from x and reads from y interchanged, both processes will be deadlocked.[9]

[8] A definition of consistency of graphs is given in [5].

[9] The writing and reading shows a close resemblance with databases, where there are transactions writing and reading data concurrently. As an example, Bernstein et al. [16] show that the order in which data is written and read matters with respect to the consistency (in the sense of interference) of the data. They distinguish three types of execution of transactions; Recoverable executions, Avoiding Cascading Aborts executions and Strict executions. Of course the updates of the data in database systems have to be committed (the updates are

Remark 5. *This is not the case for the optional parallel operator. The effect will be that one of the two processes will not participate in the reading from either x or y.*

A deadlock will not occur in the write-read or write-write combination. For example, if in one process first writes to x and then reads from y, where the other process first writes to y and then reads from x, no deadlock occurs.

When the graphs are consistent, an interchanged order of reads between two processes can not occur. Therefore, consistent graphs will have no deadlock.

Remark 6. *The order in which a process reads is not relevant with respect to a process that only writes. For example, if the first process writes x and then y and the second process reads y and then x this will not give a deadlock. The result will be that the second process cannot start reading x before the y is written.*

From a performance point of view, the graph representing the example given in Listing 1 has a length of $\ell(G_1 \boxtimes G_2) = 3$ [10], whereas for the process representing $G_1 \boxtimes G_2$ the same behaviour is achieved by the process $A'B'$ given in Listing 3. The length of the graph representing the process $A'B'$ is 2. Although this reduces the number of context switches, the synchronisation software has to deal with the order of execution of the $c\,¡\,x : T$ action and the related $c\,¿\,x : T$ action. Therefore the performance gain depends on the time the synchronisation software needs to control the order of the actions. The alphabet of A' is X' and the alphabet of B' is Y'.

$$
\begin{array}{lcl}
A' & = & c\,¡\,x : T \;\rightarrow\; \text{SKIP} \\
B' & = & c\,¿\,x : T \;\rightarrow\; \text{SKIP} \\
A'B' & = & A' \Downarrow_{X'} B'
\end{array}
$$

Listing 3: Reading from and writing to a buffer.

3. Application

To show that the new operators are useful, we consider a system that runs at 1 KHz, so with a period of 1 msec. A part of the system consists of an application process and a controller process. The controller process communicates, for example, via memory mapped I/O with a coprocessor performing a Fast-Fourier Transform (FFT) on the received data.

Assume that the application process has to calculate eight values via the coprocessor. Let the controller process have priority over the application process. Furthermore the actions of the application process and the actions of the controller process take 10 μsec to execute. This includes context switches, state changes in the processes and the like. The coprocessor takes 70 μsec to calculate the FFT on each data item. Although the related[11] **!**-actions and **?**-actions communicate as a rendezvous, so in a sense atomically, their interaction takes 20 μsec. This leads to a simple CSP specification given in Listing 4 using **!**-actions and the **?**-actions, where the alphabet of $Application$ is A and the alphabet of $Controller$ is C.

considered valid) or aborted (they are considered as if the updates never happened), which is an aspect of data we do not take into account.

[10] In this example the execution time related to an arc a, $t(a)$, is one by default.

[11] Related in a similar fashion as defined for the ¡-actions and ¿-actions in Definition 1.

$$
\begin{aligned}
Application \;=\; & c_1\,!\,x_1 : T \rightarrow c_2\,?\,y_1 : T \rightarrow \\
& \cdots \\
& c_1\,!\,x_8 : T \rightarrow c_2\,?\,y_8 : T \rightarrow \\
& display\_f(y_1, \cdots, y_8) \rightarrow \text{SKIP}
\end{aligned}
$$

$$
\begin{aligned}
Controller \;=\; & c_1\,?\,x_1 : T \rightarrow writeCoProc.x_1 \rightarrow readCoProc.y_1 \rightarrow c_2\,!\,y_1 : T \rightarrow \\
& \cdots \\
& c_1\,?\,x_8 : T \rightarrow writeCoProc.x_8 \rightarrow readCoProc.y_8 \rightarrow c_2\,!\,y_8 : T \rightarrow SKIP
\end{aligned}
$$

$$
System_1 \;=\; Application_{A} ||_{C} Controller
$$

Listing 4: Reading from and writing to a buffer.

In Figure 5 we show the time line for $System_1$ with the application process (AP), the control process (CP) and the coprocessor (CoP). Obviously there is a deadline miss, because $System_1$ needs more than one msec to execute.

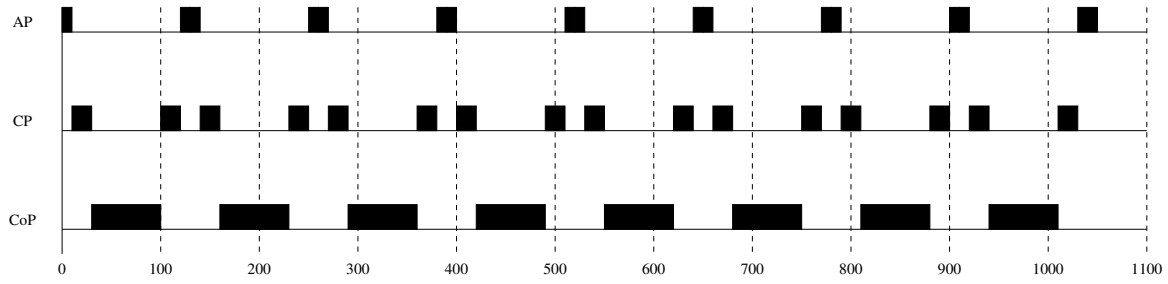

Figure 5. Time line of the application process, the control process and the coprocessor, using **!** operator and **?** operator.

Using the new $\Downarrow$ operator and the **¡**-actions and the **¿**-actions, this leads to an equally simple CSP specification given in Listing 5.

$$
\begin{aligned}
Application \;=\; & c_1\,\text{¡}\,x_1 : T \rightarrow \cdots \rightarrow c_1\,\text{¡}\,x_8 : T \rightarrow \\
& c_2\,\text{¿}\,y_1 : T \rightarrow \cdots \rightarrow c_2\,\text{¿}\,y_8 : T \rightarrow \\
& display\_f(y_1, \cdots, y_8) \rightarrow \text{SKIP}
\end{aligned}
$$

$$
\begin{aligned}
Controller \;=\; & c_1\,\text{¿}\,x_1 : T \rightarrow writeCoProc.x_1 \rightarrow readCoProc.y_1 \rightarrow c_2\,\text{¡}\,y_1 : T \rightarrow \\
& \cdots \\
& c_1\,\text{¿}\,x_8 : T \rightarrow writeCoProc.x_8 \rightarrow readCoProc.y_8 \rightarrow c_2\,\text{¡}\,y_8 : T \rightarrow SKIP
\end{aligned}
$$

$$
System_2 \;=\; Application_{A} \Downarrow_{C} Controller
$$

Listing 5: Reading from and writing to a buffer.

In Figure 6 we show the time line for $System_2$ with the application process (AP), the control process (CP) and the coprocessor (CoP). Now during the time that the coprocessor is executing, the application process is writing the $x_2, \cdots, x_8$ values via channel c. Furthermore the reading of the $y_1, \cdots, y_7$ is as well executed during the execution of the coprocessor. $System_2$ is an improvement of $System_1$ by 140 μsec as the time line in Figure 6 shows.

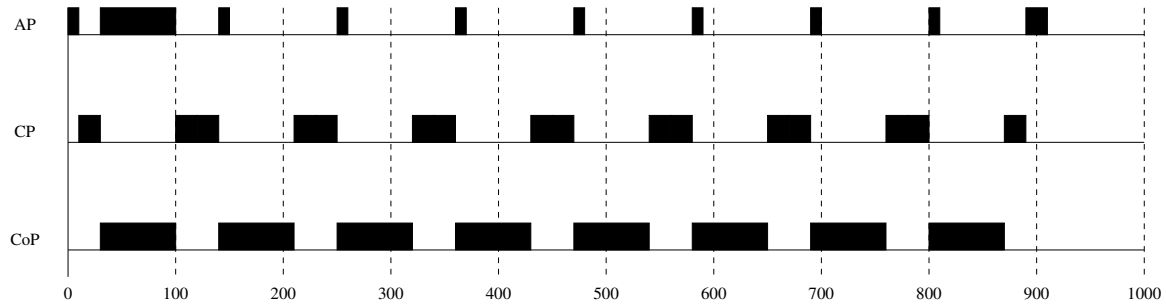

Figure 6. Time line of the application process, the control process and the coprocessor, using ¡ operator and ¿ operator.

4. Discussion and Conclusions

In this paper we have discussed a new ${}_X\Downarrow_Y$ operator and the new *¡-action* and *¿-action*, that delay the reading of a process from a buffer. The ${}_X\Downarrow_Y$ operator together with the *¡-action* and *¿-action* are an abstraction of a buffer, therefore the designer does not have to model the buffer as well. In this manner the writing process does not have to wait for the reading process to synchronise. There are three advantages of the ${}_X\Downarrow_Y$ operator in combination with the DVRSP

- it eases the design by taking away the burden of separating the writing and reading in time,
- the length of the longest path is reduced, if the operators are part of all the longest paths of the participating graphs,
- in a distributed computing system, for example a processor-coprocessor combination, the waiting time of the processor or coprocessor can be reduced.

The first advantage will make the design less error prone and therefore the design phase needs less time. Furthermore the overall design cycle will gain because the improved description on design level will lead to less effort for the implementation and less effort for testing. The second and third advantage will lead to an application which needs less execution time, thereby reducing the possibility of a deadline miss.

5. FutureWork

The ${}_X\Downarrow_Y$ operator is synchronous as far as the reading processes are concerned. This can be extended to an asynchronous-set of *¿-actions*, where the reading processes are divided into sets which are set-wise asynchronous, but intra-set-wise synchronous, giving full flexibility to the asynchronous write and reads.

Of course the algebraic properties of the monoid $(\dot{\boxtimes}, K_1)$ have to be formulated and proved. After that the implementation by [3] has to be extended with the $\dot{\boxtimes}$. A case study has to show whether significant improvement on both design and implementation level will be obtained with the half-synchronous operator.

Acknowledgement

The authors would like to express their gratitude to the anonymous reviewers for the very useful suggestions and comments. The research of the first author has been funded by the InHolland University of Applied Science, Alkmaar, The Netherlands.

References

[1] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink. Design and use of csp meta-model for embedded control software development. In P. H. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen, and A. T. Sampson, editors, *Proceedings of the 34th WoTUG Technical Meeting, Dundee, United Kingdom*, volume 69 of *Concurrent System Engineering Series*, pages 185–199, England, August 2012. Open Channel Publishing Ltd.

[2] A. H. Boode and J. F. Broenink. Performance of periodic real-time processes: a vertex-removing synchronised graph product. In *Communicating Process Architectures 2014, Oxford, UK*, 36th WoTUG conference on concurrent and parallel programming, pages 119–138, Bicester, August 2014. Open Channel Publishing Ltd.

[3] M. P. de Boer. Implementation of periodic hard real-time processes by means of a graph theoretical approach. Bsc report 011ram2016, University of Twente, June 2016.

[4] A. H. Boode, H. J. Broersma, and J. F. Broenink. Improving the performance of periodic real-time processes: a graph theoretical approach. In *Communicating Process Architectures 2013, Edinburgh, UK*, 35th WoTUG conference on concurrent and parallel programming, pages 57–79, Bicester, August 2013. Open Channel Publishing Ltd.

[5] A.H. Boode, H.J. Broersma, and J.F. Broenink. On a directed tree problem motivated by a newly introduced graph product. *GTA Research Group, Univ. Newcastle, Indonesian Combinatorics Society and ITB*, 3, no 2 (2015): Electronic Journal of Graph Theory and Applications, 2015.

[6] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, aug 1978.

[7] Stefan Gruner, Derrick G. Kourie, Markus Roggenbach, Tinus Strauss, and Bruce W. Watson. A new csp operator for optional parallelism. In *CSSE (2)*, pages 788–791. IEEE Computer Society, 2008. 978-0-7695-3336-0.

[8] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[9] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*, chapter 1, page 1. John Wiley Sons, Inc., New York, NY, USA, 1st edition, 1999.

[10] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[12] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[13] Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for while. In *Theorem Proving in Higher Order Logics*, pages 375–390. Springer, 2009.

[14] Gregory Bollella. *The Real-time Specification for Java*. Addison-Wesley Java Series. Addison-Wesley, 2000.

[15] Andrew Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, 2004.

[16] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.