Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Improving the Performance of Periodic Real-time Processes: a Graph Theoretical Approach

Antoon H. BOODE [a,1,2], Hajo BROERSMA [b] and Jan F. BROENINK [a]
[a] *Robotics and Mechatronics,*
[b] *Formal Methods and Tools,*
Faculty of Electrical Engineering, Mathematics and Computer Science,
University of Twente, The Netherlands
{A.H.Boode, H.J.Broersma, J.F.Broenink}@utwente.nl

Abstract. In this paper the performance gain obtained by combining parallel periodic real-time processes is elaborated. In certain single-core mono-processor configurations, for example embedded control systems in robotics comprising many short processes, process context switches may consume a considerable amount of the available processing power. For this reason it can be advantageous to combine processes, to reduce the number of context switches and thereby increase the performance of the application. As we consider robotic applications only, often consisting of processes with identical periods, release times and deadlines, we restrict these configurations to periodic real-time processes executing on a single-core mono-processor. By graph theoretical concepts and means, we provide necessary and sufficient conditions so that the number of context switches can be reduced by combining synchronising processes.

Keywords. CSP, graph transformations, acyclic multi-graphs, real-time periodic processes, synchronised products

Introduction

In certain single-core mono-processor configurations, for example embedded control systems in robotics comprising many short processes, process context switches may consume a considerable amount of the available processing power.

Li et al. [1] show, that the average cost of a context switch varies from $3.8\,\mu$s to over 1 ms. Veldhuizen [2] shows that the cost of a context switch is on average $7.7\,\mu s$. Clearly these figures depend on the hardware and software being used[3]. To what extent a system is suffering from context switches depends roughly on the ratio between the context switch and the process action; the higher the time consumption of an action, the less relevant the time consumption of the context switch.

As we are considering systems with many short processes, it can be advantageous to combine processes, in order to reduce the number of context switches, thereby increasing the performance of the application. In this paper we restrict these configurations to robotic appli-

[1]Corresponding Author: *Ton Boode, Robotics and Mechatronics, Faculty EEMCS, University of Twente, P.O. Box 217 7500 AE Enschede, The Netherlands.* E-mail: A.H.Boode@utwente.nl.
[2]funded by InHolland University of Applied Sciences, Alkmaar, The Netherlands
[3]occam-π context switch overheads, under the KRoC CCSP multicore scheduler [3], are of the order of 100 nanoseconds.

cations. We consider periodic real-time processes executing on a single-core mono-processor, because robotic applications (like embedded control systems) often consist of processes with identical periods, release times and deadlines. The processes typically have a period of 1 ms. This observation makes it reasonable to assume that the release time, the periods and the deadlines for the constituent processes of the application are the same. As we consider periodic real-time processes, for every process activity (i.e. action), there must be an upper bound for which the action has finished executing; otherwise one cannot guarantee the timeliness of the process. As an example, consider 100 very short processes, containing in average 3 actions, running at 1 KHz, so a period of 1 ms. Using the minimum context switch time consumption given by Li [1], the 300 context switches will need more than the available processing time in one period.

When looking at programs, we distinguish between the specification level and the execution level. On the one hand, there is the specification of a set of parallel processes (for example, in CSP [4]); on the other hand, there is the execution of processes representing the specification, on a computer system, running under an operating system.

On specification level, a process defines a series of actions. Processes sharing the same action can only perform this action if all processes sharing this action are ready to perform this action; this is atomic and performed as one action.

On execution level, as soon as a process has to synchronise with another process[4], a context switch has to be executed, to let the execution be continued by that other process. Such a context switch consumes time. One can reduce the number of these synchronisation related context switches by combining communicating processes.

At specification level, a set of parallel real-time processes can be represented by a graph consisting of several components. A single process is represented by one component, which is a finite labelled weighted directed multi-graph[5], consisting of vertices, arcs between pairs of vertices and labels associated with the arcs. A label is a string representing the (name of an) action and a number representing the worst-case execution time of the action. With each name exactly one worst-case execution time value is associated. Our interpretation of a component, representing a process, is that the vertices represent states and the arcs together with their labels represent the actions that are necessary to move from one state to another. Components have different arc sets, but some of their arcs may have the same labels, meaning that they represent the same action.

The execution of a process is, from a graph-theoretical point of view, represented by a series of arcs: a path through the graph. In process terms this is called a *trace*. Such a path has a length, which is the summation over the worst-case execution time values of the labels associated with the arcs in the path. Our goal is to reduce the worst-case execution time of the set of parallel processes, which is represented by the summation over the maximum path length of each graph, by combining synchronising processes. In graph-theoretical terms this leads to combining graphs, using notions like the Cartesian product of graphs and the synchronised product of graphs.

Via a design methodology, a process specification has to be transformed into a program. We insert into this transformation three steps, of which this paper describes the second one. Firstly, we transform the process specification into a set of graphs, secondly, where possible and meaningful (in terms of performance gain), we take synchronised products of subsets of the set of graphs, and thirdly, this set of synchronised products is transformed into a process specification.

[4]To synchronise actions, both processes have to do extra work and at least one of them will have to yield the processor (assuming single-core execution), causing a context switch.

[5]These graphs are (slightly) more general than *labelled transition systems* in that they may have more than one starting and finishing points (used in intermediate stages of the graph transformations described later).

The paper is organised as follows. Before we specify the model that we use to analyse the performance of periodic real-time processes, we introduce the necessary graph-theoretic and process algebra terminology in Section 1. In Section 2, we introduce periodic real-time processes as finite labelled weighted directed acyclic multi-graphs, and we present an overview of synchronised products. In Section 3, we discuss the transformation of a set of graphs to its Cartesian product, where we show that the longest path length for a set of graphs is identical to the longest path length of the Cartesian product of this set of graphs. In Section 4 the synchronisation constraints (disregarded by the Cartesian product) are met by means of the weak synchronised product of a set of parallel processes. In Section 5 the reduced weak synchronised product of a set of parallel processes is introduced, where not-specified behaviour represented by the Cartesian product is removed. In Section 6 the synchronised product is introduced and necessary and sufficient conditions are proved for the longest path length of that product to be less than the longest path length from the original set of graphs (representing parallel processes). We finish with Section 7, where we give our conclusions followed by a discussion and ideas for future work.

1. Terminology

We use [5] and [6] for terminology and notation on graphs and processes not defined here and consider finite labelled weighted directed acyclic multi-graphs only.

So, if we use H to denote a graph, we will always mean a finite labelled weighted directed acyclic multi-graph. Thus H consists of a set of vertices V, a multi-set of arcs A, and a mapping $\lambda : A \to L$, where L is a set of label pairs[6]. An arc $a \in A$ which is directed from a vertex $u \in V$ (the tail) to a vertex $v \in V$ (the head) will usually be denoted as $a = uv$; the reverse arc will be denoted as vu. Note that we allow multiple arcs from u to v, but that we do not allow uv and vu to be present in the same graph. 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 $l(a)$. If an arc has multiplicity $k > 1$, then all copies have different labels, otherwise we could replace two copies of an arc with identical labels by one arc, because they represent exactly the same action at the same stage of the process. If two arcs $a, b \in A$ have labels $\lambda(a) = (l(a), t(a))$ and $\lambda(b) = (l(b), t(b))$ such that $l(a) = l(b)$, then this implies that $t(a) = t(b)$; this follows since $l(a) = l(b)$ means that the arcs a and b represent the same action at different stages of a process.

A directed path in H is a sequence of *distinct* vertices $v_1 v_2 \ldots v_k$ of H such that $v_j v_{j+1} \in A$ for $j = 1, \ldots, k-1$. The length of a path $v_1 v_2 \ldots v_m$ is defined as $\sum_{i=1}^{m-1} t(v_i v_{i+1})$. A directed path defines a *total ordering* on its arcs: $v_1 v_2 < v_2 v_3 < \ldots < v_{k-1} v_k$.

A directed cycle is a directed path $v_1 v_2 \ldots v_k$ together with an additional arc $v_k v_1$, and is denoted by $v_1 v_2 \ldots v_k v_1$. An acyclic graph does not contain any directed cycles.

We consider finite acyclic graphs, H, only. In general, such a graph consists of several components, where each component, H_i, is *weakly connected* (i.e. all vertices are connected by sequences of arcs, ignoring arc directions) and corresponds to one sequential process. For such components, $\ell(H_i)$ is defined as the maximum length taken over all directed paths in H_i. For the whole graph, which corresponds to a parallel set of sequential processes that must each run to completion, the maximum path length, $\ell(H)$, is the sum of all the individual $\ell(H_i)$. A *partial ordering* on the arcs of a weakly connected graph is induced from the total orderings of its directed paths: $a < b$ if and only if a and b are so ordered in some directed path within the graph.

[6]We shall also use the notation $V(H)$ and $A(H)$ to denote the vertices and arcs for any graph H.

Time and processes are thoroughly described in CSP (for example, by Schneider [6]). Our view of time in a process is that each action takes some time to execute and this time is directly linked to the label of the action. For every process P_i, the actions of the process constitute the process alphabet set A_{P_i}, which consists of labels. A label in a process is identical to a label in a graph: both are identical strings of characters with identical associated values.

For components H_i and H_j, an arc a_i with label $\lambda(a_i)$ in component H_i is a synchronising arc with respect to components H_i and H_j, if and only if there exists an arc a_j with label $\lambda(a_j)$ in component H_j and $\lambda(a_i) = \lambda(a_j)$. If it is clear from the context, we omit the *'with respect to components H_i and H_j'* part of the definition.

Viewed as CSP processes, components are combined in parallel using the CSP *alphabetised* parallel operator with alphabet sets defined by the labels on their respective arcs. For an arc of a component H_i whose label does not occur on an arc of another component H_j, the corresponding action is not blocked from execution.

For a set of parallel processes that contains a deadlock, the graph H representing this set is said to be ill-defined or inconsistent. An example of such a set of parallel processes is called a pathological case.

Components $H_i = (V_i, A_i, \{\lambda(a) | a \in A_i\})$ and $H_j = (V_j, A_j, \{\lambda(a) | a \in A_j\})$ are said to be independent if and only if $\{\lambda(a) | a \in A_i\} \bigcap \{\lambda(a) | a \in A_j\} = \varnothing$.

The in-degree (out-degree) of a vertex v in a graph H is defined as the number of arcs with head v (tail v) and denoted by $d_H^-(v)$ $(d_H^+(v))$.

The Cartesian product $H_1 \times H_2$ of H_1 and H_2 is defined as the graph on vertex set $V_{1,2} = V_1 \times V_2$ (the Cartesian product of the vertex sets) with two types of arcs. Arcs of type 1 (type 2) are between pairs $(v_1, v_2) \in A_{1,2}$ and $(w_1, w_2) \in A_{1,2}$ with $(v_1, w_1) \in A_1$ and $v_2 = w_2$ (with $v_1 = w_1$ and $(v_2, w_2) \in A_2$), so arcs of type 1 and 2 correspond to arcs of H_1 and H_2, respectively. For $k \geqslant 3$, the Cartesian product $H_1 \times H_2 \times \ldots \times H_k$ is defined recursively as $((H_1 \times H_2) \times \ldots) \times H_k$.

Since we only consider labelled weighted directed acyclic graphs, paths, etc., for convenience we skip the adjective finite labelled weighted directed acyclic where possible in the sequel.

2. Periodic Real-time Processes as Labelled Weighted Directed Acyclic Graphs

The rationale behind modelling processes by graphs is, that a process is always in a certain state, where via performing an action another state is reached. Similarly, from a specific vertex in a graph another vertex can be reached by passing through the arc between them. A process can be defined as a labelled weighted directed acyclic graph, therefore a periodic real-time process also can be defined as a weakly connected labelled directed graph [6]. If the process specification contains cycles, due to the real-time constraints (timeliness, leading to an upper bound for the number of cycles), one can unfold the cycles leading to an acyclic path, which is a directed acyclic graph.

Hence, a set of parallel real-time periodic processes can be modelled as a graph H with components H_i. For our purpose of improving the performance of real-time periodic applications, we are going to show how execution time is reduced by combining components of graphs. A set of parallel processes and its combination into one process has to have identical behaviour (i.e. the *traces* and *failures*[7] of the set of parallel processes must be the same as those of the combined process).

Several products have been defined in the literature, like strong products, synchronised products, etc. None of these products are sufficient. The strong product as defined by [5] is a

[7]There are no *divergences* as the processes being combined are finite (repeated *periodically* by the real-time application).

labelled acyclic directed graph that is order preserving, but the arcs that produce a synchronised arc are not removed from the graph. In other words, behaviour is added to the original process set. Synchronised products have been defined by authors like Aiguier et al. [7], Caucal and Hassen [8], Hammal [9] and Wöhrle and Thomas [10]. The definition by [9] does not take into account that for a certain graph a certain label may occur more than once in a path, so the definition does not preserve the order. The definition of [8] is used to synchronise languages where the synchronised product of languages G and H is the disjunction of these languages and is also not order preserving.

For these reasons, these definitions do not meet our requirements, as the definition of the synchronised product has to be order preserving and our graphs have to reflect the behaviour of the processes on which the graphs are based. The definition by Aiguier et al. [7] stems from Input Output Symbolic Transition Systems and is almost similar to our product, although the terminology is different. Also we allow the source of the graph to be a set of vertices and Aiguier et al. require a single start state. Even the definition by Wöhrle and Thomas [10] does not fit our needs, although this product preserves the order as shown in Figure 1 (the dashed arc is the synchronising arc). In their approach it is possible for the synchronised product of two weakly connected graphs to contain again two weakly connected graphs (where the diamond-shaped component represents states and transitions unreachable according to the synchronisation rules).

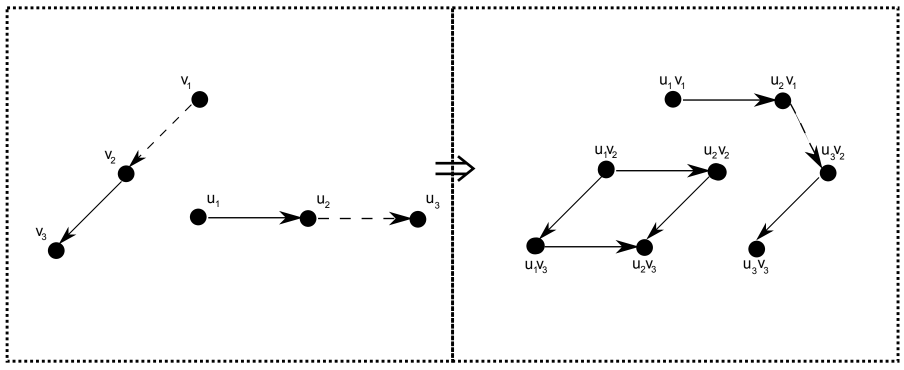

Figure 1. Synchronising product according to Wöhrle and Thomas.

At first sight, the Cartesian product seems to be a good way to express the combination of parallel processes. However, this product does not take care of necessary synchronisations. Therefore, we propose a modification of the product by [10] that will be developed in a number of steps. Figure 2 shows an example, where dashed arcs are synchronising actions. It shows five steps, where a synchronised product is built from a set of graphs. These five steps are elaborated in Sections 3 through 6.

The first diagram ($H1 + H2$) shows state-action transition graphs for two parallel processes, synchronising on one common action.

The second diagram ($H_1 \Box H_2$) shows the Cartesian product of those two graphs. The vertices are the cross-product of the original vertices and the transitions between them are in one of two dimensions (one for each of the original graphs). This corresponds to the CSP interleaving of the two processes (i.e. where each is free to engage in actions, regardless of whether they are held in common). Clearly, this is not a suitable serialisation of the original parallel system. This is presented in Section 3.

The third diagram ($H_1 \boxminus H_2$) is called the Weak Synchronised product of the (original) two graphs. It is derived from the Cartesian product by removing arcs representing common actions, if those arcs proceed from a vertex in only one of the dimensions (i.e. the action was engaged in by only one of the original processes). Common arcs remain always in the form

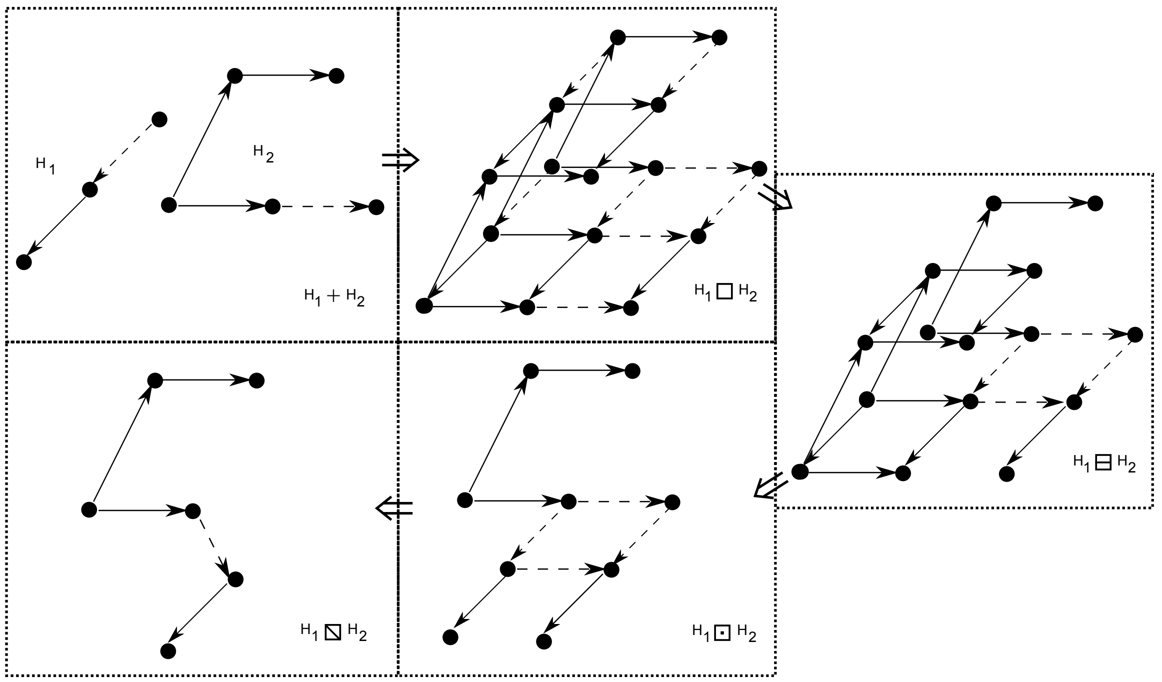

Figure 2. Transformations from parallel (+) through Cartesian (□), weak synchronised (⊟), reduced weak synchronised (⊡) to synchronised (⧅) product.

of two dimensional parallelograms (one dimension for each original process engaging in the action). If there is deadlock in the system, this will appear as vertices with no out-flowing arcs. This is in Section 4.

The fourth diagram ($H_1 \boxdot H_2$) is called the Reduced Weak Synchronised product. It is derived from the third by (iteratively) removing all vertices that have been left with no in-flowing arcs (other than those in the Cartesian product that had none – i.e. the starting points), together with the out-flowing arcs from those removed vertices. This is Section 5.

Finally, the fifth diagram ($H_1 ⧅ H_2$) is the Synchronised product. This collapses the common action parallelograms into single action arcs across the diagonal, leaving the two isolated vertices. The same iterative process from the fourth stage (for removing vertices with no in-flowing arcs and their out-flowing arcs) cleans up. This is Section 6.

3. The Cartesian Product of a Set of Parallel Processes

To visualise all our transformations we use a simplified version of an untimed example by [11] given in Listing 1, shown in Figure 3. The example contains three serial processes running in parallel, synchronising on their common actions respectively. Clearly they can be serialised simply by concatenating them , removing the middle SKIPs and merging the common actions to a single occurrence. The example is chosen to illustrate the stages of transformation and kept simple for this purpose. In the Appendix, Listing 2 and the related graph transformation in Figure 12 give a (slightly) more complex, and interesting example.

The process SEQUENCE_CONTROL is tail recursive, each element of the recursion being one period of the control logic. We use the constituent processes of this period for our transformations (starting in Figure 4). We assume that the actions have a given upper bound time value. We abbreviate the actions and their related upper bound time values in the several product figures: for example, (read_distance_sensors, 120 μs) will become rds. As before, a dashed arc represents a synchronising arc.

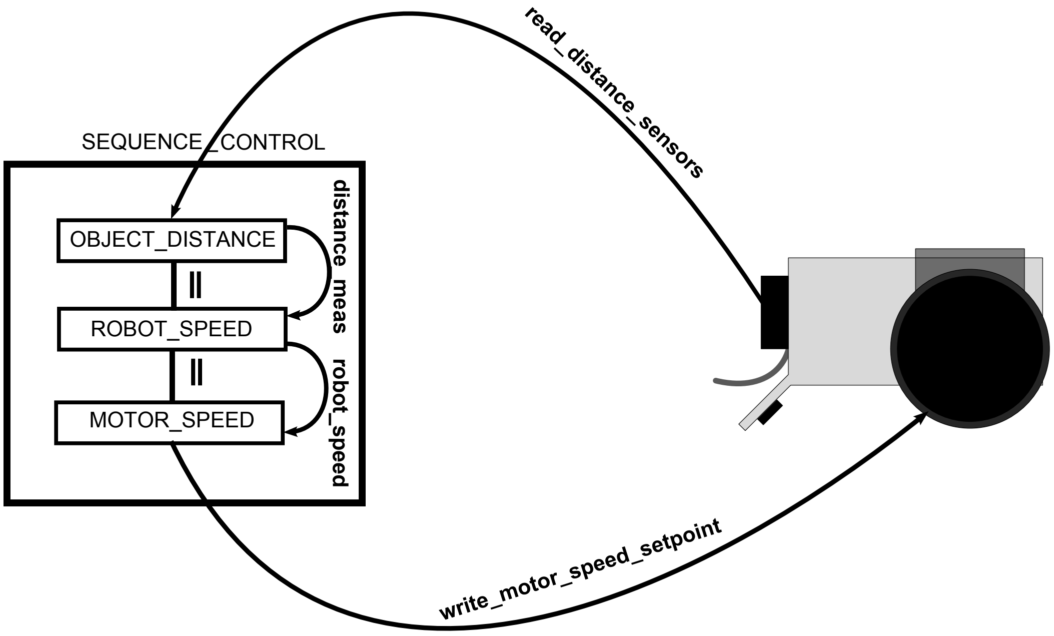

Figure 3. Untimed sequence control processes of a mobile robot.

```
OBJECT_DISTANCE  = read_distance_sensors →
                   compute_object_distance → distance_meas → SKIP

ROBOT_SPEED = distance_meas → compute_robot_speed → robot_speed → SKIP

MOTOR_SPEED      = robot_speed →
                   compute_motor_speed → write_motor_speed_setpoint → SKIP

SEQUENCE_CONTROL = (OBJECT_DISTANCE || ROBOT_SPEED || MOTOR_SPEED);
                   SEQUENCE_CONTROL;
```

Listing 1. Description of the SEQUENCE_CONTROL process.

The graph $SQ = MS + RS + OD$ representing the processes $MS =$ MOTOR_SPEED, $RS =$ ROBOT_SPEED and $OD =$ OBJECT_DISTANCE and using abbreviated actions is given by:

$$\begin{aligned} MS = & \ (V(H_1), A(H_1), \{\lambda(a) | a \in A(H_1)\}) \\ = & \ (\{v_1, v_2, v_3, v_4\}, \{v_1v_2, v_2v_3, v_3v_4\}, \{(v_1v_2, rs), (v_2v_3, cms), (v_3v_4, wmss)\}) \end{aligned}$$

$$\begin{aligned} RS = & \ (V(H_2), A(H_2), \{\lambda(a) | a \in A(H_2)\}) \\ = & \ (\{v_5, v_6, v_7, v_8\}, \{v_5v_6, v_6v_7, v_7v_8\}, \{(v_5v_6, dm), (v_6v_7, crs), (v_7v_8, rs)\}) \end{aligned}$$

$$\begin{aligned} OD = & \ (V(H_3), A(H_3), \{\lambda(a) | a \in A(H_3)\}) \\ = & \ (\{v_9, v_{10}, v_{11}, v_{12}\}, \{v_9v_{10}, v_{10}v_{11}, v_{11}v_{12}, \}, \{(v_9v_{10}, rds), (v_{10}v_{11}, cod), (v_{11}v_{12}, dm)\}) \end{aligned}$$

The Cartesian product of the graph SQ, $MS \square RS \square OD$, contains 64 states; therefore, we do not show the formal definition of the graph. From Figure 4, it may be checked that $\ell(MS + RS + OD)$, which is $\ell(MS) + \ell(RS) + \ell(OD)$, is equal to $\ell(MS \square RS \square OD)$. Next, we will show that this holds in the general case for finite labelled weighted directed acyclic multi-graphs.

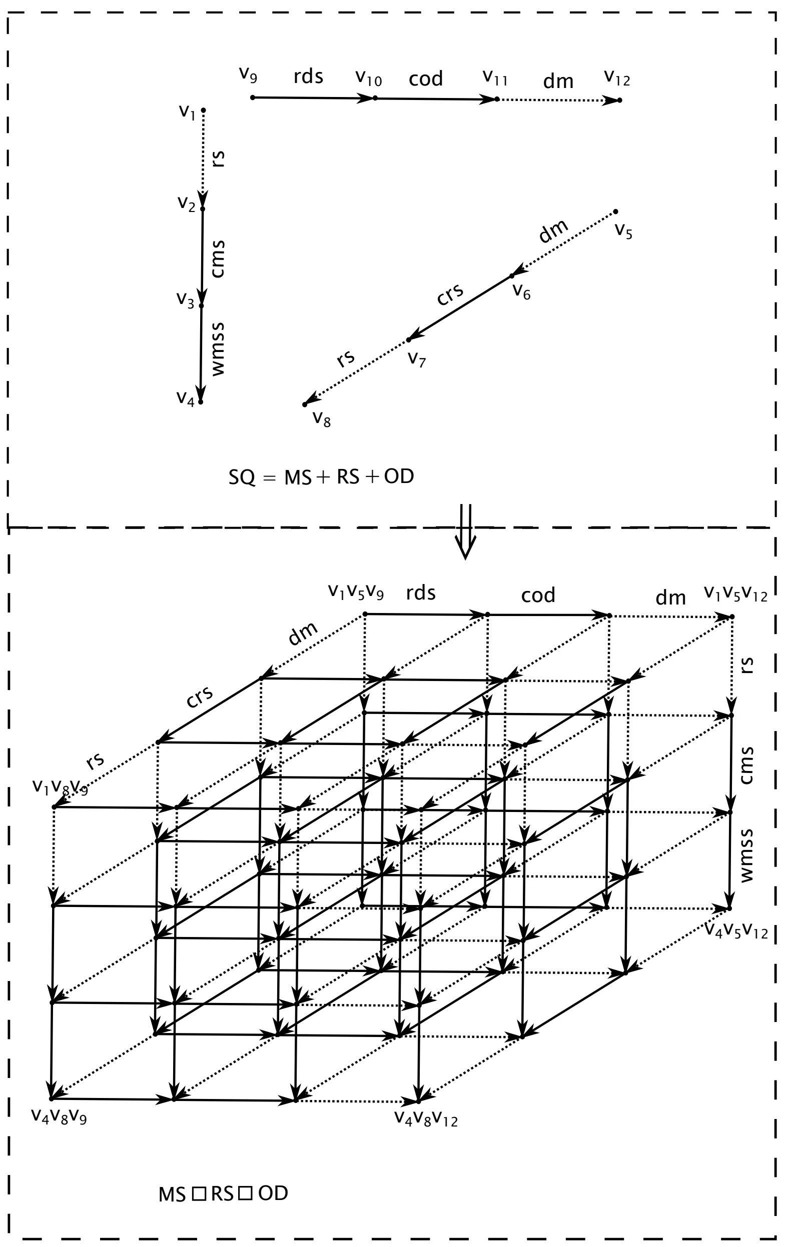

Figure 4. Sequence control processes of a mobile robot, from $+$ to $\square$.

In the Cartesian product $H_1 \square H_2$ of H_1 and H_2, we distinguish between two types of arcs. Arcs of type H_1 (type H_2) are between pairs $(v_1, w_1) \in V(H_1 \square H_2)$ and $(v_2, w_2) \in V(H_1 \square H_2)$ with $v_1v_2 \in A(H_1)$ and $w_1 = w_2$ (with $v_1 = v_2$ and $w_1w_2 \in A(H_2)$), so arcs of type H_1 and type H_2 correspond to (are in fact copies of) arcs of H_1 and H_2, respectively.

For $k \geqslant 3$, the Cartesian product $\square_{i=1}^{k} H_i = H_1 \square H_2 \square \ldots \square H_k$ is defined recursively as $((H_1 \square H_2) \square \ldots) \square H_k$. If no ambiguity can arise, we write $\square H_i$ for $\square_{i=1}^{k} H_i$. In this product of k directed graphs, we distinguish between arcs of type H_i for $i = 1, \ldots, k$, analogously as for the case $k = 2$. Note that in case the H_i are labelled, the labels of the arcs of type H_i in $\square H_i$ correspond to the labels of the arcs of H_i: each copy of arc $a \in A(H_i)$ in $\square H_i$ has label $\lambda(a)$. If H_i is a multi-graph, an arc $a \in A(H_i)$ can appear more than once in H_i, but in that case the copies of a in H_i have distinct labels, so each of the copies can be identified by its label. In $\square H_i$ similarly, we can distinguish the copies of a by their labels ($\lambda_1(a)$, $\lambda_2(a)$, ...).

For the sequel, we need a number of useful properties of acyclic directed graphs. Most of these properties are straightforward and easy to prove – see [12].

Let G be an acyclic directed (multi-)graph. Then G has at least one vertex v_1 with in-degree 0. If we delete v_1 and all the arcs with tail v_1 from G, we obtain a new acyclic directed (multi-)graph, so we can again find a vertex v_2 with in-degree 0, etc. We can repeat this procedure as long as there are vertices, and we obtain a so-called acyclic ordering $v_1, v_2, \ldots$ of the vertex set of G. It is important to observe that this ordering implies that arcs of G can only exist from v_i to v_j with $i < j$. We will use a slightly different (partial) ordering for our purposes, as follows.

We assume throughout that all our graphs H_i are acyclic directed multi-graphs. For the moment, we disregard the labels and weights, so in the following paragraphs the length of a directed path is just the number of arcs.

For each H_i we define S^i_0 to denote the set of vertices with in-degree 0 in H_i, S^i_1 the set of vertices with in-degree 0 in the graph obtained from H_i by deleting the vertices of S^i_0 and all arcs with tails in S^i_0, and so on, until the final set $S^i_{t_i}$ contains the remaining vertices with in-degree 0 and there are no arcs in the remaining graph. As in the acyclic ordering, this ordering implies that arcs of H_i can only exist from a vertex in $S^i_{j_1}$ to a vertex in $S^i_{j_2}$ if $j_1 < j_2$. This also implies that the vertices of $S^i_{t_i}$ have out-degree 0 in H_i, and that t_i is the length of a longest directed path in H_i, so $t_i = \ell(H_i)$. In fact, all longest directed paths of H_i have their starting vertex in S^i_0 and their terminating vertex in $S^i_{t_i}$. If a vertex $v \in V(H_i)$ is in the set S^i_j in the above ordering, we also say that v is at level j in H_i. Note that a vertex v of level $j > 0$ can only be reached from a vertex of level smaller than j, and that there always exists at least one vertex u of level $j-1$ with $uv \in A(H_i)$. Similarly, there exists a directed path of length p between some (not any) vertex at level j and some (not any) vertex at level $j+p$, but no longer directed paths (but possibly shorter directed paths). So, in particular, if there is a directed path of length p from a vertex u to a vertex v, and u is at level j, then v is at level at least $j+p$.

Apart from the inheritance of (copies of) the arcs and labels, the Cartesian product preserves some other important properties for our analysis. First of all, we show that the Cartesian product of a series of acyclic graphs $H_1, H_2, \ldots, H_k$ is again an acyclic graph, and that the length of a longest path in the Cartesian product is the sum of the lengths of longest paths in H_i, $i = 1, 2, \ldots, k$. In fact, we prove the stronger statement that each longest path P in $\square H_i$ corresponds to longest paths in all H_i, in the sense that P contains exactly one copy of each of the arcs of a longest path Q_i in H_i, $i = 1, 2, \ldots, k$. We say that P is the interleaved concatenation of these Q_i.

Lemma 1. *Let H_i be an acyclic graph for $i = 1, 2, \ldots, k$, where $k \geqslant 2$. Then $\square H_i$ is acyclic and every longest path in $\square H_i$ is the interleaved concatenation of longest paths Q_i in H_i, $i = 1, 2, \ldots, k$. In particular, $\ell(\square H_i) = \ell(H_1) + \ell(H_2) + \ldots + \ell(H_k)$.*

Proof. First note that it suffices to prove the statements for $k = 2$, since for integers $k \geqslant 3$, $H_1 \square H_2 \square \ldots \square H_k$ is $((H_1 \square H_2) \square \ldots) \square H_k$, hence $H'_1 \square H'_2$, and the result follows by induction. So we want to prove that $H_1 \square H_2$ is acyclic and that every longest path in $H_1 \square H_2$ is the interleaved concatenation of longest paths Q_1 and Q_2 in H_1 and H_2, respectively.

It is easy to show that there exists a path in $H_1 \square H_2$ that is the interleaved concatenation of two longest paths Q_1 and Q_2 in H_1 and H_2, respectively. In fact, if $P = p_1 p_2 \ldots p_{k_1}$ and $Q = q_1 q_2 \ldots q_{k_2}$ are two vertex-disjoint (longest) paths, then clearly $P \square Q$ contains the path $(p_1, q_1)(p_1, q_2) \ldots (p_1, q_{k_2})(p_2, q_{k_2}) \ldots (p_{k_1}, q_{k_2})$ with a length that is the sum of the lengths of P and Q.

The keys to the remaining part of the proof are the following observations on paths in $H_1 \square H_2$. Consider a (longest) path P in $H_1 \square H_2$ that starts with a subpath Q_1 of type H_1 arcs only, followed by a subpath R_1 with a first arc of type H_2 (and with R_1 possibly containing arcs of type H_1 as well). Then Q_1 corresponds directly to a path Q'_1 in H_1 (with the first coordinates of the vertex pairs corresponding to the vertices in Q_1 as the vertices of Q'_1; all the second coordinates are identical and equal to one particular vertex of $V(H_2)$), while the vertex pairs corresponding to the two vertices of the arc connecting the end of Q_1 to the beginning of R_1 have the same first coordinate $x \in V(H_1)$. The vertex pairs corresponding to the vertices of R_1 keep this first coordinate x as long as the arcs are of type H_2. In case these arcs are followed by an arc of type H_1, this arc of type H_1 corresponds to an arc in H_1 starting from x. So, all the subsequent subpaths of P with only arcs of type H_1 correspond directly to paths $Q'_1, Q'_2, \ldots$ in H_1, and similarly all the subsequent subpaths of P with only arcs of type H_2 correspond directly to paths $R'_1, R'_2, \ldots$ in H_2. Moreover, there is an arc in H_1 between the end vertex of Q'_1 and the first vertex of Q'_2 (if any), and so on, and similarly for R'_1 and R'_2, and so on (if any) in H_2. By symmetry, the same observations can be made if the path P starts with an arc of type H_2, and contains arcs of both types.

To prove that $H_1 \square H_2$ is acyclic, suppose that it is not and contains a cycle C. Then the first and last vertices of C are identical, say equal to (p_1, q_1). It is clear that C contains arcs of both types; otherwise C corresponds directly to a cycle in H_1 or H_2, contradicting our assumption that H_1 and H_2 are both acyclic. Assuming, without loss of generality, that the first $k \geqslant 1$ arcs of C are of type H_1, p_1 is the first vertex of a path $Q'_1 = p_1 p_2 \ldots p_{k+1}$ in H_1, with the corresponding subpath of C in $H_1 \square H_2$ consisting of vertex pairs (p_i, q_1), $i = 1, \ldots, k+1$. Then the first arc of type H_2 in C we encounter is from (p_{k+1}, q_1) to (p_{k+1}, q_2) for some $q_1 q_2 \in A(H_2)$, and so on, so q_1 is the first vertex of a path R'_1 in H_2, as in the above argumentation. Since (p_1, q_1) also appears as the last vertex of C, by similar arguments p_1 and q_1 both appear as the last vertex of two paths Q'_t and R'_s in H_1 and H_2, respectively. Since by the above argumentation all the subpaths of type Q'_i are connected, this implies that H_1 contains a cycle: a *contradiction*. This proves that $H_1 \square H_2$ is acyclic.

Suppose now that P is a longest path in $H_1 \square H_2$. Assume that P has length $\ell(H_1 \square H_2) > \ell(H_1) + \ell(H_2)$. Using the above argumentation and the fact that $H_1 \square H_2$ is acyclic, the two paths Q and R formed by the Q'_i in H_1 and the R'_i in H_2, respectively, together have length $\ell(H_1 \square H_2) > \ell(H_1) + \ell(H_2)$, but this contradicts that the length of Q is at most $\ell(H_1)$ and the length of R is at most $\ell(H_2)$. Together with the above arguments, this shows that P has length exactly $\ell(H_1) + \ell(H_2)$ and that P is the interleaved concatenation of the two longest paths Q and R in H_1 and H_2, respectively. This completes the proof of Lemma 1. $\square$

Remark 3.1

The expression in Lemma 1 on the length of longest paths in the Cartesian product is not valid if we drop the condition that each of the H_i is acyclic. It is easy to present counterexamples. For instance, consider $H_1 \square H_2$, where H_1 consists of two arcs connecting 3 vertices and H_2 has two arcs between two vertices, but in opposite directions (i.e. a cycle). As can be observed in Figure 5, the longest directed path lengths of H_1 and H_2 are 2 and 1, respectively[8], making their sum 3. However, $H_1 \square H_2$ contains a directed path of length 5.

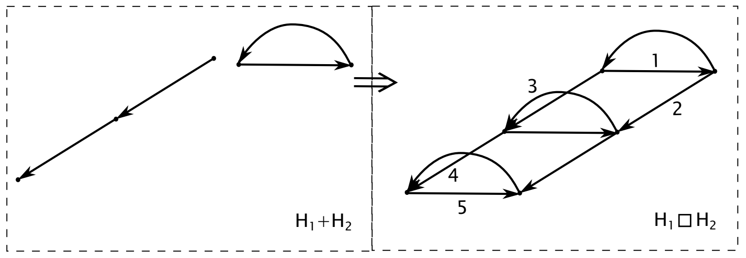

Figure 5. Cyclic graph counterexample.

[8]The vertices in a directed path must be distinct – see Section 1.

Remark 3.2

The notion of the level of a vertex in an acyclic directed graph, that we introduced before, has a natural extension to the Cartesian product, in the following sense. For a vertex $(v_1, v_2, \ldots, v_k) \in V(\square H_i)$ we define the level vector $(f_1, f_2, \ldots, f_k)$, in which f_i denotes the level of vertex v_i in H_i. Then the vertices with in-degree 0 in $\square H_i$ are precisely all vertices with level vector $(0,0,\ldots 0)$, whereas level vector $(t_1, t_2, \ldots, t_k)$ with $t_i = \ell(H_i)$ corresponds to all vertices that are terminals of some longest path in $\square H_i$. For each integer vector $(x_1, x_2, \ldots, x_k)$ with $0 \leqslant x_i \leqslant f_i$, there exists a vertex in $\square H_i$ with this level vector, and if $x_i < f_i$, there also exists an arc of type H_i between a vertex with level vector $(x_1, x_2, \ldots, x_i, \ldots, x_k)$ and $(x_1, x_2, \ldots, x_i + 1, \ldots, x_k)$. This implies there are several longest paths in $\square H_i$, each represented by adding one of the total of $t_1 + \ldots t_k$ units to one of the coordinates in the level vector between subsequent vertices on the path. On the other hand, there cannot be any arcs in $\square H_i$ between a vertex with level vector $(x_1, x_2, \ldots, x_i, \ldots, x_k)$ and a vertex with level vector $(y_1, y_2, \ldots, x_i - 1, \ldots, y_k)$; all arcs imply an increase (by 1 or more) in precisely one entry of the level vector, while the other entries remain the same.

This shows how the partial ordering on the vertices of an acyclic directed graph has a natural extension to the Cartesian product. The same holds for the partial ordering on the arcs. Since the Cartesian product is again acyclic, we can define the same ordering there. So we define that for $a, b \in A(\square H_i)$, $a < b$ if and only if a precedes b on some directed path in $\square H_i$. From the structure it then follows that the ordering of the arcs in the individual H_i is preserved in the Cartesian product, in the following sense. If $a < b$ for two arcs $a, b \in A(\square H_i)$ of the same type H_i, then for the corresponding arcs a' of a in H_i and b' of b in H_i, it holds that $a' < b'$ in H_i. The simplest way to see this is by the level vectors: if $b' < a'$, then the level of the head of b' in H_i is smaller than the level of the tail of a' in H_i, but then the corresponding coordinate in the level vector of the head (and thus of the tail) of b is also smaller than that of the tail (and thus of the head) of a in $\square H_i$, contradicting that there is a directed path in $\square H_i$ in which a precedes b.

Remark 3.3

In the above proof, we did not specifically consider the possibility of having multiple arcs, and we did not use the labels and weights in our arguments, for convenience. It is obvious that the proof is the same if we allow multiple arcs, because any directed path can contain at most one of these arcs, and we have already indicated how we can identify the corresponding arc in H_i from the inherited labels. It is also rather straightforward how the proof should be adapted if we consider weighted arcs and the length of a path is the sum of the weights of its arcs. The crucial observation is that every arc with a specific weight in $H_1 \square H_2$ corresponds to either an arc in H_1 or an arc in H_2 with exactly the same weight, so instead of a contribution of 1 (which can be interpreted as weight 1) of an arc to the length of a path, we then have to use this specific weight. The weights have no influence on the level vectors, since the levels of the vertices are determined by the (non)existence of arcs, not by their weights. Of course, longest paths in terms of the highest total weight do not necessarily coincide with longest paths in terms of the largest number of arcs, so longest paths in the weighted sense may jump more than one unit in one of the coordinates of the level vector. Note, however, that these paths still start in a vertex with level vector $(0, 0, \ldots, 0)$ and terminate in a vertex with level vector $(t_1, t_2, \ldots, t_k)$ (where the t_i refer to the unweighted case of Remark 3.2 above).

Remark 3.4

An (acyclic) directed graph can have an exponentially high number of longest paths in terms of its number of vertices n. Consider for instance such a graph G with a square number of vertices, with $\sqrt{n}$ vertices of level i for all $i = 1,2,\ldots,\sqrt{n}$, and arcs between any two vertices from a lower level to a higher level, all with weight 1. Then the number of longest paths in G is $\sqrt{n}^{\sqrt{n}}$, so clearly exponential in n.

4. The Weak Synchronised Product of a Set of Parallel Processes

The Cartesian product of graphs is an adequate model for the interleaved execution of processes as long as the graphs represent independent processes. The model fails if the processes are not independent, for instance in case the processes must synchronise over certain actions. The different paths in the Cartesian product represent all possible (interleaved) traces of the constituent processes, thereby also representing behaviour that is simply impossible, due to synchronisation. For this reason we need a more restrictive notion than the Cartesian product of graphs. As for the Cartesian product, this has to be order-preserving. This product we are going to introduce next is based on the synchronised product by [10]. Figure 6 gives for our example the transformation of SQ consisting of the Cartesian product of the graphs $MS\Box RS\Box OD$ to the weak synchronised product $MS\boxminus RS\boxminus OD$.

The weak synchronised product $H_1\boxminus H_2$ of H_1 and H_2 is defined as the graph on vertex set $V(H_1)\times V(H_2)$ (the Cartesian product of the vertex sets) and arc set $A_{1,2}$ with four types of arcs.

The first two types correspond to arcs in H_1 and H_2 that have labels that only appear in one of H_1 and H_2. We call this set of arcs the *asynchronous arc set* and denote it by $A^a_{1,2}$. Therefore, $A^a_{1,2}$ is the set of all pairs $(v_1,x)(v_2,x)$ with $x\in V(H_2)$ and the associated label $\lambda(v_1v_2)$ (*a-type* H_1 arcs) or $(y,w_1)(y,w_2)$ with $y\in V(H_1)$ and the associated label $\lambda(w_1w_2)$ (*a-type* H_2 arcs), where for arcs $v_1v_2\in A(H_1)$ label $\lambda(v_1v_2)$ does not appear in H_2 and for arcs $w_1w_2\in A(H_2)$ label $\lambda(w_1w_2)$ does not appear in H_1.

The other types correspond to arcs in H_1 and H_2 with the same label. We call this set of arcs the *synchronous arc set* and denote it by $A^s_{1,2}$. Therefore, $A^s_{1,2}$ is the set of all arcs $(v_1,w_1)(v_2,w_1)$, $(v_1,w_1)(v_1,w_2)$, $(v_1,w_2)(v_2,w_2)$, $(v_2,w_1)(v_2,w_2)$, with the associated label $\lambda(v_1v_2)$, where for arcs $v_1v_2\in A(H_1)$ and $w_1w_2\in A(H_2)$ label $\lambda(v_1v_2)=\lambda(w_1w_2)$. The first and third are *s-type* H_1-arcs and the others are *s-type* H_2-arcs.

For $k\geqslant 3$, the weak synchronised product $H_1\boxminus H_2\boxminus\ldots\boxminus H_k$ is defined recursively as $((H_1\boxminus H_2)\boxminus\ldots)\boxminus H_k$. If no confusion can arise, we denote it as $\boxminus H_i$.

Although this weak synchronised product, like the Cartesian product, might represent behaviour that is not allowed by the original process specification, we will use it as an intermediate result. For example, in Figures 2 and 6, it is possible in both the Cartesian product and the weak synchronised product to reach a vertex that represents a non-reachable state in the process specification, by using only one of the synchronous arcs of a parallelogram.

We will first show that longest paths cannot be longer than in the Cartesian product, and that this new product also preserves acyclicity and the order on the arcs.

Lemma 2. *Let H_i be an acyclic graph for $i = 1,2,\ldots,k$, where $k\geqslant 2$. Then $\boxminus H_i$ is acyclic and $\ell(\boxminus H_i)\leqslant\ell(\Box H_i)$.*

Proof. As in the proof of Lemma 1, it suffices to prove the statements for $k = 2$, since for integers $k\geqslant 3$, the weak synchronised product $H_1\boxminus H_2\boxminus\ldots\boxminus H_k$ is defined recursively as $((H_1\boxminus H_2)\boxminus\ldots)\boxminus H_k$, hence $H_1'\boxminus H_2'$, and the result follows by induction.

It is obvious from the definitions that $H_1\boxminus H_2$ is a spanning subgraph of $H_1\Box H_2$ (i.e. that the vertex set of $H_1\Box H_2$ equals the vertex set of $H_1\boxminus H_2$, and that the arc set of $H_1\boxminus H_2$ is a subset of the arc set of $H_1\Box H_2$). From this observation, it follows by Lemma 1 that $H_1\boxminus H_2$ is acyclic and that $\ell(H_1\boxminus H_2)\leqslant\ell(H_1\Box H_2)$. □

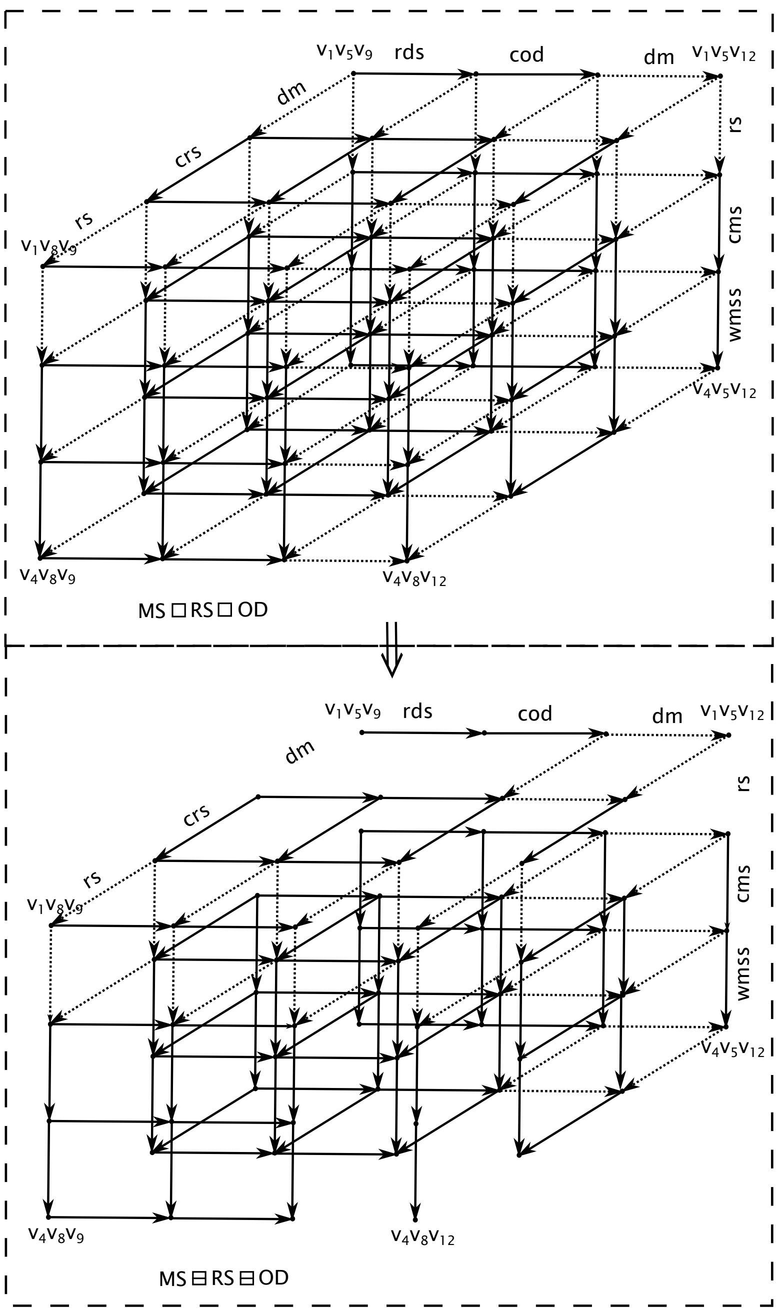

Figure 6. Sequence control processes of a mobile robot, from $\square$ to $\boxminus$.

Remark 4.1

It is not difficult to give examples of labelled directed acyclic graphs H_1 and H_2 with $\ell(H_1 \boxminus H_2) < \ell(H_1 \square H_2)$. For example, Figure 7 shows H_1 consisting of one directed path $v_1v_2v_3$ with labels $\lambda(v_1v_2) = a$ and $\lambda(v_2v_3) = b$, and H_2 also consisting of one directed path $w_1w_2w_3$ with $\lambda(w_1w_2) = b$ and $\lambda(w_2w_3) = a$. In the context of processes, this example is ill-defined in the sense that it is immediately clear that the two processes are deadlocked from the start. The graph representing this pathological example is inconsistent. In Figure 8, we give a three dimensional example where we show that such a pathological case can occur distributed over several graphs, although only the final step reduces the out-degree of the source to zero. The source vertex in Figure 8 is marked by a circle around the vertex. From three dimensions it is not difficult to extend to n-dimensions. A set of graphs H is said to be consistent if it does not contain a (possibly n-dimensional) pathological case.

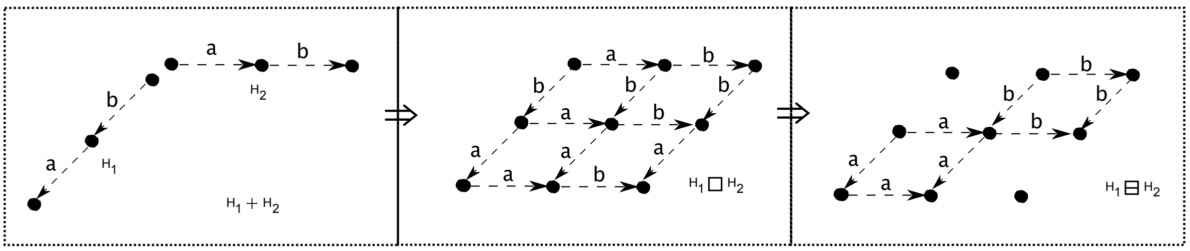

Figure 7. Two-dimensional pathological case.

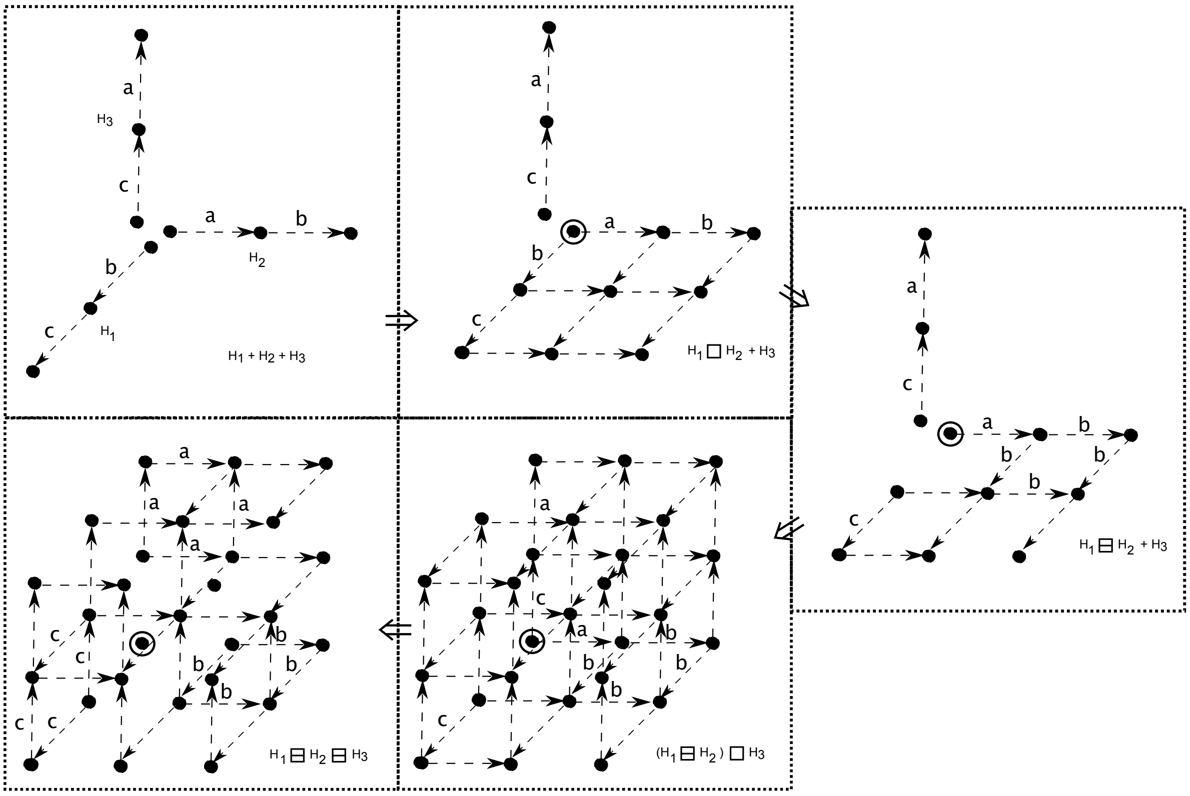
a
b
c
H_1
H_2
H_3
$H_1 + H_2 + H_3$
$H_1 \square H_2 + H_3$
$H_1 \boxminus H_2 + H_3$
$(H_1 \boxminus H_2) \square H_3$
$H_1 \boxminus H_2 \boxminus H_3$

Figure 8. Three-dimensional pathological case.

We are now going to show that inconsistency can always be concluded if $\ell(\boxminus H_i) < \ell(\square H_i)$. By induction, we can again restrict our attention to the case that $k = 2$.

Let P be a longest path in $H_1 \square H_2$. Then P is the interleaved concatenation of two longest paths R and Q in H_1 and H_2, respectively. Let $R = r_1 r_2 \dots r_{k_1}$ and $Q = q_1 q_2 \dots q_{k_2}$. If P is not a longest path in $H_1 \boxminus H_2$, there is at least one label λ that appears on arcs in both R and Q. Consider the first label on R (starting from r_1) that also appears in Q – say this is label λ_1 that appears on r_j. If λ_1 is also the first label on Q that appears in both Q and R, say on q_t, then $R \boxminus Q$ contains a path of length $j+t$ corresponding to the subpath of P of length $j+t$ from the starting vertex.

Continuing this way, if all labels that appear in both Q and R also appear in the same order in Q and R (with possible repetitions, also in the same order), then $H_1 \boxminus H_2$ contains a path with the same length as P. So, if $\ell(H_1 \boxminus H_2) < \ell(H_1 \square H_2)$, we may assume there is an i^{th} instance of a label λ_r on R that is also the i^{th} instance of that label on Q, and a j^{th} instance of a label λ_q on Q that is also the j^{th} instance of that label on R, and such that λ_r is after λ_q on R but before λ_q on Q. But then the process is ill-defined in a similar way as in the pathological example, a situation that can and should be avoided.

For the sequel, we are going to assume that the processes are defined and specified in such a way, that the above unwanted situation does not occur and that the related graphs H_i are therefore consistent. This then automatically implies that for consistent graphs H_i, $\ell(\boxminus H_i) = \ell(\Box H_i)$.

Remark 4.2

From the fact that $\boxminus H_i$ is a spanning subgraph of $\Box H_i$, it follows that the ordering of the arcs in the individual H_i is preserved in the weak synchronised product, in the following sense. If $a < b$ for two arcs $a, b \in A(\boxminus H_i)$ of a-type or s-type for the same H_i, then for the corresponding arcs a' of a in H_i and b' of b in H_i, it holds that $a' < b'$ in H_i.

This can be seen by using the level vectors. Suppose $b' < a'$. Then, the level of the head of b' in H_i is smaller than the level of the tail of a' in H_i. This means that the corresponding coordinate in the level vector of the head (and thus of the tail) of b is also smaller than that of the tail (and thus of the head) of a in $\boxminus H_i$, contradicting that there is a directed path in $\boxminus H_i$ in which a precedes b. Therefore, the supposition is false.

Remark 4.3

The weak synchronised product, like the Cartesian product, may still represent behaviour that is not possible by the specification of the corresponding set of processes, as can be seen in the examples in Figures 2 and 6 ($\Box \Rightarrow \boxminus$). One obvious thing we can do about this is, that we iteratively remove vertices (and the related arcs) that have an in-degree $\neq 0$ in the Cartesian product, but have an in-degree $= 0$ in the weak synchronised product.

5. The Reduced Weak Synchronised Product of a Set of Parallel Processes

The reduced weak synchronised product $H_1 \boxdot H_2$ of H_1 and H_2 is defined as the graph obtained from the synchronised product $H_1 \boxminus H_2$ by first removing all vertices with level 0 in $H_1 \boxminus H_2$ that have level > 0 in $H_1 \Box H_2$, 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 with level 0 in the current graph that have level > 0 in $H_1 \Box H_2$. The resulting graph for our standard example is shown in Figure 9.

For $k \geqslant 3$, the reduced weak synchronised product $H_1 \boxdot H_2 \boxdot \ldots \boxdot H_k$ is defined recursively as $((H_1 \boxdot H_2) \boxdot \ldots) \boxdot H_k$, and denoted as $\boxdot H_i$ if no confusion can arise.

Lemma 3. *Let H_i be an acyclic graph and let $\boxdot H_i$ be the reduced weak synchronised product of H_i for $i = 1, 2, \ldots, k$, where $k \geqslant 2$. Then $\ell(\boxdot H_i) = \ell(\boxminus H_i)$.*

Proof. One direction is clear: since $\boxdot H_i$ is a subgraph of $\boxminus H_i$, we have that $\ell(\boxdot H_i) \leqslant \ell(\boxminus H_i)$. For the other direction, consider a longest path P in $\boxminus H_i$. By previous arguments, we know that P starts in a vertex v with level vector $(0, 0, \ldots, 0)$ and terminates in a vertex w with level vector $(t_1, t_2, \ldots, t_k)$. For each vertex $x \neq v, w$ on P, $d^-(x) \geqslant 1$ and $d^+(x) \geqslant 1$. This implies that none of the vertices of P is removed from $\boxminus H_i$, so P is a path and therefore a longest path in $\boxdot H_i$. This completes the proof of Lemma 3. $\Box$

Remark 5.1

Note that the paths that represent the behaviour of the specified processes, all start in the source of the graph and end in the sink of the graph. Because we only remove vertices that are not in the source of the graph and have an in-degree of zero, behaviour not specified by the original set of processes is removed.

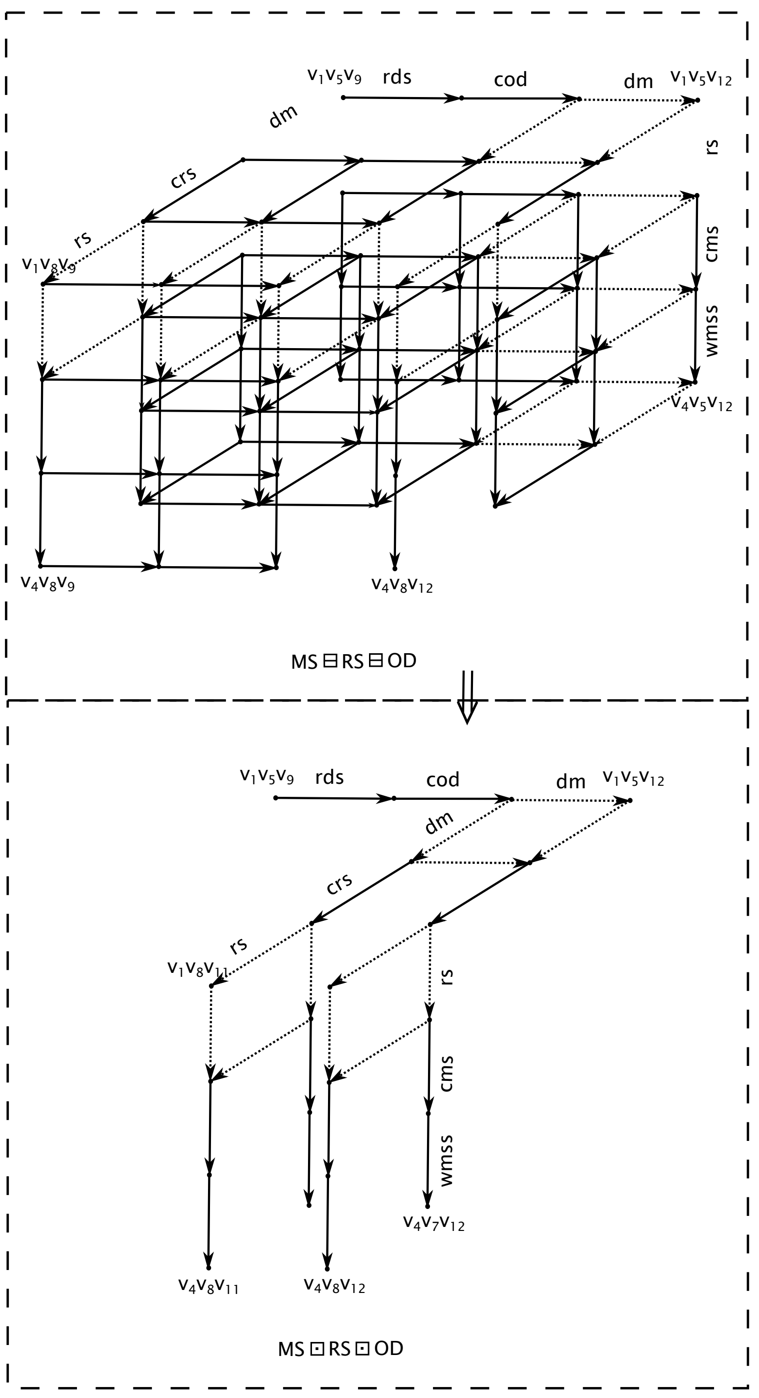

Figure 9. Sequence control processes of a mobile robot, from ⊟ to ⊡.

Remark 5.2

Also note that, although this newly introduced product may filter out vertices and arcs representing unwanted process behaviour, it does not filter out all unwanted behaviour, see Figure 9. In Section 6, we translate additional restrictions into our product.

6. The Synchronised Product of a Set of Parallel Processes

The synchronised product $H_1 \boxbslash H_2$ of H_1 and H_2 is defined from the reduced weak synchronised product, by first replacing quadruples of arcs that represent synchronised arcs as follows.

Replace each parallelogram of arcs $(v_1,w_1)(v_2,w_1)$, $(v_1,w_1)(v_1,w_2)$, $(v_1,w_2)(v_2,w_2)$ and $(v_2,w_1)(v_2,w_2)$, with $\lambda((v_1,w_1)(v_2,w_1)) = \lambda((v_1,w_1)(v_1,w_2)) = \lambda((v_1,w_2)(v_2,w_2)) = \lambda((v_2,w_1)(v_2,w_2))$, by one diagonal arc $(v_1,w_1)(v_2,w_2)$ with label $\lambda((v_1,w_1)(v_2,w_2)) = \lambda((v_1,w_1)(v_2,w_1))$. These new arcs of $H_1 \boxbslash H_2$ are called synchronous arcs, and the set of these arcs is denoted as $A^s_{1,2}$. This intermediate stage is shown in Figure 10.

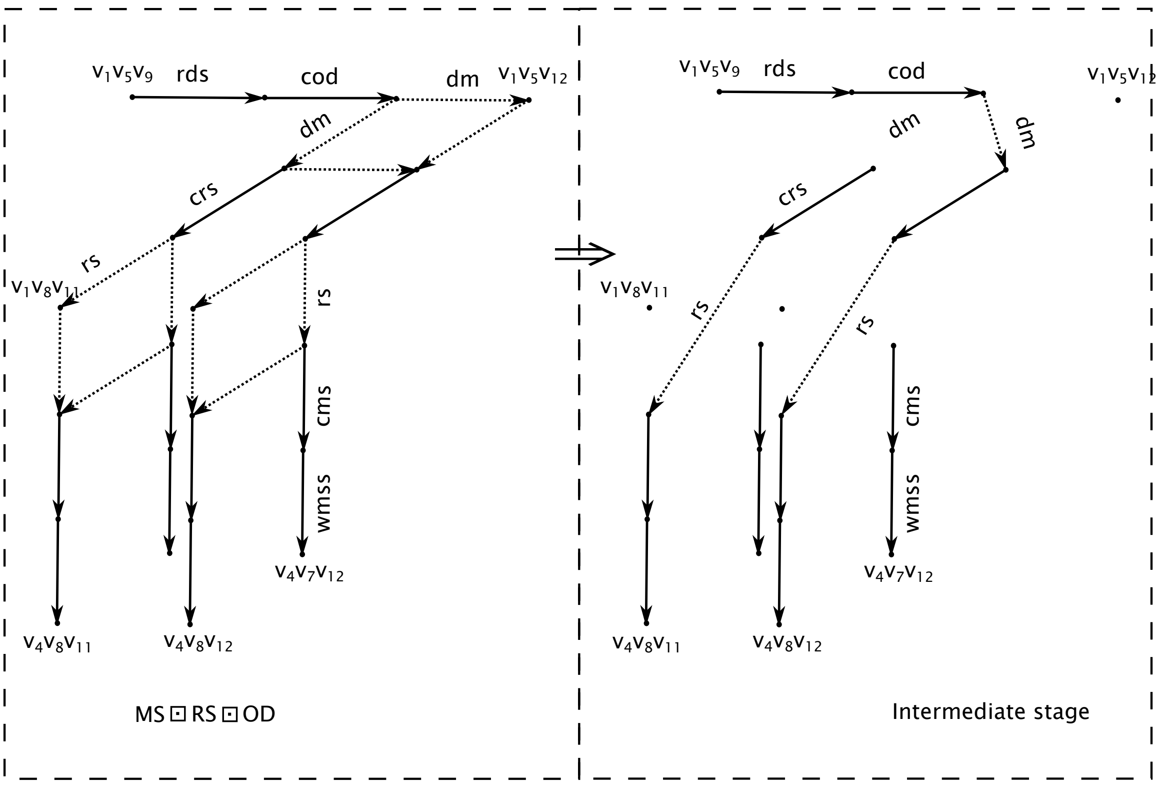

Figure 10. Sequence control processes of a mobile robot, from $\boxdot$ to intermediate stage.

Secondly, all vertices with level 0 in the resulting graph that have level > 0 in $H_1\square H_2$ 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 with level 0 in the current graph that have level > 0 in $H_1\square H_2$. The resulting graph is called the synchronised product and denoted as $H_1 \boxbslash H_2$. The set of arcs consisting of the other remaining (asynchronous) arcs of $H_1 \boxbslash H_2$ is denoted as $A^a_{1,2}$. The resulting graph for our standard example is shown in Figure 11.

For $k \geqslant 3$, the synchronised product $H_1 \boxbslash H_2 \boxbslash \ldots \boxbslash H_k$ is defined recursively as $((H_1 \boxbslash H_2) \boxbslash \ldots) \boxbslash H_k$, and denoted as $\boxbslash H_i$ if no confusion can arise.

Lemma 4. *Let H_i be an acyclic graph and let $\boxbslash H_i$ be the synchronised product of H_i for $i = 1,2,\ldots,k$, where $k \geqslant 2$. Then $\ell(\boxbslash H_i) \leqslant \ell(\boxdot H_i)$.*

Proof. As in the proof of Lemma 1, it suffices to prove the statement for $k = 2$, since for integers $k \geqslant 3$, the synchronised product $H_1 \boxbslash H_2 \boxbslash \ldots \boxbslash H_k$ is defined recursively as $((H_1 \boxbslash H_2) \boxbslash \ldots) \boxbslash H_k$, hence $H'_1 \boxbslash H'_2$, and the result follows by induction.

From the definitions of reduced weak synchronised product and synchronised product, it follows that the vertex set of $H_1 \boxbslash H_2$ is a subset of the vertex set of $H_1 \boxdot H_2$, and the asynchronous arc set $A^a_{1,2}$ of $H_1 \boxbslash H_2$ is a subset of the asynchronous arc set of $H_1 \boxdot H_2$. For the synchronous arc set $A^s_{1,2}$ of $H_1 \boxbslash H_2$ every arc replaces a quadruple of arcs in $H_1 \boxdot H_2$, as follows: $\{(v_1,w_1)(v_2,w_1), (v_1,w_1)(v_1,w_2), (v_1,w_2)(v_2,w_2), (v_2,w_1)(v_2,w_2)\}$ with an associated label $\lambda((v_1,w_1)(v_2,w_1)) = \lambda((v_1,w_1)(v_1,w_2)) = \lambda((v_1,w_2)(v_2,w_2)) = \lambda((v_2,w_1)(v_2,w_2))$

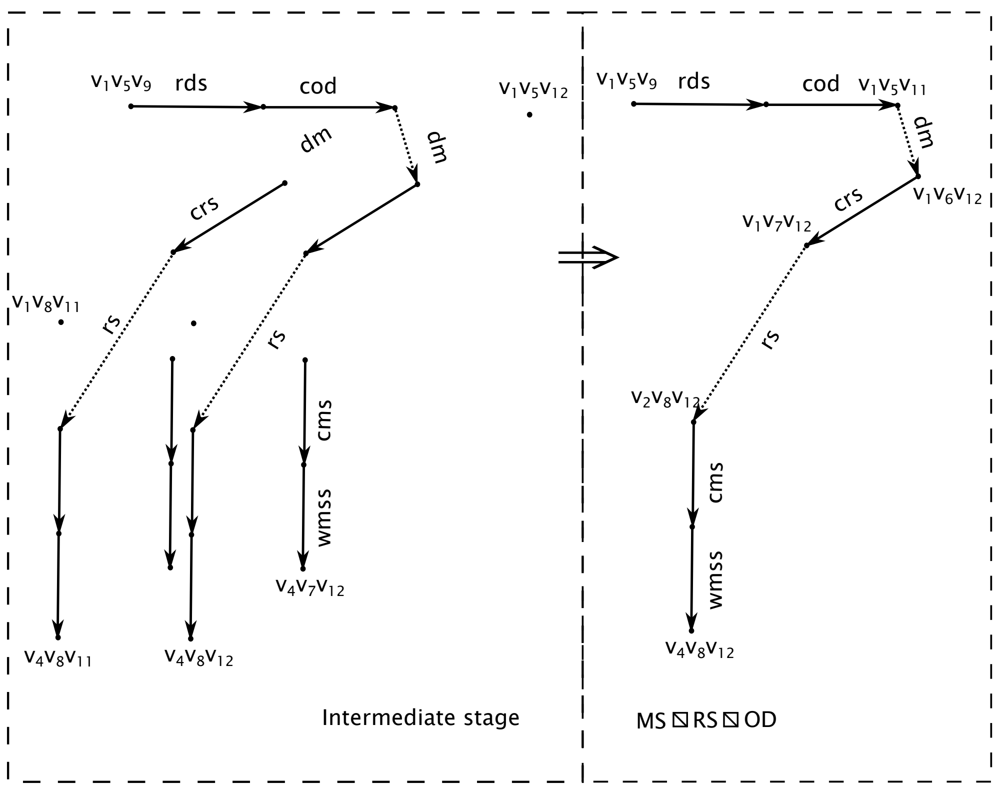

Figure 11. Sequence control processes of a mobile robot, from intermediate stage to $\boxbslash$.

in $H_1 \boxdot H_2$ is replaced by $(v_1,w_1)(v_2,w_2)$ with the associated label $\lambda((v_1,w_1)(v_2,w_2))$ = $\lambda((v_1,w_1)(v_2,w_1))$. Clearly, the length of (a longest path in) the graph with vertex set $\{(v_1,w_1), (v_1,w_2), (v_2,w_1), (v_2,w_2)\}$ and arc set $\{(v_1,w_1)(v_2,w_1), (v_1,w_1)(v_1,w_2), (v_1,w_2)(v_2,w_2), (v_2,w_1)(v_2,w_2)\}$ is twice the length of the arc $(v_1,w_1)(v_2,w_2)$. This shows that the length of a longest path in the synchronised product is not greater than the length of a longest path in the reduced synchronised product (but it will be smaller if synchronisation occurs between the constituent paths). From these observations it follows that, because $H_1 \boxdot H_2$ is acyclic, $H_1 \boxbslash H_2$ is acyclic and $\ell(H_1 \boxbslash H_2) \leqslant \ell(H_1 \boxdot H_2)$. □

Remark 6.1

The proof of Lemma 4 shows that combining processes *may* lead to a performance gain, where the gain $\mathcal{G}$ is defined by $\mathcal{G} = \sum_{i=1}^{k} \ell(H_i) - \ell(\boxbslash_{i=1}^{k} H_i)$. It is clear from the above that a gain is only guaranteed if $\ell(\boxbslash H_i) < \ell(\boxdot H_i)$. Logically, this means that we can only be sure of a gain if there exist distinct indices i and j such that for every longest path P in H_i and for every longest path Q in H_j, the paths P and Q contain at least one synchronising arc, so there are arcs $a \in A(P)$ and $b \in A(Q)$ with $\lambda(a) = \lambda(b)$. To get a performance gain we need necessary and sufficient conditions that will reduce the length of the synchronised product with respect to the length of its constituent graphs. It is obvious (follows from Lemma 4) that a reduction can only be achieved by synchronising arcs. As the length of a graph is defined as the size of its longest paths, we only have to consider the synchronisation of synchronising arcs in longest paths.

Lemma 5. *Let H_i be an acyclic graph for $i = 1,2,\ldots,k$, where $k \geqslant 2$. Then $\ell(\boxbslash H_i) = \ell(H_1) + \ell(H_2) + \ldots + \ell(H_k)$ if and only if every H_i has at least one longest path without synchronising arcs.*

Proof. First note that it suffices to prove the statement for $k = 2$, since for integers $k \geqslant 3$, $H_1 \boxbslash H_2 \boxbslash \ldots \boxbslash H_k$ is $((H_1 \boxbslash H_2) \boxbslash \ldots) \boxbslash H_k$, hence $H_1' \boxbslash H_2'$, and the result follows by induction.

$\Leftarrow$ By Lemma 1, $\ell(H_1 \Box H_2) = \ell(H_1) + \ell(H_2)$. If $P = p_1 p_2 \ldots p_{k_1}$ and $Q = q_1 q_2 \ldots q_{k_2}$ are two vertex-disjoint longest paths without synchronising arcs of H_1, H_2 respectively, then clearly $P \Box Q$ contains the path PQ, where PQ denotes the path $PQ = (p_1, q_1)(p_1, q_2) \ldots (p_1, q_{k_2})(p_2, q_{k_2}) \ldots (p_{k_1}, q_{k_2})$. By the definition of $H_1 \boxminus H_2$, it follows that $H_1 \boxminus H_2$ contains the path PQ, even so by definition $H_1 \boxdot H_2$ and $H_1 \boxbslash H_2$ contain the path PQ. As $\ell(PQ) = \ell(H_1) + \ell(H_2)$ it follows that $\ell(H_1) + \ell(H_2) = \ell(H_1 \boxbslash H_2)$.

$\Rightarrow$ The proof is by contra-position. Suppose that all longest paths $P = p_1 p_2 \ldots p_{k_1}, Q = q_1 q_2 \ldots q_{k_2}$ of H_1 and H_2, without loss of generality, contain one synchronising arc a with label $\lambda(a)$, say from p_i to p_{i+1} and q_j to q_{j+1}. The synchronised product of paths P and Q is $P' \Box Q' \bigcup (p_i p_{i+1} \boxbslash q_j q_{j+1}) \bigcup P'' \Box Q''$, with $P' = p_1 p_2 \ldots p_i$, $P'' = p_{i+1} \ldots p_{k_1}$, $Q' = q_1 q_2 \ldots q_j$, $Q'' = q_{j+1} \ldots q_{k_2}$. Therefore it follows that $\ell(P \boxbslash Q) = \ell(P' \Box Q') + \ell((p_i p_{i+1} \boxbslash q_j q_{j+1})) + \ell(P'' \Box Q'')$.

Note that $p_i p_{i+1}$ and $q_j q_{j+1}$ have the same label and therefore the same weight t. Therefore $\ell(p_i p_{i+1} \Box q_j q_{j+1}) = 2 \times t = 2 \times \ell(p_i p_{i+1} \boxbslash q_j q_{j+1})$ (due to the synchronisation constraint) and it follows that $\ell(P \boxbslash Q) = \ell(P' \Box Q') + \ell((p_i p_{i+1} \boxbslash q_j q_{j+1})) + \ell(P'' \Box Q'') = \ell(P' \Box Q') + t + \ell(P'' \Box Q'') < \ell(P' \Box Q') + 2 \times t + \ell(P'' \Box Q'') = \ell(P' \Box Q') + \ell((p_i p_{i+1} \Box q_j q_{j+1})) + \ell(P'' \Box Q'')$, $\ell(P \Box Q)$ so the synchronised product will reduce the length of the longest paths in H_1 and H_2. This leads to $\ell(H_1 \Box H_2) > \ell(H_1 \boxbslash H_2)$. □

We need necessary and sufficient conditions to get to $\ell(\boxbslash H_i) < \ell(\boxdot H_i)$.

Lemma 6. *Let H_i be an acyclic graph for $i = 1, 2, \ldots, k$, where $k \geqslant 2$. Then $\ell(\boxbslash H_i) < \ell(\boxdot H_i)$ if there exists H_n, H_m, $n \neq m, 1 \leqslant n, m \leqslant k$, such that each longest path in H_n, H_m, contains at least one same labelled synchronising arc.*

Proof. Again it suffices to prove the statements for $k = 2$, since for integers $k \geqslant 3$, $H_1 \boxbslash H_2 \boxbslash \ldots \boxbslash H_k$ is $((H_1 \boxbslash H_2) \boxbslash \ldots) \boxbslash H_k$, hence $H_1' \boxbslash H_2'$, and the result follows by induction.

From Lemma 5, we have that every H_i has at least one longest path without synchronising arcs if and only if $\ell(H_1 \boxbslash H_2 \boxbslash \ldots \boxbslash H_k) = \ell(H_1) + \ell(H_2) + \ldots + \ell(H_k)$, therefore as both H_1 and H_2 contain only longest paths with at least one synchronisation arc, both H_1 and H_2 do not contain a longest path without synchronising arcs. From this observation it follows that $\ell(H_1 \boxbslash H_2) \neq \ell(H_1) + \ell(H_2)$. By Lemma 4, $\ell(H_1 \boxbslash H_2) \leqslant \ell(H_1 \boxdot H_2)$, it follows that $\ell(H_1 \boxbslash H_2) < \ell(H_1) + \ell(H_2)$. Together with the observation that $\ell(H_1 \boxdot H_2) = \ell(H_1) + \ell(H_2)$ this gives $\ell(H_1 \boxbslash H_2) < \ell(H_1 \boxdot H_2)$. □

Lemma 6 is rather restrictive. We can loosen the requirements on two graphs containing only longest paths with synchronisation arcs, to one graph containing only longest paths with synchronisation arcs and another graph containing at least one longest path containing a synchronisation arc. The rationale behind it is that a longest path P_1 without a synchronisation arc, and a longest path P_2 with a synchronisation arc, both in H_1, combined with a longest path Q with a synchronisation arc in H_2, will lead to graphs consisting of the reduced weak synchronised products of P_1 and the part of Q up to the synchronisation arc, and the reduced weak synchronised products of P_2 and Q. It is obvious that $\ell(P_1 \boxdot Q)$ is smaller than $\ell(P_2 \boxdot Q)$.

Theorem 1. *Let H_i be an acyclic graph for $i = 1, 2, \ldots, k$, where $k \geqslant 2$. Then $\ell(\boxbslash H_i) < \ell(\boxdot H_i)$ if there exists H_n, H_m, $n \neq m, 1 \leqslant n, m \leqslant k$, such that each longest path in H_n, contains at least one synchronising arc and there is at least one longest path with a same labelled synchronisation arc in H_m.*

Proof. Again it suffices to prove the statements for $k = 2$, since for integers $k \geqslant 3$, $H_1 \boxbslash H_2 \boxbslash \ldots \boxbslash H_k$ is $((H_1 \boxbslash H_2) \boxbslash \ldots) \boxbslash H_k$, hence $H'_1 \boxbslash H'_2$, and the result follows by induction.

Let all longest paths of H_1 be of the structure $P = p_1 p_2 \ldots p_{k_1}$, without loss of generality, containing one synchronising arc a with label $\lambda(a)$, say from p_i to p_{i+1}. Let there be at least one longest path $Q = q_1 q_2 \ldots q_{k_2}$ of H_2 containing one synchronising arc a with label $\lambda(a)$, say from q_j to q_{j+1}. Note that $p_i p_{i_1}$ and $q_j q_{j+1}$ have the same label and therefore the same weight t. Let there be at least one longest path $R = r_1 r_2 \ldots r_{k_3}$ of H_2 containing no synchronising arc.

Let $P' = p_1 p_2 \ldots p_i, P" = p_{i+1} p_{i+2} \ldots p_{k_1}$, $Q' = q_1 q_2 \ldots q_j, Q" = q_{j+1} q_{j+2} \ldots q_{k_2}$. Then $\ell(P \boxbslash Q) = \ell(P' \boxbslash Q') + \ell(p_i p_{i+1} \boxbslash q_j q_{j+1}) + \ell(P" \boxbslash Q") = \ell(P' \square Q') + 1 \times t + \ell(P" \boxbslash Q") < \ell(P' \square Q') + 2 \times t + \ell(P" \boxbslash Q") = \ell(P' \boxdot Q') + \ell(p_i p_{i+1} \boxdot q_j q_{j+1}) + \ell(P" \boxdot Q") = \ell(P \boxdot Q)$.

Because both Q and R are longest paths of H_2, $\ell(Q) = \ell(R)$. Due to the synchronisation constraints, $\ell(P \boxdot R) = \ell(P' \boxdot R) = \ell(P') + \ell(R) < \ell(P) + \ell(R) = \ell(P) + \ell(Q) = \ell(P \boxdot Q)$.

These two results, $\ell(P \boxbslash Q) < \ell(P \boxdot Q)$ and $\ell(P \boxdot R) < \ell(P \boxdot Q)$, complete the proof of Theorem 1. □

Remark 6.2

We may have the case that there are no more H_n, H_m that can be combined in the manner of Theorem 1. Still further synchronisation is possible, if there exists $H_{m_i}, m_i \neq n$, where for each longest path in H_n, there is a longest path containing a synchronising arc in $\mathop{\square}\limits_{i=1}^{l} H_{m_i}, l < k$.

7. Conclusions

With Theorem 1, we have proved that if one wants to reduce the worst-case performance of periodic real-time parallel processes, one can combine processes, where all longest traces for at least one process must contain synchronising actions and at least one other process must contain at least one longest trace with a synchronising action. To reach this point we have introduced graph products that can help us to analyse and combine a number of parallel processes. We were able to identify the pathological case in a natural manner by introducing the weak synchronised product. This made it visible that a set of parallel processes may contain unwanted behaviour, for example a deadlocked state. We have shown in the proof of Lemma 4 and Remark 4.1, that we can filter out this unwanted or ill-defined behaviour.

We informally introduced the notion of a consistent and an inconsistent set of graphs (representing real-time periodic processes). The latter represents behaviour of processes that is unwanted, but might appear in a non-trivial process specification. From our proof, it follows that one can detect whether such a situation occurs in a process specification: one just has to find paths that shrink when the weak synchronised product is taken.

Finally, we have shown how to get to the synchronised product, which can be used to improve the worst case performance of parallel processes, and how processes might be combined on synchronising actions in order to obtain a performance gain.

7.1. Discussion

The performance gain is significant if the set of parallel processes will miss deadlines if not synchronised, but will meet its deadlines if synchronised. Whether such a significant performance gain is achieved by combining processes is not clear. Firstly, a tool that will produce a synchronised product of parallel processes based on the transformations described in our paper does not exist yet. Secondly, whether or not a significant performance gain is achieved by combining processes depends on the ratio of the context switch time and the calculation time of the processes itself; clearly this depends on the type of hardware and operating system used.

For context switches, Li [1] distinguishes between direct and indirect costs with respect to the processing power. The direct costs consist of issues like saving and restoring registers, translation table look aside buffer entries that need to be reloaded, flushing of the processor pipe-line, but also kernel code that has to execute. Indirect costs include cache misses caused when context switches to a process whose cache lines have been reused. Such costs may degrade performance in a significant way. Li [1] also shows that the average direct cost is $3.8\,\mu s$. The indirect cost varies from a few microseconds to more than one millisecond, all on a 2.0 GHz Intel Pentium Xeon CPU, with 512 kB L2 cache. The operating system is a Linux 2.6.17 kernel with Redhat 9.

Veldhuijzen [2] shows that the cost is on average $7.7\mu s$ on a 560 MHz Pentium IV processor, running under the QNX operating system. A typical control loop as used in [2] takes 70 μs. Together with the motion profile and the many context switches it takes up to 650 μs. This is well within the boundary of 1 *ms*, the period of a control loop. Veldhuijzen [2] gives a gain of 100 - 140 μs. This would mean a gain up to 15 to 20 %.

As a contrast, Ritson et al. [3] show context switch overheads for occam-π, under the KRoC CCSP multicore scheduler, of the order of 100 nanoseconds - often much less (around 30 nanoseconds). For such systems, the value of the transformations described occurs when the granularity of concurrency becomes too fine even for that language and scheduler – and our ambitions for ever-more complex behaviour from systems drive us in that direction.

7.2. Future Work

The graph products we introduced will form the basis for further research. One of the main aims of further study is to develop exact algorithms and heuristics for optimising the performance gain by combining processes. To get the set of all longest paths in a graph is exponential in n, as shown in Remark 3.4 and therefore not tractable. It is essential that *all* longest paths are found. It is obvious that, for example, a breadth-first search will give an answer to the length of the longest path; but this is not sufficient. If there is a longest path in graph H_1 that does not have a synchronising arc in common with a longest path in graph H_2, the synchronised product will have the same length as the Cartesian product and no gain is achieved. If such a situation exists in a hard real-time system, then, if the original parallel specification has a deadline-miss due to these two (interleaved) traces, also the specification represented by the synchronised product may have the same deadline-miss.

In our case, robotic applications, although the number of states and the number of actions in a process is limited, this may lead to the need for a heuristic giving a reasonable performance gain.

This research is restricted to periodic real-time applications, where the periods, release time and deadlines are the same. This can be extended to applications where this is not the case. An issue that will arise is the scheduling of the synchronised product, because this product will have internal deadlines.

Furthermore the aspect of memory usage is not taken into account. Combining graphs leads to a state space explosion, although synchronisation may reduce the magnitude of the explosion. To reduce such an explosion, it might be necessary to combine only a subset of the set of graphs representing the parallel process specification. Decomposition of the original graph into its prime factors, will give the optimal set of graphs from which synchronised products can be taken. This leads to a set of graphs that still fits in the available memory and has a maximal performance gain. Another application of the decomposition into prime products is in case the original specification does not fit in the available memory. It might not always be possible to just extend the available memory. As an example, imagine a robot, running around on Mars, needs a software update, but the model just does not fit in the available memory. It would be difficult to get a memory extension in place. The same is applicable in robots operating in, for example, nuclear fusion reactors. Maintenance under these (hot) circumstances of the robot is also difficult. Clearly these situations apply to much more complex applications than we are considering. For the above mentioned reasons we have to show the associativity and commutativity of our synchronised product. Furthermore we have to define the constraints for the prime factors of our synchronised product and give an algorithm that calculates such factors.

A valid question is: what is the maximum gain that can be achieved by combining processes within a certain amount of available memory? With that knowledge, we can improve the performance of the systems produced with tools like LUNA [13] and TERRA [14]. These issues are for future research.

Acknowledgements

We would like to express our gratitude to the anonymous reviewers for the very useful suggestions and comments.

References

[1] 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.

[2] B. Veldhuijzen. Redesign of the CSP Execution Engine. MSc thesis 036CE2008, Control Engineering, University of Twente, February 2009.

[3] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009. `http://www.cs.kent.ac.uk/pubs/2009/2928/`.

[4] Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

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

[6] S. Schneider. *Concurrent and Real Time Systems: the CSP Approach*. John Wiley Sons, Inc., New York, NY, USA, 1st edition, 1999.

[7] M. Aiguier, C. Gaston, P. Le Gall, D. Longuet, and A. Touil. A Temporal Logic for Input Output Symbolic Transition Systems. In *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, pages 8 pp.–, 2005.

[8] Caucal, D and Hassen, S. Synchronization of Grammars. In *Proceedings of the 3rd International Conference on Computer Science: Theory and Applications*, CSR'08, pages 110–121, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Youcef Hammal. A Component-based Approach for Consistency Checking of UML Dynamic Diagrams. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, SEA '07, pages 192–197, Anaheim, CA, USA, 2007. ACTA Press.

[10] S. Wöhrle and W. Thomas. Model Checking Synchronized Products of Infinite Transition Systems. In *Proc. 19th LICS, IEEE Comp. Soc*, pages 2–11. IEEE Computer Society Press, 2004.

[11] O. Oguz, J.F. Broenink, and A. H. Mader. Schedulability Analysis of Timed CSP Models Using the PAT Model Checker. In P. H. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen, and A. T. Sampson, editors, *Communicating Process Architectures 2012, Dundee, Scotland*, pages 65–88. Open Channel Publishing, August 2012. WoTUG-34.

[12] J Bang-Jensen and G.Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[13] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink. LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework. In P. H. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J. F. Broenink, and F. R. M. Barnes, editors, *Communicating Process Architectures 2011, Limerick, Ireland*, volume 68 of *Concurrent System Engineering Series*, pages 157–175, Amsterdam, November 2011. IOS Press BV. WoTUG-33.

[14] 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, *Communicating Process Architectures 2012, Dundee, Scotland*, pages 185–199, England, August 2012. Open Channel Publishing Ltd. WoTUG-34.

Appendix

In Listing 2, we give an example for the serialisation of two processes containing choice. Two processes synchronise over the actions a, c, and e. According the process specification of Listing 2, two traces can occur, $d \rightarrow c \rightarrow e$ and $a \rightarrow b \rightarrow e$. The last stage of Figure 12 shows the graph representing these two traces.

```
H1  =  (a → b → H1')
        □
        (d → c → H1')
H1' =  e → SKIP

H2  =  (a → H2')
        □
        (c → H2')
H2' =  e → SKIP
H   =  H1 {a,b,c,d,e}||{a,c,e} H2
```

Listing 2. Description of the `CHOICE` in two parallel processes.

Assuming that the weight of all arcs is 1, the graph consisting of the two components H_1 and H_2 has 8 vertices and a length $\ell(H_1) + \ell(H_2) = 5$, whereas the synchronized product $\ell(H_1 \boxbslash H_2)$ has 5 vertices and a length $\ell(H_1 \boxbslash H_2) = 3$. This example shows a gain for both the memory occupancy, as the performance of the application.

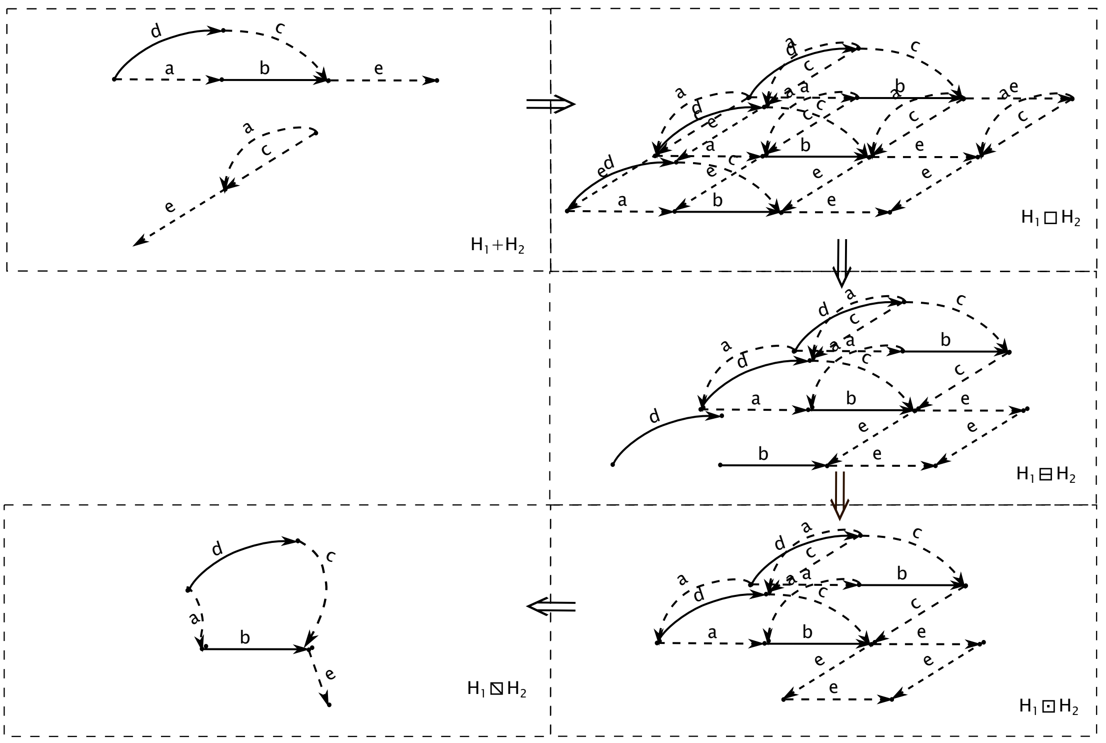

Figure 12. Choice in two parallel processes, from $+$ through $\boxbslash$.