Communicating Process Architectures 2017
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2017

Asynchronous Readers and Asynchronous 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 actions and writing actions are synchronous, and there is a one-to-one relationship between occurrences of pairs of these actions. In the CPA conference 2016, we introduced the half-synchronous alphabetised parallel operator $X \Downarrow Y$, which disconnects the writing to and reading from a channel in time. We introduce in this paper an extension of $X \Downarrow Y$, where the definition of $X \Downarrow Y$ is relaxed; the reading processes are divided into sets which are set-wise asynchronous, but intra-set-wise synchronous, giving full flexibility to the asynchronous writes and reads. Furthermore, we allow multiple writers to the same channel and we study the impact on a Vertex Removing Synchronised Product. The advantages we accomplish are that the extension of $X \Downarrow Y$ gives more flexibility by indexing the reading actions and allowing multiple write actions to the same channel. Furthermore, the extension of $X \Downarrow Y$ reduces the end-to-end processing time of the processor or coprocessor in a distributed computing system. We show the effects of these advantages in a case study describing a Controlled Emergency Stop for a processor-coprocessor combination.

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

Introduction

Periodic Hard Real-Time Control Systems (PHRCSs) modelled using process algebras comprise many short processes, which leads to fine-grained concurrency. To let the PHRCS perform its task as required by the specification, the processes synchronise over actions, asserting a certain order of the actions of the processes.

Due to the fine-grained concurrency and the related synchronisation of the involved processes, a significant part of the execution time (up to 20%) is consumed by context switches. The performance of such PHRCSs can be improved by reducing the number of context switches of the threads representing these processes.

The logic controlling the behaviour of these processes can be implemented by Finite State Machines (FSMs). These FSMs are in essence finite, directed, acyclic, labelled

[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`.

multi-graphs. To reduce the number of context switches, we introduced in [1] and [2] a Vertex-Removing Synchronised Product (VRSP) that combines processes by multiplication of the graphs representing the behaviour of these processes. The algebraic characteristics of VRSP are described in [3].

In process algebra, information is communicated in a synchronous manner. In Communicating Sequential Processes (CSP) the **!** and **?** symbols can be used to transfer data from one process to another. For example, **c!x:T** in one process and **c?x:T** in another process proceed synchronously as if both were written as **c.x** [4].

As we have shown in [5], disconnecting the writing and reading in time eases the task of a designer if such disconnections are required from the perspective of performance of the application. For this reason, we have introduced in [5] for CSP a half-synchronous alphabetised parallel operator ${}_{\alpha}\Downarrow_{\beta}$ [1] with alphabets α, β, together with half-synchronous actions **c¡x:T** and **c¿x:T**, that lie in between synchronous and asynchronous writing to enable asynchronicity between reading and writing. We have given the syntax and the semantics of the half-synchronous alphabetised parallel operator, together with a case study which shows the advantage of the half-synchronous alphabetised parallel operator with respect to memory occupation and performance. Furthermore, we have studied the impact of the half-synchronous alphabetised parallel operator on the VRSP which has led to the Dot Vertex-Removing Synchronised Product (DVRSP) ([5]).

Although reading actions and writing actions are asynchronous for the half-synchronous alphabetised parallel operator, the readers are still synchronising their reading actions. Therefore, we extend in this paper the half-synchronous alphabetised parallel operator such that the readers are allowed to read asynchronously. To achieve this asynchronous reading by readers, we add an index to the **¿** symbols such that read actions with the same index read synchronously and read actions with a different index read asynchronously. For example, **c¡x:T**$\in P_1$, **c¿x:T** $\in P_2$ and **c¿x:T** $\in P_3$ becomes **c¡x:T** $\in P_1$, **c¿**$_1$**x:T** $\in P_2$ and **c¿**$_2$**x:T** $\in P_3$. Furthermore, we allow more than one process to write to the same channel. Allowing only one process to write to a channel is a restriction from the early versions of CSP [7], but, for example, lifted to any-to-any channel in [8].

Whenever confusion can arise in the use of the term *processes* in the case of process algebra, and the term *processes* in the case of a process executing on some operating system, we use *process* to indicate a process-algebraic process, and we use *thread* when we mean a process or thread that executes on some operating system.

We start with a description of the terminology in Section 1. In Section 2 we introduce the extension of the half-synchronous operator with asynchronous readers, the *extended half-synchronous alphabetised parallel operator* (${}_X\Updownarrow_Y$), and describe the semantics of the ${}_{Y_i}\Updownarrow_{Y_j}$. Furthermore, we describe the impact of ${}_{Y_i}\Updownarrow_{Y_j}$ on the VRSP and the DVRSP, which leads to the definition of the Extended Dot Vertex-Removing Synchronised Product (EVRSP). We finish with a case study of the ${}_{Y_i}\Updownarrow_{Y_j}$, the Controlled Emergency Stop, showing the advantages of the newly introduced ${}_{Y_i}\Updownarrow_{Y_j}$ in Section 3.

1. Terminology

We use Bondy and Murty [9], Hammack et al. [10], Hell and Nešetřil [11], Milner [12], Schneider [4], Hoare [7] and Roscoe [13] for terminology and notation on graphs and

[1]The half-synchronous alphabetised parallel operator $X \Downarrow Y$ is based on the optional parallel operator of Gruner et al. [6]

processes not defined here. We consider finite, deterministic, directed, acyclic, labelled multi-graphs based on acyclic, deterministic processes written in the formal specification language CSP [7] only.

For convenience, we give definitions related to the half-synchronous operator given in [3] and [5], together with new definitions related to the new half-synchronous operator.

1.1. Process-Algebraic Aspects

We repeat from Boode and Broenink [5] the following notions which were necessary to describe the semantics of our new operator.

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.

The alphabets of the processes $P_1, \cdots, P_n, Q_1, \cdots, Q_m, R$ are denoted as $X_1, \cdots, X_n$, $Y_1, \cdots, Y_m, Z$ respectively. Furthermore, for alphabets $A_1, A_2, \cdots, A_n$ we define $A_1 \bigcap A_2 \cdots \bigcap A_n = (A_1 \cdot A_2 \cdot \ldots \cdot A_n)$ and $A_1 \bigcup A_2 \cdots \bigcup A_n = (A_1, A_2, \cdots, A_n)$.

Two actions are related if and only if

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

1.2. Graph-Theoretical Aspects

The graphs we consider consist of a vertex set V, an arc set A, a set of label pairs L, and two mappings. The first mapping $\mu : A \to V \times V$ is an incidence function that identifies the tail and head of each arc $a \in A$, so $\mu(a) = (v_i, v_j)$ means that the arc a is directed from $v_i \in V$ to $v_j \in V$, where $tail(a) = v_i$ and $head(a) = v_j$. The second mapping $\lambda : A \to L$ assigns a label pair $\lambda(a) = (l(a), t(a))$ to each arc $a \in A$, where $l(a)$ is a string representing the (name of an) action and $t(a)$ is the weight of the arc a. This weight $t(a)$ is a real positive number representing the worst case execution time of the action represented by $l(a)$.

A sequence of distinct vertices $v_0 v_1 \ldots v_k$ and arcs $a_1 a_2 \ldots a_k$ of G is a (directed) path in G if $\mu(a_i) = (v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. We denote such a path by $P = v_0 a_1 v_1 a_2 \ldots a_k v_k$.

An arc $a \in A(G)$ is called an in-flowing arc of $v \in V(G)$ if $head(a) = v$; the in-degree of v, denoted by $d^-(v)$ is the number of distinct in-flowing arcs.

Similarly, $a \in A(G)$ is an out-flowing arc of $v \in V(G)$ if $tail(a) = v$; the out-degree of v, denoted by $d^+(v)$ is the number of distinct out-flowing arcs.

The subset of V consisting of vertices v with $d^-_G(v) = 0$ is called the source of G, denoted as S'_G.

The subset of V consisting of vertices v with $d^+_G(v) = 0$ is called the sink of G, denoted as S''_G.

For each graph G, we define $S^0(G)$ to denote the set of vertices with in-degree 0 (the source of G) in G, $S^1(G)$ the set of vertices with in-degree 0 in the remaining graph obtained from G by deleting the vertices of $S^0(G)$ and all arcs with tails in $S^0(G)$, and so on, until the final set $S^t(G)$ contains the remaining vertices with in-degree 0 and out-degree 0 in the remaining graph. This ordering implies that arcs of G can only exist from a vertex in $S^{j_1}(G)$ to a vertex in $S^{j_2}(G)$ if $j_1 < j_2$. If a vertex $v \in V$ is in the set $S^j(G)$ in the above ordering, we also say that v is at *level* j in G.

We require that the following property holds for all the graphs we consider: any two distinct arcs $a \in A$ and $b \in A$ with $\mu(a) = \mu(b)$ have $l(a) \neq l(b)$.

For each pair $(v_i, v_j) \in V \times V$, we denote by $A(v_i, v_j)$ all $a_k \in A$ with $\mu(a_k) = (v_i, v_j)$.

A graph G is called deterministic[2] if all arcs in G have the following property. If $\lambda(a) = \lambda(b)$ for two arcs $a \in A$ and $b \in A$ with $head(a) \neq head(b)$, then $tail(a) \neq tail(b)$.

Let $a \in A(G)$ with $\mu(a) = (u, v)$. By contracting a we mean replacing u and v by a new vertex $\overline{uv}$, deleting all arcs $b \in A(G)$ with $\mu(b) = (u, v)$ or $\mu(b) = (v, u)$, and replacing each pair of arcs $c \in A(G)$ and $d \in A(G)$ with $\mu(c) = (u, x)$, $\mu(d) = (v, x)$ and $\lambda(c) = \lambda(d)$ by one arc e with $\mu(e) = (\overline{uv}, x)$ and $\lambda(e) = \lambda(c)$, and, similarly replacing each pair of arcs $c \in A(G)$ and $d \in A(G)$ with $\mu(c) = (x, u)$, $\mu(d) = (x, v)$ and $\lambda(c) = \lambda(d)$ by one arc e with $\mu(e) = (x, \overline{uv})$ and $\lambda(e) = \lambda(c)$.

Let T be the set of asynchronous arcs in $G_1 \boxbslash G_2$ that correspond to arcs in G_1. Then the contraction of $G_1 \boxbslash G_2$ with respect to G_1, denoted by $\rho_{G_1}(G_1 \boxbslash G_2)$, is defined as the graph obtained from $G_1 \boxbslash G_2$ by successively contracting each arc $a \in T$. Likewise, the contraction of $G_1 \boxbslash G_2$ with respect to G_2, denoted by $\rho_{G_2}(G_1 \boxbslash G_2)$, is the graph obtained from $G_1 \boxbslash G_2$ by successively contracting all asynchronous arcs of $G_1 \boxbslash G_2$ that correspond to arcs in G_2.

The Cartesian product $G_i \Box G_j$ of G_i and G_j is defined as the labelled multi-graph on vertex set $V_{i,j} = V_i \times V_j$, with two types of labelled arcs. For each arc $a \in A_i$ with $\mu(a) = (v_i, w_i)$, an arc of type i is introduced between tail $(v_i, v_j) \in V_{i,j}$ and head $(w_i, w_j) \in V_{i,j}$ whenever $v_j = w_j$; such an arc receives the label $\lambda(a)$. This implicitly defines parts of the mappings μ and λ for $G_i \Box G_j$. Similarly, for each arc $a \in A_j$ with $\mu(a) = (v_j, w_j)$, an arc of type j is introduced between tail $(v_i, v_j) \in V_{i,j}$ and head $(w_i, w_j) \in V_{i,j}$ whenever $v_i = w_i$; such an arc receives the label $\lambda(a)$. This completes the definition of the mappings μ and λ for $G_i \Box G_j$. So, arcs of type i and j correspond to arcs of G_i and G_j, respectively, and have the associated labels. For $k \geqslant 3$, the Cartesian product $G_1 \Box G_2 \Box \cdots \Box G_k$ is defined recursively as $((G_1 \Box G_2) \Box \cdots) \Box G_k$.

Since we are particularly interested in synchronising arcs, we modify the Cartesian product $G_i \Box G_j$ according to the existence of synchronising arcs, i.e., pairs of arcs with the same label pair, with one arc in G_i and one arc in G_j.

The first step in this modification consists of ignoring the synchronising arcs while forming arcs in the product, but additionally combining pairs of synchronising arcs of G_i and G_j into one arc, yielding the intermediate product which we denote by $G_i \boxtimes G_j$. To be more precise, $G_i \boxtimes G_j$ is obtained from $G_i \Box G_j$ by first ignoring all except for the so-called asynchronous arcs, i.e., by only maintaining all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_j = w_j$ and $\lambda(a) \notin L_j$, as well as all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_i = w_i$ and $\lambda(a) \notin L_i$. This set of arcs is denoted by $A^a_{i,j}$. Additionally, we add arcs that replace synchronising pairs $a_i \in A_i$ and $a_j \in A_j$ with $\lambda(a_i) = \lambda(a_j)$. If $\mu(a_i) = (v_i, w_i)$ and $\mu(a_j) = (v_j, w_j)$, such a pair is replaced by an arc $a_{i,j}$ with $\mu(a_{i,j}) = ((v_i, v_j), (w_i, w_j))$ and $\lambda(a_{i,j}) = \lambda(a_i) (= \lambda(a_j))$. The set of these so-called synchronous arcs of $G_i \boxtimes G_j$ is denoted by $A^s_{i,j}$.

The second step in this modification consists of removing (from $G_i \boxtimes G_j$) the vertices $(v_i, v_j) \in V_{i,j}$ and the arcs a with $tail(a) = (v_i, v_j)$, whenever (v_i, v_j) has $level > 0$ in $G_i \Box G_j$ and (v_i, v_j) has $level\, 0$ in $G_i \boxtimes G_j$. 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 $G_i \Box G_j$.

The resulting graph is called the Vertex Removing Synchronised Product (VRSP) of G_i and G_j, denoted as $G_i \boxbslash G_j$.

[2]This is equivalent to determinism in the set of processes that is represented by the graph G.

For $k \geqslant 3$, the VRSP $G_1 \boxbslash G_2 \boxbslash \cdots \boxbslash G_k$ is defined recursively as $((G_1 \boxbslash G_2) \boxbslash \cdots) \boxbslash G_k$.

Graphs G_i and G_j are consistent, denoted as $G_i \doteqdot G_j$, if and only if the following two requirements hold:

1. $\rho_{G_i}(G_i \boxbslash G_j) \cong G_j$ and $\rho_{G_j}(G_i \boxbslash G_j) \cong G_i$.
2. $S'_{G_i \boxbslash G_j} = S'_{G_i} \times S'_{G_j}$ and $S''_{G_i \boxbslash G_j} = S''_{G_i} \times S''_{G_j}$.

2. Extension of the Half-Synchronous Operator with Asynchronous Readers and Asynchronous Writers

In this section two extensions of the Half-Synchronous Operator are elaborated:

- Indexing of the **¿**-action, allowing set-wise asynchronous reading and intra-set-wise synchronous reading. The semantics of the ${}_{Y_i}\Updownarrow_{Y_j}$ is given in Section 2.1.
- Allowing more than one writer to write to the same channel. The DVRSP as defined in [5] and improved in Appendix A, inhibits two different writers to the same channel. The extension of the DVRSP, the Extended Dot Vertex-Removing Synchronised Product (EVRSP), is given in Section 2.2 on page 86.

Two or more writers to the same channel would be synchronous as the labels of the two actions are identical as, for example, in Listing 1. Because we want the writers to write asynchronously, the relational semantics of the half-synchronous alphabetised parallel operator has to be adapted. In the sequel, the *extended half-synchronous alphabetised parallel operator* is called the *extended half-synchronous operator* and is denoted as ${}_{Y_i}\Updownarrow_{Y_j}$. The EVRSP is denoted as $\overset{\diamond}{\boxbslash}$.

Remark 1. *Of course, we could index the asynchronous writes in a similar fashion as the asynchronous reads. We choose* not *to, because the writing at any point in time, when delivering* identical *objects to the readers, would lead to the passing of one object only, delaying all, but the last, threads that participate in the synchronisation. This is counter-intuitive to the idea that threads can write to a channel asynchronously, with the guarantee that their instance of an object is written to the channel at that point in time.*

2.1. Semantics of the Extended Half-Synchronous Alphabetised Parallel Operator

Let $P = \{P_1, \cdots, P_n\}$ be the set of processes containing asynchronous writes to the same channel, therefore $c¡x : T \in X_i, i = 1, \cdots, n$. Let $Q = \{Q_1, \cdots, Q_m\}$ be the set of processes containing an indexed asynchronous $¿_i - action, i \in I = \{1, \cdots, k\}$. Let $\bigcup\limits_{i \in I} I_i = \{1, \cdots, n\}$, $\bigcap\limits_{i \in I} I_i = \varnothing, i = 1, \cdots, k$, where the $j \in I_i$ is an index for the synchronous $¿_i - action$ for the subset of processes $\{Q_j | j \in I_i\}$.

Furthermore, in Figure 1 we give

- the semantics of the extended half-synchronous operator,
- if we need more than one process P we use P_i otherwise we use P and
- for ease of reading, we omit the alphabets for the extended half-synchronous operator, therefore $Q_i \; {}_{Y_i}\Updownarrow_{Y_j} \; Q_j$ is denoted as $Q_i \Updownarrow Q_j$.

Remark 2. *The **¿**$_i$-action is prone to deadlocks. If one process contains $c¿_i x : T$ followed by $c¿_j x : T, i \neq j$ and another process contains the same actions in reversed order the two processes may deadlock. Because we consider processes represented by consistent graphs only, such a process definition is inhibited.*

$$\frac{P_i \xrightarrow{c\,\text{¡}\,x:T} P_i',\ P_j \xrightarrow{c\,\text{¡}\,x:T} P_j'}{(P_i \Updownarrow P_j) \rightarrow ((P_i' \Updownarrow P_j) \oplus (P_i \Updownarrow P_j'))}$$

$$\frac{P \overset{c\,\text{¡}\,x:T}{\rightsquigarrow} P',\ Q_{i_1} \xrightarrow{c\,\text{¿}_i\,x:T} Q'_{i_1}, \cdots, Q_{i_j} \xrightarrow{c\,\text{¿}_i\,x:T} Q'_{i_j}}{P \Updownarrow Q_{i_1} \Updownarrow \cdots \Updownarrow Q_{i_j} \overset{c\,\text{¡}\,x:T}{\rightsquigarrow} P' \Updownarrow Q_{i_1} \Updownarrow \cdots \Updownarrow Q_{i_j} \xrightarrow{c\,\text{¿}_i\,x:T} P' \Updownarrow Q'_{i_1} \Updownarrow \cdots \Updownarrow Q'_{i_j}},\ I_i = \{i_1, \cdots, i_j\}$$

$$\frac{P \rightsquigarrow P',\ Q_j \xrightarrow{c\,\text{¿}_i\,x:T} Q'_j}{P \rightsquigarrow P'},\ c\,\text{¡}\,x : T \notin \alpha(\rightsquigarrow), (\alpha(\rightsquigarrow) \cdot (Y_1, \cdots, Y_n, Z)) = \emptyset$$

$$\frac{Q_{i_j} \xrightarrow{c\,\text{¿}_i\,x:T} Q'_{i_j},\ Q_{i_k} \xrightarrow{y} Q'_{i_k}}{Q_{i_j} \Updownarrow Q_{i_k} \xrightarrow{y} Q_{i_j} \Updownarrow Q'_{i_k}},\ y \neq c\,\text{¿}_i\,x : T, i_j, i_k \in I_x, x \in \{1, \cdots, k\}$$

Figure 1. Relational semantics of the extended half-synchronous operator.

2.2. EVRSP of the Extended Half-Synchronous Alphabetised Parallel Operator

As we are taking into account pairs of consistent graphs only, $c\text{¿}_n x : T$ in one process without a $c\text{¡}x : T$ in any process is inhibited, because the process may end in a deadlock and the deadlock violates the consistency requirements. But we still have to address issues like

- a series of identical writes $c\text{¡}x : T$ in one process and the related reads $c\text{¿}_n x : T$ in another process,
- a series of consecutive identical writes $c\text{¡}x : T$ to the same channel by different processes.

These issues are not inhibited by the semantics of ${}_{\alpha}\Updownarrow_{\beta}$. As an example, the processes P_1, P_2, P_3, P_{123} in Listing 1 are represented by the graph in Figure 3, which contains consistent graphs G_1, G_2, G_3 leading to $G_{123} = \overset{3}{\underset{i=1}{\overset{\diamond}{\boxbslash}}} G_i$.

$$\begin{array}{lcl}
P_1 & = & c\,\text{¡}\,x : T \ \rightarrow \ \text{SKIP} \\
P_2 & = & c\,\text{¡}\,x : T \ \rightarrow \ \text{SKIP} \\
P_3 & = & c\,\text{¿}_1\,x : T \ \rightarrow \ c\,\text{¿}_1\,x : T \ \rightarrow \ \text{SKIP} \\
P_{13} & = & P_1 \Updownarrow P_3 \\
P_{123} & = & P_1 \Updownarrow P_2 \Updownarrow P_3
\end{array}$$

Listing 1: Two processes writing to the same channel.

A schematic process sketch of the processes given in Listing 1, is given in Figure 2. This process sketch describes the communication flow of the involved processes and shows that there is no predefined order in which P_1 and P_2 communicate with P_3. It follows that the graphs representing these processes must be G_1, G_2, G'_{13}[3] and G_{123}, given in Figure 3, because $G''_{13} \overset{\diamond}{\boxbslash} G_2 \ncong G_{123}$.

[3]Because $G_1 \cong G_2$ the choice for G_{23} leads to the same result.

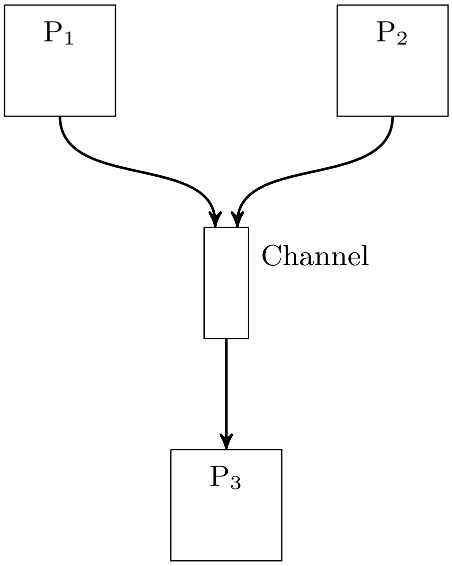


Figure 2. Process schema describing the communication flow of the processes P_1, P_2, P_3 (Listing 1).

Therefore it is clear that the graph G_{123} in Figure 3 represents the behaviour of the concurrent process P_{123}. But it is not clear what the graph should be that represents the concurrent process P_{13} given in Listing 1, because the write action could be related to the first read action or to the second read action. Therefore there are two choices for this example given by the graphs G'_{13} and G''_{13} in Figure 3. Following the process sketch in Figure 2, the graph G'_{13} should be chosen because the first two actions can be executed directly by the processes that represent these graphs, whereas the process representing the graph G''_{13} has to wait for the $c\,\text{¡}\,x : T$ of the process representing the graph G_2.

This problem becomes even worse if we consider the processes given in Listing 2.

$$
\begin{array}{lll}
P_1 & = & doX_1 \;\rightarrow\; c\,\text{¡}\,x : T \;\rightarrow\; \text{SKIP} \\
 & & \Box \\
 & & doX_2 \;\rightarrow\; c\,\text{¡}\,x : T \;\rightarrow\; c\,\text{¡}\,x : T \;\rightarrow\; \text{SKIP} \\
P_2 & = & doY_1 \;\rightarrow\; c\,\text{¿}_1\,x : T \;\rightarrow\; \text{SKIP} \\
 & & \Box \\
 & & doY_2 \;\rightarrow\; c\,\text{¿}_1\,x : T \;\rightarrow\; c\,\text{¿}_1\,x : T \;\rightarrow\; \text{SKIP} \\
P_{12} & = & P_1 \Updownarrow P_2
\end{array}
$$

Listing 2: The ambiguity of a writing process and a reading process via the same channel.

The graph representing the processes of Listing 2 contains a path represented by the trace $doX_1 \rightarrow c\,\text{¡}\,x : T \rightarrow doY_2 \rightarrow c\,\text{¿}_1\,x : T \rightarrow c\,\text{¿}_1\,x : T \rightarrow$ SKIP (the thick and dotted arrows in Figure 4[4]), which is obviously wrong. But the dashed and dotted arrows in Figure 4 represent a trace that has to be possible. The problem lies in the black vertex in Figure 4, that allows two traces to be possible with a different number of write actions.

Remark 3. *The problem described in Figure 4 also occurs in DVRSP. For this reason, we redefine DVRSP in a similar fashion as EVRSP in Appendix A.*

[4]For ease of reading the not-relevant labels are removed in Figure 4.

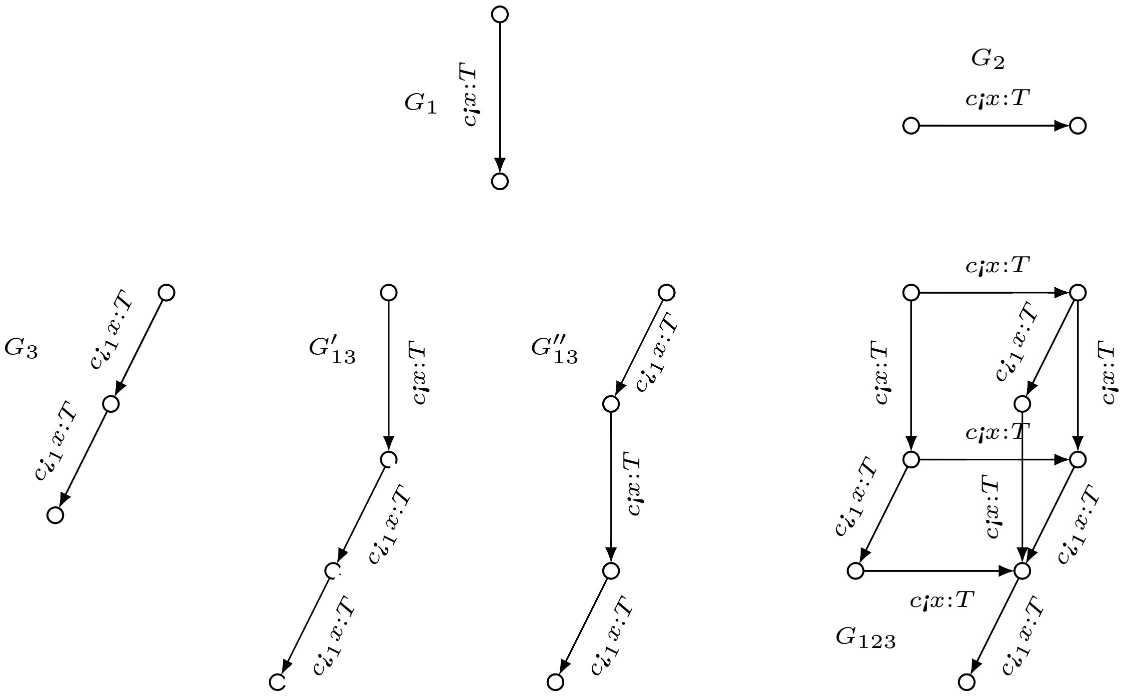


Figure 3. Graphs $G_1, G_2, G_3, G_{123} = \overset{3}{\underset{i=1}{\overset{\diamond}{\boxbslash}}} G_i$, and $G'_{13} = G_1 \overset{\diamond}{\boxbslash} G_3$ or $G''_{13} = G_1 \overset{\diamond}{\boxbslash} G_3$ representing processes P_1, P_2, P_3, P_{123} and P_{13} (Listing 1).

Because the graphs G_1 and G_2 are consistent according to the definition of consistency under VRSP, we have to adjust that definition incorporating the number of writes and reads in each path from the source to the sink of a graph.

The number of occurrences of a write action $c¡x : T$ in the path P_i, is called the *path write cardinality* of a path with respect to $c¡x : T$, denoted as $P_i(c¡x : T)$.

The number of occurrences of a read action $c¿_n x : T$ in the path P_i, is called the *path read cardinality* of a path with respect to $c¿_n x : T$, denoted as $P_i(c¿_n x : T)$.

Graphs G_i and G_j are consistent, denoted as $G_i \doteqdot G_j$, if and only if the following three requirements hold:

1. $\rho_{G_i}(G_i \boxbslash G_j) \cong G_j$ and $\rho_{G_j}(G_i \boxbslash G_j) \cong G_i$.
2. $S'_{G_i \boxbslash G_j} = S'_{G_i} \times S'_{G_j}$ and $S''_{G_i \boxbslash G_j} = S''_{G_i} \times S''_{G_j}$.
3. Whenever P_m, P_n are paths from the source to the sink of G_i $(G_j, G_i \overset{\diamond}{\boxbslash} G_j)$, $P_m(c¡\text{x:T}) = P_n(c¡x : T)$ and $P_m(c¿_k\text{x:T}) = P_n(c¿_k x : T)$.

Obviously the graphs representing the processes in Listing 2 are not consistent. But the processes in Listing 1 are consistent and therefore EVRSP has to determine the order of the read actions with respect to the write actions.

For EVRSP whenever two processes contain identical ¡-actions, these actions are treated asynchronously. For indexed ¿-actions, the index makes the ¿-actions different and therefore EVRSP handles these actions identically to DVRSP. Hence, VRSP must be extended to handle the ¡-actions for the graphs representing different processes only.

The Extended Dot Vertex-Removing Synchronised Product (EVRSP) of G_i and G_j, $G_i \overset{\diamond}{\boxbslash} G_j$ is constructed in two stages, where the definition of the intermediate stage of DVRSP is identical to the intermediate stage of EVRSP, $G_i \overset{\bullet}{\boxtimes} G_j = G_i \overset{\diamond}{\boxtimes} G_j$, with

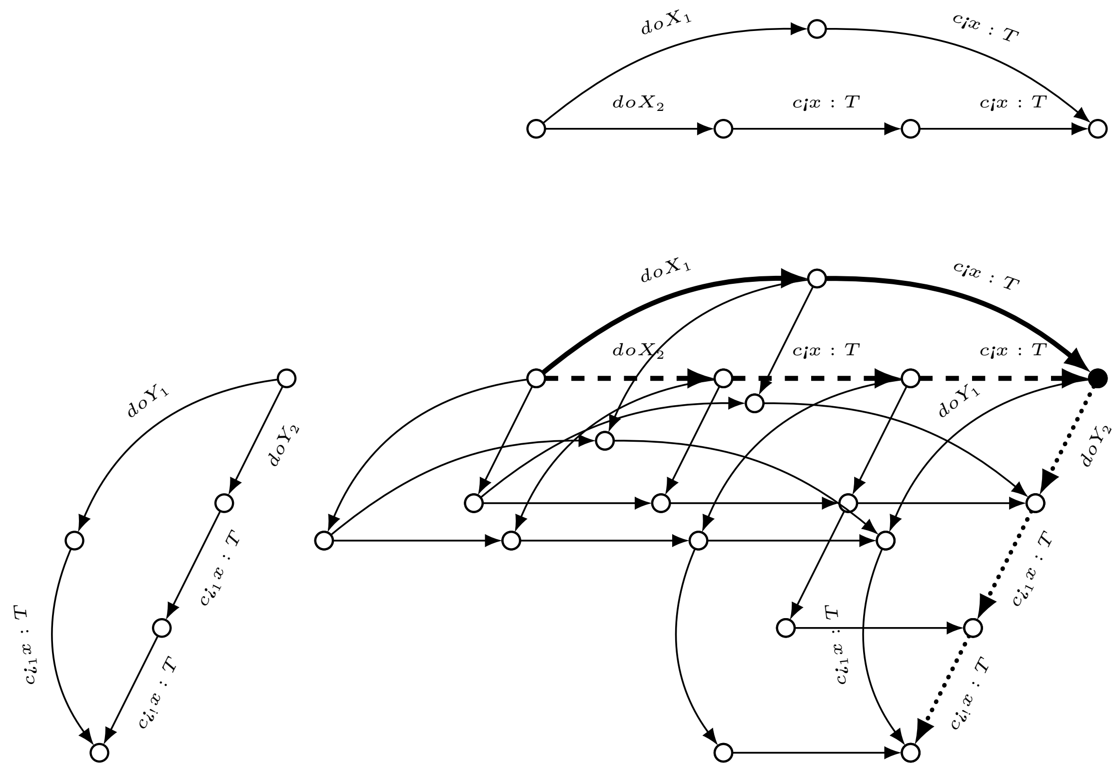


Figure 4. Graphs G_1, G_2 and $G_{12} = \mathop{\overset{\diamond}{\boxtimes}}_{i=1}^{2} G_i$ representing processes P_1, P_2 and P_{12} of Listing 2.

- $v_x w_x \in A_{i,j}$ is an arc with operator $¿_n \in l(v_x w_x) = l_r$,
- P_n is a path from the source of $G_i \overset{\diamond}{\boxtimes} G_j$ through w_x,
- P_m is the path from the source to the sink of $G_i \overset{\diamond}{\boxtimes} G_j$.

Again, we modify the Cartesian product $G_i \square G_j$ according to the existence of synchronising arcs, but now with the extra constraint that labels containing a ¡ character are asynchronous i.e., pairs of arcs with the same label pair without a ¡ character, with one arc in G_i and one arc in G_j.

The first step in this modification consists of ignoring the synchronising arcs while forming arcs in the product, but additionally combining pairs of synchronising arcs of G_i and G_j into one arc, yielding the intermediate product which we denote by $G_i \overset{\diamond}{\boxtimes} G_j$. To be more precise, $G_i \overset{\diamond}{\boxtimes} G_j$ is obtained from $G_i \square G_j$ by first ignoring all except for the so-called asynchronous arcs, i.e., by only maintaining all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_j = w_j$ and $\lambda(a) \notin L_j$ or $v_j = w_j$ and $\lambda(a) \in L_j$ and $¡ \in l(a)$, as well as all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_i = w_i$ and $\lambda(a) \notin L_i$ or $v_i = w_i$ and $\lambda(a) \in L_i$ and $¡ \in l(a)$. This set of arcs is denoted by $A^a_{i,j}$. Additionally, we add arcs that replace synchronising pairs $a_i \in A_i$ and $a_j \in A_j$ with $\lambda(a_i) = \lambda(a_j)$ and $¡ \notin l(a_j)$. If $\mu(a_i) = (v_i, w_i)$ and $\mu(a_j) = (v_j, w_j)$, such a pair is replaced by an arc $a_{i,j}$ with $\mu(a_{i,j}) = ((v_i, v_j), (w_i, w_j))$ and $\lambda(a_{i,j}) = \lambda(a_i)$ and $¡ \notin l(a_i)$. The set of these so-called synchronous arcs of $G_i \overset{\diamond}{\boxtimes} G_j$ is denoted by $A^s_{i,j}$.

The second step in this modification consists of removing (from $G_i \overset{\diamond}{\boxtimes} G_j$) the vertices $(v_i, v_j) \in V_{i,j}$ and the arcs a with $tail(a) = (v_i, v_j)$, whenever (v_i, v_j) has $level > 0$ in $G_i \square G_j$ and (v_i, v_j) has $level\ 0$ in $G_i \overset{\diamond}{\boxtimes} G_j$ and 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 $¿_n \in l(v_x w_x)$ for which there does not exist at least n related arcs $v_y w_y$ with operator $¡ \in l(v_y w_y)$ with $v_y w_y < v_x w_x$. 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 Vertex Removing Synchronised Product (VRSP) of G_i and G_j, denoted as $G_i \overset{\diamond}{\boxtimes} G_j$.

For $k \geqslant 3$, the VRSP $G_1 \overset{\diamond}{\boxtimes} G_2 \overset{\diamond}{\boxtimes} \cdots \overset{\diamond}{\boxtimes} G_k$ is defined recursively as $((G_1 \overset{\diamond}{\boxtimes} G_2) \overset{\diamond}{\boxtimes} \cdots) \overset{\diamond}{\boxtimes} G_k$.

Remark 4. *Because arcs $v_i w_i$ with $¿ \in l(v_i w_i)$ are indexed, the arcs $v_i w_i$ with different indexes represent asynchronous actions, because they have different labels due to different indexes.*

Remark 5. *The EVRSP allows two or more processes to write a value to the same channel.*

In Figure 5 we give an example that shows the stages of the EVRSP. Figure 5.a

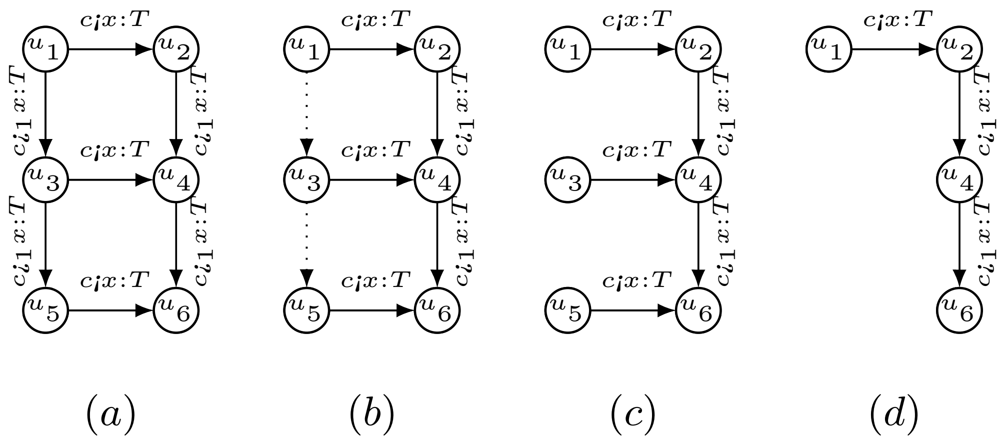


Figure 5. EVRSP from $G_1 \square G_3$ (a), two stages of $G_1 \overset{\diamond}{\boxtimes} G_3$ (b, c), to $G_1 \overset{\diamond}{\boxbslash} G_3$ (d).

shows the Cartesian Product of the graphs G_1, G_3 given in Figure 3. The dotted arcs in Figure 5.b are selected for removal. For the arcs $u_1 u_3$ and $u_3 u_5$ both with label $c¿_1 x{:}T$, there exists a related arc $u_1 u_2$ with label $c¡x{:}T$. Then, because $P_1 = u_1 u_2, P_2 = u_1 \cdots u_6$, $P_1(c¿_1 x : T) = 1 > P_1(c¡x : T) = 0$ and $P_1(c¿_1 x : T) = 1 \leqslant P_2(c¡x : T) = 2$, $u_1 u_3$ and $u_3 u_5$ are removed in Figure 5.c. The last stage of EVRSP removes u_3, u_5 and the arcs that have u_3, u_5 as a tail because $d^-_{G_1 \square G_3}(u_3) = d^-_{G_1 \square G_3}(u_5) = 1$ and $d^-_{G_1 \overset{\diamond}{\boxtimes} G_3}(u_3) = d^-_{G_1 \overset{\diamond}{\boxtimes} G_3}(u_5) = 0$, which leads to Figure 5.d.

3. Case Study of the Extended Half-Synchronous Alphabetised Parallel Operator

To show that the extended operators are useful, we consider a system that runs at 1 kHz, so with a period of 1 ms. The hardware of the system consists of one processor, two controllers, a FPGA, two sensors and two actuators.

A part of the system must be able to perform a controlled emergency stop. This part, running on the processor, consists of a *Controlled Emergency Stop (CES)* thread, two *Application* threads (A_1, A_2) and two *Controller* threads (C_1, C_2).

Assume that the total amount of data used by these threads does not fit in the L2 cache, therefore every context switch leads to a cache flush. This increases the context-switch time [14]. According to [14] due to L2 cache flushes the context-switch time can take up to 1.5 ms for the hardware and software under consideration. In average [14] measured a context-switch time of 3.8 μs.

Taking into account the measured timing for a context switch, we assume that the worst-case context-switch time for our example is 20 μs. Because the CES case study describes a fictive PHRCS, we use estimated guesses for the timing of all actions of the processes, the controllers, the FPGA and the devices.

Each Application process controls the behaviour of one Controller thread. Each Controller process communicates, for example, via memory mapped I/O, with a controller responsible for the behaviour of a sensor and an actuator.

To calculate the values that drive the actuators, the Controller threads interact with an Algorithmic Software process ($Alg.Soft.$). The Algorithmic Software process calculates, for example, the Fast Fourier Transform (FFT) of the data by communicating via memory mapped I/O to an FPGA. The FPGA performs a FFT on the data. This architecture is shown in Figure 6.

Furthermore, assume that

- the controller threads and the algorithmic software thread have priority over the application threads,
- the CES and Application threads have equal priority,
- the Controller threads have equal priority,
- the actions of the CES thread, the Application threads and the Controller threads take 20 μs to execute, this includes context switches, state changes in the threads and the like,
- the Algorithmic Software takes 130 μs to calculate the FFT on each data item, which includes the calculation time of the FPGA. It buffers commands from the Controller threads.
- the Controller takes 80 μs to read the sensor value and 160 μs to write the actuator value to the actuator.

This leads to a simple CSP specification given in Listing 3 using the extended half-synchronous operator, the **¡**-actions and the indexed **¿**$_i$-actions, where the alphabet of CES is CES, the alphabet of A_i is A_i and the alphabet of C_j is C_j.

Remark 6. *The* c_2**¡***stop of* A_1 *and* A_2 *are asynchronous writes. Because both* A_1 *and* A_2 *perform this action and the* C_1 *and* C_2 *read this action only once, one of the writes is not read. This is an example of a writing without reading, which is intended, as the* C_1 *and* C_2 *have to start stopping as soon as possible.*

Remark 7. *In [5] we showed that writing without reading is pointless, because there could be only one writer for several readers. For the extended half-synchronous operator and asynchronous writers with at least one reading action, the Controlled Emergency Stop example shows a smaller model and therefore less execution time, because no buffers are necessary.*

Remark 8. *Because the reads have different indexes, the* C_1, C_2 *do not delay one another.*

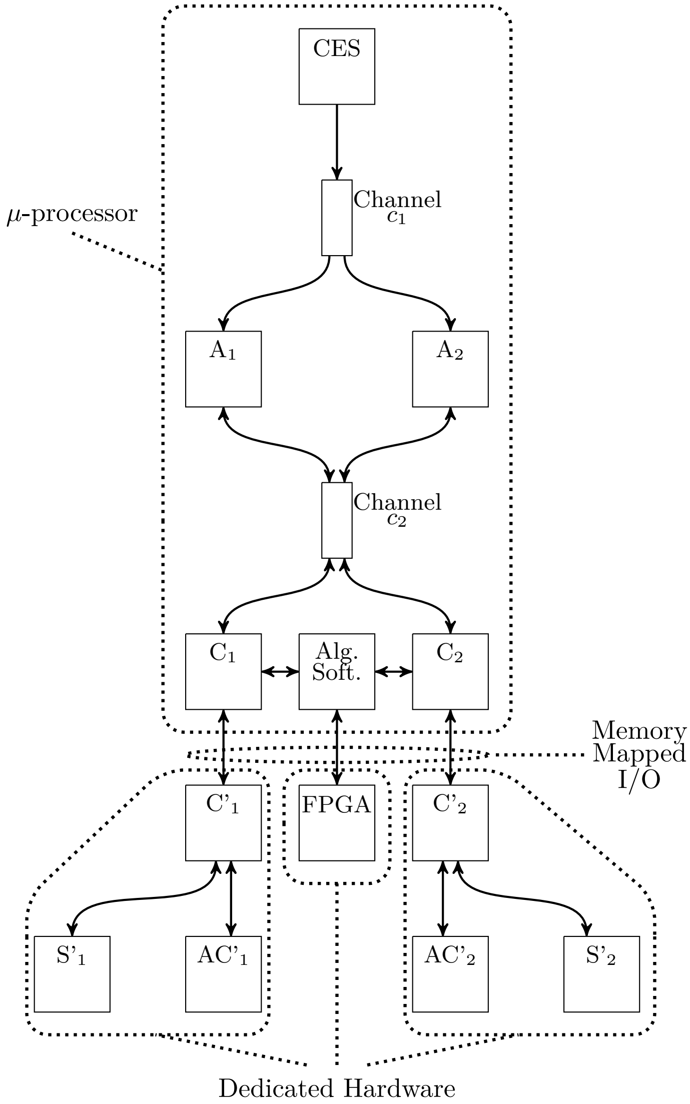


Figure 6. Communication Flow of the Controlled Emergency Stop.

Remark 9. *The c_1-channel is unidirectional because CES only writes to A_1, A_2. The c_2-channel is bidirectional because A_1, A_2 write to C_1, C_2 and vice versa.*

The graphs representing the processes in Listing 3 are given in Figure 7. The behaviour not modelled in Listing 3, the $\cdots$, are left out of Figure 7.

Remark 10. *It is up to the process software to handle the state transitions. This includes the handling of guarded actions, which are labels in the graph.*

The processes C_1, C_2 in Listing 3 are synchronising over the c_2¿$_1$*boot*-action and *waitForNextPeriod*-action. Only the *waitForNextPeriod*-action occurs in all longest paths. But still the worst-case performance is improved by the execution time of one *waitForNextPeriod*-action, together with two context switches. Therefore for the EVRSP of $C_1, C_2, C_1\overset{\diamond}{\boxbslash}C_2$, there is some gain. The memory occupancy is not quadratic with respect to the number of vertices of C_1 and C_2, because of the order that the ¡-actions and ¿-actions impose on the product. For A_1, A_2, CES the gain is better, because both the *ack*-action and *waitForNextPeriod*-action are on all longest paths. For example, in Figure 8 the longest path of $A_1\overset{\diamond}{\boxbslash}A_2$ contains seven arcs, whereas the longest path of A_1 plus the longest path of A_2 is equal to 10.

$$
\begin{aligned}
CES \;=\;& readStatus.s \rightarrow (s == stop; c_1 \text{¡} stop \rightarrow ack \rightarrow writeStatus.boot \rightarrow CES_1\\
& \qquad\qquad\qquad\qquad \Box\\
& \qquad\qquad\qquad\quad s == boot; c_1 \text{¡} boot \rightarrow ack \rightarrow writeStatus.init \rightarrow CES_1)\\
& \qquad\qquad\qquad\qquad \Box\\
& \qquad\qquad\qquad\qquad \ldots\\
& \qquad\qquad\qquad\quad s == \cdots \rightarrow CES_1)\\
CES_1 \;=\;& waitForNextPeriod \rightarrow SKIP
\end{aligned}
$$

$$
\begin{aligned}
A_1 \;=\;& c_1 \text{¿}_1 stop \rightarrow c_2 \text{¡} stop \rightarrow c_2 \text{¿}_1 stopAck_1 \rightarrow A_{11}\\
& \quad \Box\\
& c_1 \text{¿}_1 boot \rightarrow c_2 \text{¡} boot \rightarrow c_2 \text{¿}_1 bootAck_1 \rightarrow A_{11}\\
& \quad \Box\\
& \quad \ldots\\
A_{11} \;=\;& ack \rightarrow waitForNextPeriod \rightarrow SKIP
\end{aligned}
$$

$$
\begin{aligned}
A_2 \;=\;& c_1 \text{¿}_2 stop \rightarrow c_2 \text{¡} stop \rightarrow c_2 \text{¿}_2 stopAck_2 \rightarrow A_{21}\\
& \quad \Box\\
& c_1 \text{¿}_1 boot \rightarrow c_2 \text{¡} boot \rightarrow c_2 \text{¿}_1 bootAck_2 \rightarrow A_{21}\\
& \quad \Box\\
& \quad \ldots\\
A_{21} \;=\;& ack \rightarrow waitForNextPeriod \rightarrow SKIP
\end{aligned}
$$

$$
\begin{aligned}
C_1 \;=\;& c_2 \text{¿}_1 stop \rightarrow readSensor.s_1 \rightarrow writeAlgSoft.s_1 \rightarrow readAlgSoft.v_1 \rightarrow\\
& \quad writeAC_1.v_1 \rightarrow readAckAC1 \rightarrow c_2 \text{¡} stopAck_1 \rightarrow C_{11}\\
& \quad \Box\\
& \quad c_2 \text{¿}_1 boot \rightarrow resetSensorS_1 \rightarrow writeInitAC_1 \rightarrow readAckAC1 \rightarrow c_2 \text{¡} bootAck_1\\
& \quad \rightarrow C_{11}\\
& \quad \Box\\
& \quad \ldots\\
C_{11} \;=\;& waitForNextPeriod \rightarrow SKIP
\end{aligned}
$$

$$
\begin{aligned}
C_2 \;=\;& c_2 \text{¿}_2 stop \rightarrow readSensor.s_2 \rightarrow writeAlgSoft.s_2 \rightarrow readAlgSoft.v_2 \rightarrow\\
& \quad writeAC_2.v_2 \rightarrow readAckAC2 \rightarrow c_2 \text{¡} stopAck_2 \rightarrow C_{21}\\
& \quad \Box\\
& \quad c_2 \text{¿}_1 boot \rightarrow resetSensorS_2 \rightarrow writeInitAC_2 \rightarrow readAckAC2 \rightarrow c_2 \text{¡} bootAck_2\\
& \quad \rightarrow C_{21}\\
& \quad \Box\\
& \quad \ldots\\
C_{21} \;=\;& waitForNextPeriod \rightarrow SKIP
\end{aligned}
$$

$$
System \;=\; CES \;{}_{CES} \Updownarrow_{A_1 \cup A_2 \cup C_1 \cup C_2} ((A_1 \;{}_{A_1} \Updownarrow_{A_2}\; A_2)_{A_1 \cup A_2} \Updownarrow_{C_1 \cup C_2} (C_1 \;{}_{C_1} \Updownarrow_{C_2}\; C_2))
$$

Listing 3: The Controlled Emergency Stop Process Specification.

This reduces the overhead of synchronisation considerably. Also the memory occupancy with respect to the number of vertices and arcs is 26 vertices and 39 arcs for $A_1 \overset{\diamond}{\boxbslash} A_2$ and 16 vertices and 16 arcs (two times 8 vertices and 8 arcs) for A_1 and A_2. The polynomial space complexity in this case is arguably reasonable, considering that the space complexity for the Cartesian product without synchronisation would be exponential.

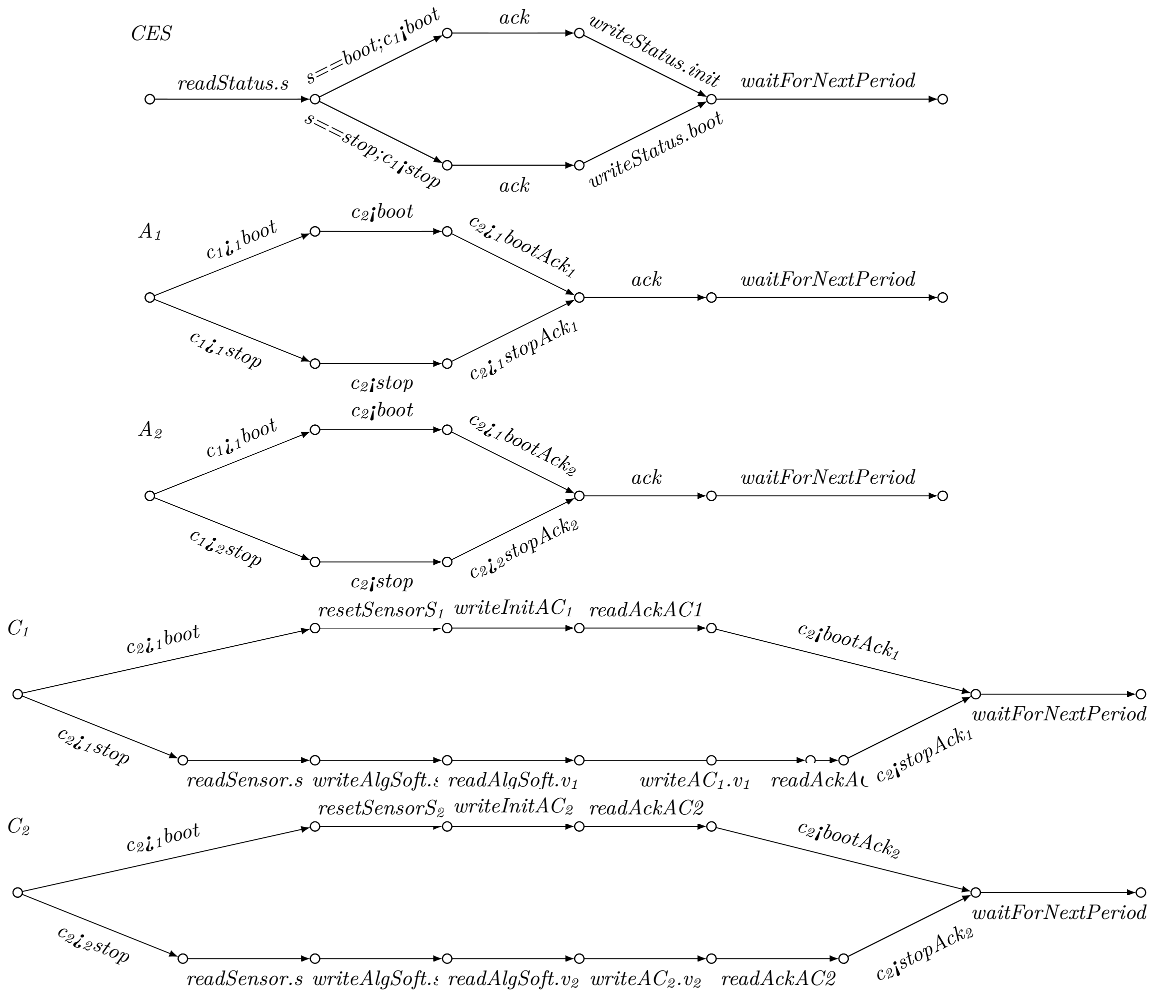


Figure 7. Graphs CES, A_1,A_2, C_1 and C_2.

All other products are left out because the number of vertices these graphs contain makes the figures unreadable.

One trace, due to a *stop*-action shown in Listing 3, is of particular interest because it shows the longest path in the combined graph representing the *System* process for a *stop*-action and a *boot*-action.

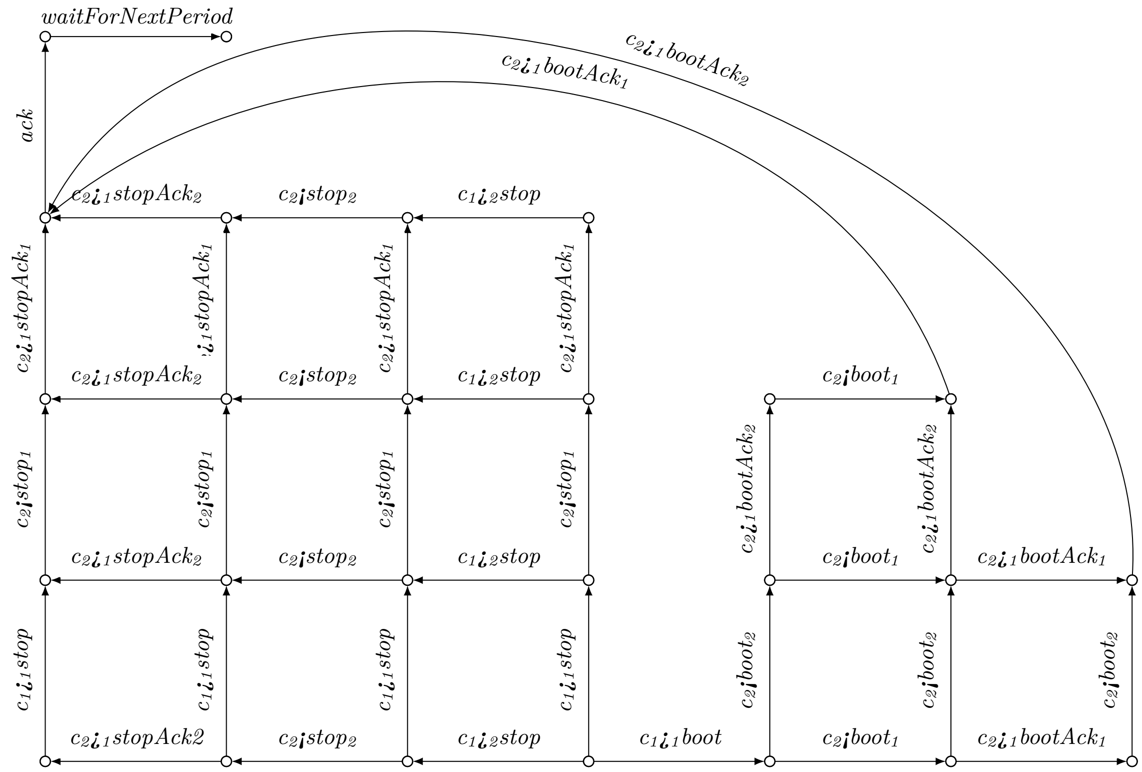
$readStatus.s, 20 \rightarrow c_1\text{¡}stop, 20 \rightarrow c_1\text{¿}_1 stop, 20 \rightarrow c_1\text{¿}_2 stop, 20 \rightarrow c_2\text{¡}stop, 20 \rightarrow c_2\text{¿}_1 stop, 20 \rightarrow readSensor.s_1, 80 \rightarrow c_2\text{¿}_2 stop, 20 \rightarrow readSensor.s_2, 80 \rightarrow c_2\text{¡}stop, 20 \rightarrow writeAlgSoft.s_1, 130 \rightarrow writeAlgSoft.s_2, 130 \rightarrow readAlgSoft.v_1, 20 \rightarrow writeAC_1.v_1, 160 \rightarrow readAckAC1, 20 \rightarrow c_2\text{¡}stopAck_1, 20 \rightarrow c_2\text{¿}_1 stopAck_1, 20 \rightarrow readAlgSoft.v_2, 20 \rightarrow writeAC_2.v_2, 160 \rightarrow readAckAC2, 20 \rightarrow c_2\text{¡}stopAck_2, 20 \rightarrow c_2\text{¿}_1 stopAck_2, 20 \rightarrow ack, 60$[5] $\rightarrow writeStatus.boot, 20 \rightarrow waitForNextPeriod, 100$ [6]$\rightarrow SKIP$

Listing 4: Trace of the CES.

The worst-case execution time is the summation over the time part of the labels. To stop both the actuators in our example, this adds up to 1240 μs. Because the controllers

[5]The processes CES, A_1 and A_2 synchronise over the *ack*-action. Therefore the execution time adds up to 60 μs.

[6]The processes CES, A_1, A_2, C_1 and C_2 synchronise over the $waitForNextPeriod$-action. Therefore the execution time adds up to 100 μs.



$A_1 \overset{\diamond}{\boxbslash} A_2$

Figure 8. Graph $A_1 \overset{\diamond}{\boxbslash} A_2$.

for the sensors and actuators, and the FPGA are running partially in parallel, the execution time is 940 μs.

The shortest run time of the controllers and the FPGA is 260 μs. This happens when the controllers and the FPGA are running in parallel at the same time. Therefore the best case execution time is 880 μs.

Although there is no deadline-miss in this fictive example for the stop part of the CES, when the model would support the writing to and reading from buffers, the best case execution time would increase. For example, adding three buffers with each two actions to perform, there is an extra 120 μs execution time. This leads to an execution time in the best case of 1000 μs. Then a deadline-miss seems inevitable.

In Figure 9 the time line of a possible trace of the *stop* part of the CES is given. Each gray block represents the time that the thread is executing. The label of each hardware related action contains the overall time. If applicable, this includes the time the hardware needs to reply. The dashed arrows represent a call to the hardware and the reply from the hardware.

As Figure 9 shows, the stop part of the CES takes 940 μs to execute. This can be improved by using the EVRSP of the graphs, $SynchronisedSystem = CES \overset{\diamond}{\boxbslash} A_1 \overset{\diamond}{\boxbslash} A_2 \overset{\diamond}{\boxbslash} C_1 \overset{\diamond}{\boxbslash} C_2$. The actions that synchronise are $waitForNextPeriod$ and ack, therefore the processor needs at most 820 μs to execute the thread represented by the graph *SynchronisedSystem*.

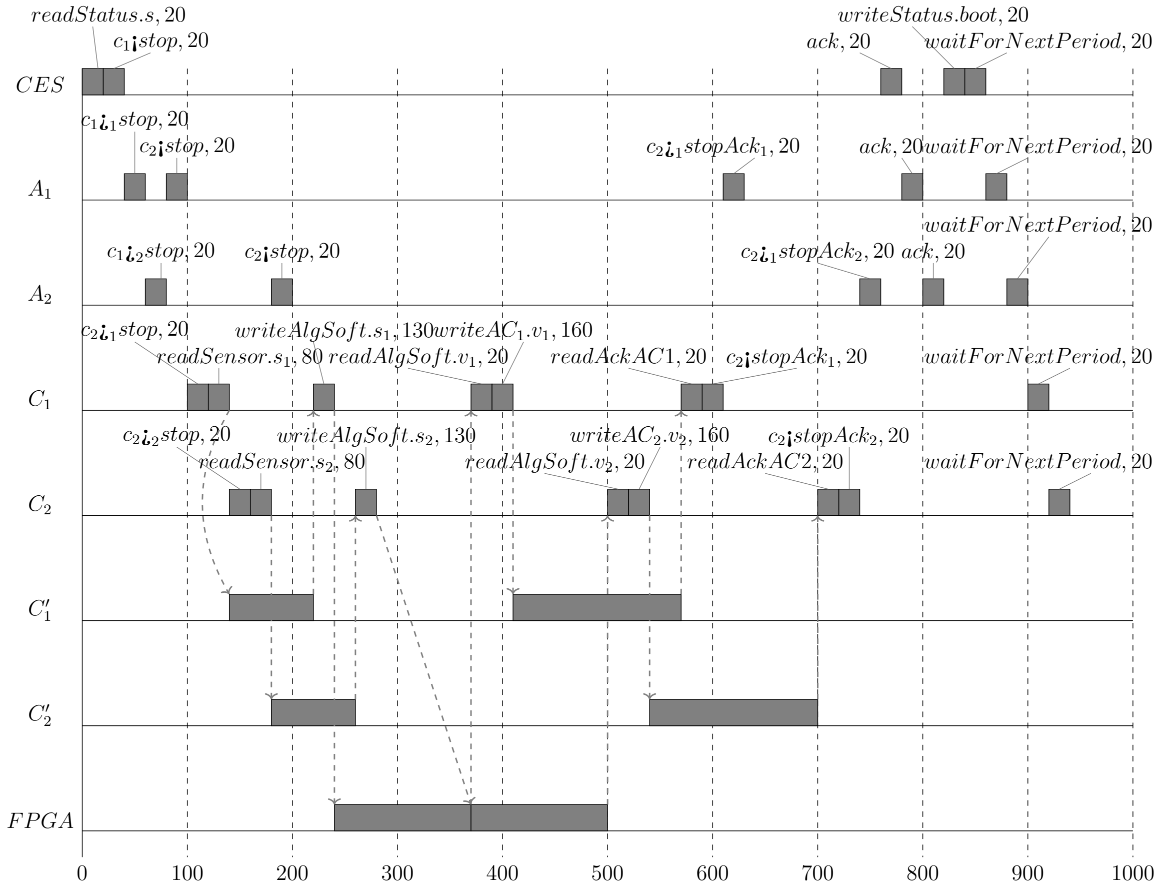


Figure 9. Time line of the *stop*-part of the Controlled Emergency Stop.

The improvement with respect to timeliness can be easily seen when we model the CES using standard CSP as shown in Figure 10, although this example gives an improvement of only 50μs. This is because we do not have to model buffers as well due to the simplicity of the example. The $c_2.stop, 20$ actions of A_1, A_2, C_1 and C_2 are executed atomically, therefore it is immaterial which of the processes A_1, A_2, C_1 and C_2 executes the action $c_2.stop, 20$ first. In fact the priority inheritance protocol [15] is implemented for the processes A_1, A_2 and C_1, C_2 for the action $c_2.stop, 20$.

4. Discussion and Conclusions

In this paper we have discussed an extension of the ${}_X\Downarrow_Y$ operator, the new ${}_X\Updownarrow_Y$ operator and the ¡-*action* together with the new ¿$_i$-*action*, that delay the reading of a process from a buffer. The ${}_X\Updownarrow_Y$ operator together with the ¡-*action* and ¿$_i$-*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 five advantages of the ${}_X\Updownarrow_Y$ operator in combination with the EVRSP with respect to standard CSP:

- it eases the design by taking away the burden of separating the writing actions and reading actions in time, which eliminates the necessity of a buffer,
- it gives maximum flexibility by indexing the reading actions,
- it allows multiple write actions to the same channel,

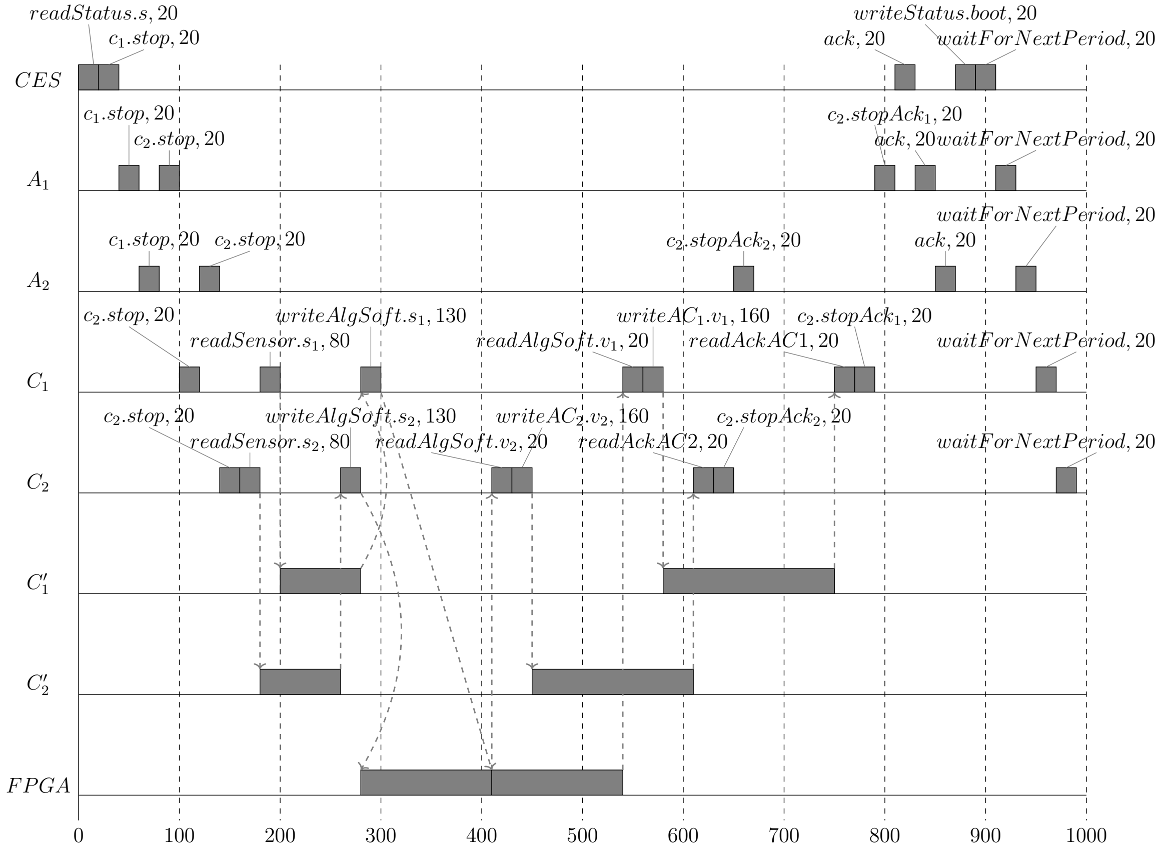


Figure 10. Time line of the *stop*-part of the Controlled Emergency Stop without asynchronous readers and writers.

- the length of the longest path is reduced, if the writing actions and reading actions 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 makes the design less error prone and therefore the design phase needs less time. The absence of a buffer leads to less actions that have to be performed by the involved threads and therefore to a reduction of the utilisation of the processor,

Furthermore, the overall design cycle gains because the improved description on design level leads to less effort for the implementation and less effort for testing, achieved by the second and third advantage.

The fourth advantage is due to EVRSP only and leads to an application that needs less execution time,

The fifth advantage is due to a reduction of the end-to-end execution time during one period and therefore leads to an application for which the possibility of a deadline-miss is reduced.

Of course there is also a drawback, when using EVRSP. The designer has to figure out whether the disconnection of reads and writes leads to a greater reduction of the end-to-end execution time in one period than using synchronous writing actions and reading actions.

5. Future Work

With this contribution, together with our contributions [1,2,3,5], we have dealt with the graph theoretical aspects of improving the performance of PHRCSs by reduction of the number of context switches and reducing the end-to-end execution time.

But several issues in our design cycle have not been addressed yet. With respect to the system architecture we have described in [2] the transformation functions that transform a graph into a algebraic specification, but they are not defined yet. Furthermore, although partially implemented by [16], there is no fully operational tool-chain that automatically, based on the process algebraic specification, produces software which can be compiled and built, thereby producing a set of Periodic Hard Real-Time Control Processes (PHRCPs). Also tooling that supports the choice for synchronous writing actions and reading actions versus EVRSP has to be developed.

So far we have used a fixed period of 1 ms. Allowing the PHRCP to have different periods and taking into account the priority of processes will lead to a not explored area of EVRSP. All these issues will introduce scheduling problems that have to be solved by an adapted version of EVRSP.

The end result to go for could be allowing cyclic and non-deterministic process specifications and study the impact on EVRSP.

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] 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.

[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] Antoon H. Boode, Hajo Broersma, and Jan 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.

[4] 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.

[5] A. H. Boode and J. F. Broenink. Asynchronous Readers and Writers. In *Communicating Process Architectures 2016, Copenhagen, Denmark*, 38th WoTUG conference on concurrent and parallel programming, pages 125–137, Bicester, August 2016. Open Channel Publishing Ltd.

[6] 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.

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

[8] Peter H. Welch and Jeremy M. R. Martin. A CSP model for java multithreading. In *International Symposium on Software Engineering for Parallel and Distributed Systems, PDSE 2000, Limerick, Ireland, June 10-11, 2000*, pages 114–122, 2000.

[9] J.A. Bondy and U.S.R. Murty. *Graph Theory.* Springer, Berlin, 2008.

[10] Richard Hammack, Wilfried Imrich, and Sandi Klavžar. *Handbook of product graphs*. Discrete Mathematics and its Applications (Boca Raton). CRC Press, Boca Raton, FL, second edition, 2011. With a foreword by Peter Winkler.

[11] P. Hell and J. Nešetřil. *Graphs and Homomorphisms*. Oxford Lecture Series in Mathematics and Its Applications. OUP Oxford, 2004.

[12] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

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

[14] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, September 1990.

[16] M.P. de Boer. *Implementation of Periodic Hard Real-Time Processes, Bachelor Thesis*. University of Twente, June 2016.

Appendix

A. Redefinition of the Dot Vertex-Removing Synchronised Product (DVRSP)

The Dot Vertex-Removing Synchronised Product (DVRSP) of G_i and G_j, $G_i \dot{\boxbslash} G_j$ is a modification of the Cartesian product $G_i \square G_j$ according to the existence of synchronising arcs, but now with two extra constraints that labels of the type $c¡x : T$ are allowed in only one process i.e., pairs of arcs with the same label pair $c¡x : T$, with one arc in G_i and one arc in G_j are inhibited, and that labels of the type $c¡x : T$ and $c¿x : T$ are asynchronous i.e., pairs of arcs with one arc with the label $c¡x : T$ in G_i and the other arc with the label $c¿x : T$ in G_j are asynchronous.

$$\begin{aligned}\text{Assume that } & a \in A_{i,j}, \mu(a) = ((v_x, v_y), (w_x, w_y)) \text{ is an arc with } ¿ \in l(a) = l_r, \\ & P_n \text{ is a path from the source of } G_i \dot{\boxtimes} G_j \text{ through } (w_x, w_y), \\ & P_m \text{ is the path from the source to the sink of } G_i \dot{\boxtimes} G_j.\end{aligned}$$

The first step in this modification consists of ignoring the synchronising arcs while forming arcs in the product, but additionally combining pairs of synchronising arcs of G_i and G_j into one arc, yielding the intermediate product which we denote by $G_i \dot{\boxtimes} G_j$. To be more precise, $G_i \dot{\boxtimes} G_j$ is obtained from $G_i \square G_j$ by first ignoring all except for the so-called asynchronous arcs, i.e., by only maintaining all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_j = w_j$ and $\lambda(a) \notin L_j$, as well as all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_i = w_i$ and $\lambda(a) \notin L_i$.

This set of arcs is denoted by $A^a_{i,j}$. Additionally, we add arcs that replace synchronising pairs $a_i \in A_i$ and $a_j \in A_j$ with $\lambda(a_i) = \lambda(a_j)$ and $¡ \notin l(a_i)\,(l(a_j))$. If $\mu(a_i) = (v_i, w_i)$ and $\mu(a_j) = (v_j, w_j)$, such a pair is replaced by an arc $a_{i,j}$ with $\mu(a_{i,j}) = ((v_i, v_j), (w_i, w_j))$ and $\lambda(a_{i,j}) = \lambda(a_i)$ and $¡ \notin l(a_i)$. The set of these so-called synchronous arcs of $G_i \dot{\boxtimes} G_j$ is denoted by $A^s_{i,j}$.

The second step in this modification consists of removing (from $G_i \dot{\boxtimes} G_j$) the vertices $(v_i, v_j) \in V_{i,j}$ and the arcs a with $tail(a) = (v_i, v_j)$, whenever (v_i, v_j) has $level > 0$ in $G_i \square G_j$ and (v_i, v_j) has $level\,0$ in $G_i \dot{\boxtimes} G_j$, and all arcs $a_{x_1,y_1} \in A_{i,j}, \mu(a_{x_1,y_1}) = ((v_{x_1}, v_{y_1}), (w_{x_1}, w_{y_1}))$ with $l(a_{x_1,y_1}) = l_r$ for which there exists a related arc $a_{x_2,y_2} \in A_{i,j}, \mu(a_{x_2,y_2}) = ((v_{x_2}, v_{y_2}), (w_{x_2}, w_{y_2}))$ with label l_w, where $P_n(l_r) > P_n(l_w)$ and $P_n(l_r) \leqslant P_m(l_w)$. 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 $G_i \overset{\bullet}{\boxtimes} G_j$. For $k \geqslant 3$, the VRSP $G_1 \overset{\bullet}{\boxtimes} G_2 \overset{\bullet}{\boxtimes} \cdots \overset{\bullet}{\boxtimes} G_k$ is defined recursively as $((G_1 \overset{\bullet}{\boxtimes} G_2) \overset{\bullet}{\boxtimes} \cdots) \overset{\bullet}{\boxtimes} G_k$.

Remark 11. *The definition of DVRSP inhibits identical write actions to the same channel, i.e.* $\dfrac{Q_i \overset{c\,¡\,x:T}{\rightarrow} Q_i',\ Q_j \overset{c\,¡\,x:T}{\rightarrow} Q_j'}{SKIP}, i \neq j$ *is ensured.*