Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014

Performance of Periodic Real-Time Processes: a Vertex-Removing Synchronised Graph Product

Antoon H. BOODE [1] and Jan F. BROENINK

Robotics and Mechatronics, CTIT Institute, Faculty EEMCS, University of Twente, The Netherlands

Abstract. In certain single-core mono-processor configurations, e.g. embedded control systems, like robotic applications, 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. Reducing the number of context switches decreases the execution time and thereby increases 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. These processes can be represented by finite directed acyclic labelled multi-graphs. The vertex-removing synchronised product of such graphs gives graphs that represent processes which have less context switches. To reduce the memory occupancy, the vertex-removing synchronised product removes vertices that are not reachable; i.e. represents states that can never occur. By means of a lattice, we show all possible products of a set of graphs, where the number of products is given by the Bell number. We finish with heuristics from which a set of graphs can be calculated that represents a set of processes that will not miss their deadline and which fits in the available memory.

Keywords. graph transformation, vertex-removing synchronised product, performance of real-time periodic processes, process algebra

Introduction

Embedded control systems, like periodic real-time robotic applications, can be designed using formal methods like process algebras. While designing, the designer distributes the required behaviour over up to several hundreds of processes. These processes very often synchronise over actions, e.g. to assert that a set of processes will be ready to start executing at the same time. Another example is mutual exclusion of resources, where a number of processes is allowed in their critical section.

Due to this synchronisation the application suffers from a considerable overhead related to extra context switches. We recognise two kinds of sources for these context switches, synchronisation over an action by two or more processes and a series of I/O actions between two processes, of which the former is the issue of this paper. The latter will be dealt with in future research.

[1]Corresponding Author: *Ton Boode, Robotics and Mechatronics, CTIT Institute, Faculty EEMCS, University of Twente, P.O. Box 217 7500 AE Enschede, The Netherlands*. E-mail: `A.H.Boode@utwente.nl`.

In [4] we define periodic real-time processes as finite directed acyclic multi-graphs, where these graphs are closely related to state transition systems. As, per action, there is a context switch, the longest path in such a graph is the most time consuming with respect to the context switch and therefore the worst case. We introduced in [4] a Vertex-Removing Synchronised Product (VRSP) to reduce the number of context switches. VRSP is based on the synchronised product of Wöhrle and Thomas [10], which is used in model-checking synchronised products of infinite transition systems.

The VRSP reduces the number of context switches and realises a performance gain for periodic real-time applications. This is achieved by (repetitively) combining two graphs representing two processes that synchronise over some action. The resulting process will have only one context switch per synchronising action, where the two processes each have a contest switch per synchronising action [4].

For our applications, short processes often consist of three or four sequential actions, where the first and the last action synchronise with other processes. For these applications a significant performance gain is expected.

An example of an overall system architecture[1] is described in Figure 1.

On Design level the designer gives a specification using some process algebra, in our case FSP [8][2]. Using VRSP this set of processes is transformed into a set of processes which will meet their deadlines and fit into the available memory. This new set of processes is transformed into Threads containing Finite State Machines (FSMs), where each FSM represents the behaviour of the corresponding FSP process.

The Synchronisation Software is the controller of the whole system. It decides whether a process is allowed to do a step in its FSM. To make hardware interaction possible, the Hardware Dependent Software contains as well a FSM, but related to an action/event is also some hardware interaction. This is also the case for Algorithmic Software, e.g. representing 20-SIM models, where together with a step in the FSM also some algorithm is executed.

In this manner there is a clear separation of concerns between the application and the hardware controlling software.

The contribution of this paper is an improvement on the design cycle and is illustrated in Figure 2.

In this cycle we start with the process specification written in some process algebraic form, in our case FSP[3]. By a transformation function $T, T : \{\text{Process Specification}\} \rightarrow \{H_i\}$, we get a set of finite directed acyclic labelled multi-graphs. Using the VRSP, the set of graphs is transformed into a new set of graphs. For this new set of graphs, either the processes that they represent meet their deadline and fit into the available memory, or there is no set of processes with strong-bisimular behaviour with respect to the original set of processes that will do so. In the former case, we again obtain a specification in some process algebraic form, in our case-study FSP, by using the inverse of the transformation function T, T^{-1}.

To be able to compose the set of graphs in a meaningful manner, the VRSP has to be idempotent, commutative and associative. For CSP this is all well known and because VRSP is similar to the CSP parallel composition, we can leave this for future research to be formalised.

Furthermore we investigate the number of products that are possible. These products are represented by a lattice. The lattice shows all possible products, which are partitions of the

[1]The first authors students at the InHolland University of Applied Sciences implement such a system as part of their curriculum.

[2]For our case-study the specification in FSP is more compact than e.g. CSP, although it lacks some of the nice features of CSP [6].

[3]We describe a product which is not restricted to FSP. In the future research the transformation function should be able to deal with any selected process algebra, which could easily be e.g. ACP [2] or CSP. Obviously the process algebra has to be a timed one, but this is not necessary for our case study.

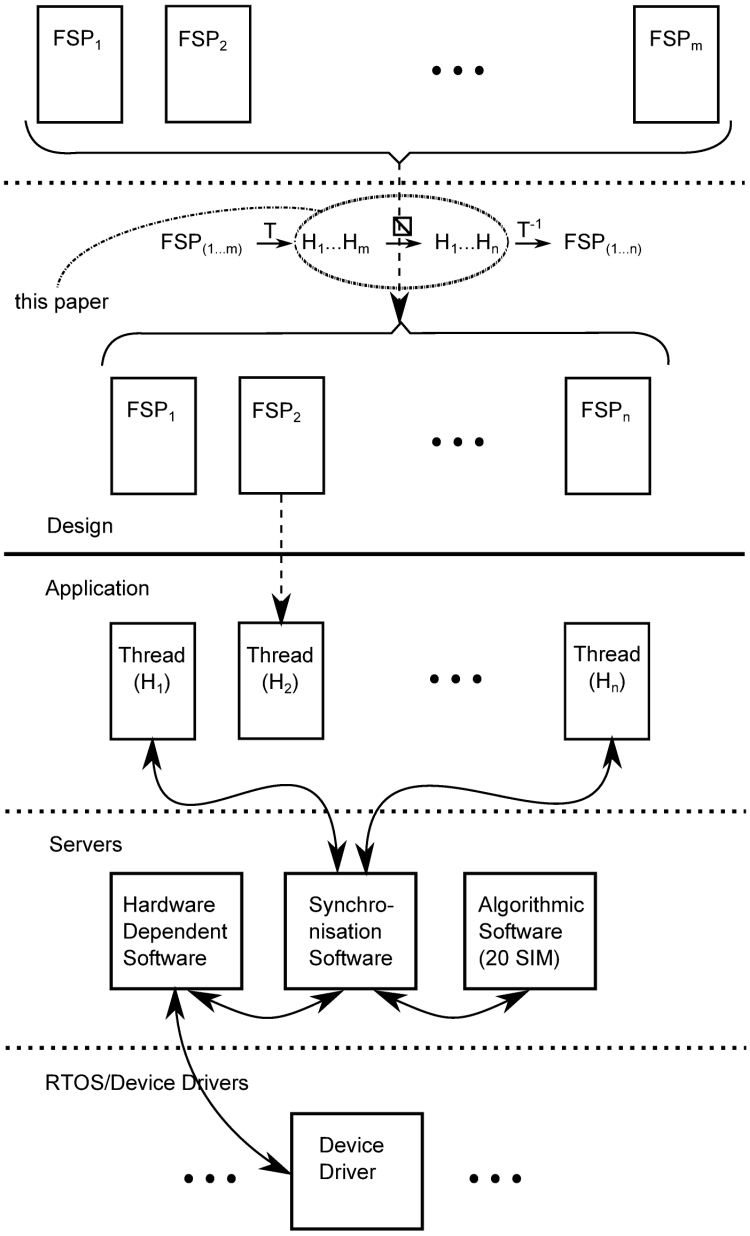

Figure 1. Overall System Architecture

original set of graphs. Because VRSP is exponential with respect to the number of vertices, we give heuristics that will calculate in polynomial time the VRSP of a set of graphs. Because the solution fulfilling the requirements is one out of exponentially many, these heuristics give no guarantee that the requirements will be fulfilled.

An introduction, giving some intuition for the used terminology, is given in Section 1. In Section 2 we give a lattice representation of all the combinations of graphs representing a process specification with respect to the summation over the synchronised products. In Section 3, Subsection 3.1 through 3.3 we give heuristics for calculating a solution for the initial set of processes that miss their deadlines or do not fit into the available memory by means of our VRSP. In Section 4 we present a case-study where the performance of a model of a Production Cell is optimised. In Section 5 we give the conclusions and we finish with a preview on the (near) future of our research in Section 6.

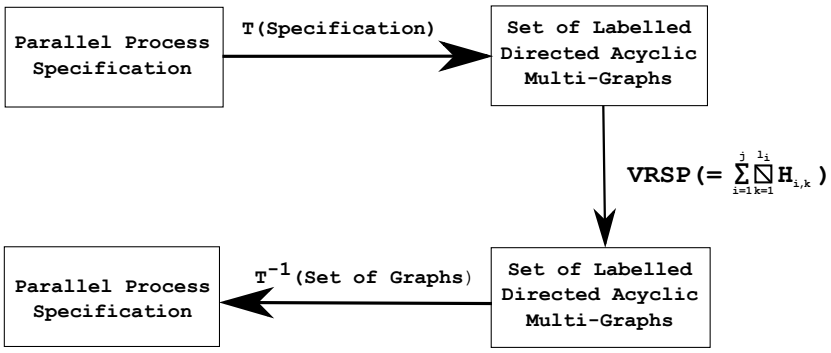

Figure 2. The design cycle

1. The Vertex Removing Synchronised Product

In this section we give an informal introduction to VRSP.

In [4] we have stated that "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, consisting of vertices, arcs between pairs of vertices and labels associated with the arcs.".

Processes which have no action in common, will execute interleaved when the generalised parallel operator is used. These actions are called asynchronous actions and are represented by asynchronous arcs. The interleaved execution of processes can be represented by the Cartesian product of the components (Figure 3). To build the Cartesian product of two components, each component is copied over all vertices of the other component and vice versa. In this manner a path through the Cartesian product will be identical to traversing in an interleaved manner through the components.

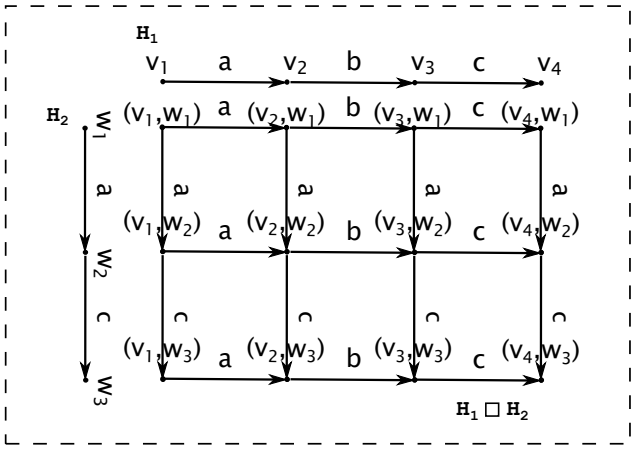

Figure 3. The Cartesian product of $H_1, H_2 \Rightarrow H_1 \square H_2$

Processes that have actions in common, will synchronise over these so called synchronous actions. The vertex removing synchronised product adds to the Cartesian product that whenever there is a quadrilateral of arcs with identical labels in the Cartesian product, this is replaced by one diagonal arc with the same label. These arcs that represent synchronous actions, are called synchronous arcs. In this manner there is a jump from one copy to the other for both participating components. Because these jumps will lead to unreachable vertices, all these vertices (and their arcs) will be removed from the product (see Figure 4.)

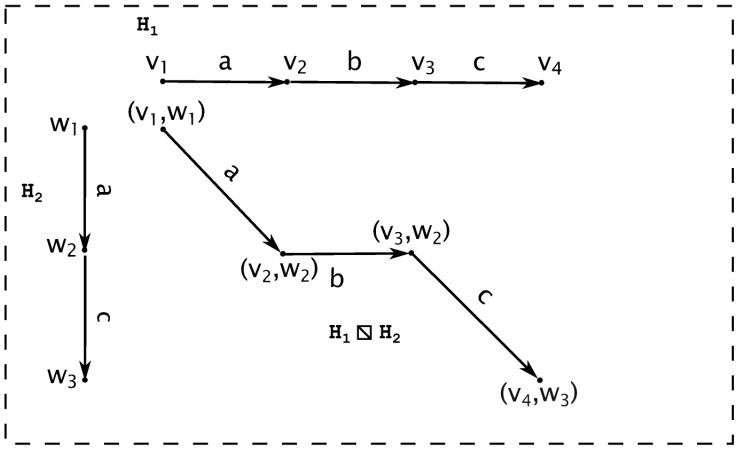

Figure 4. Synchronised product of $H_1, H_2 \Rightarrow H_1 \boxtimes H_2$

The two processes represented by the components H_1 and H_2 contain three and two actions. To execute the processes represented by these components, there will be five context switches. By using VRSP to create the process represented by $H_1 \boxtimes H_2$, the number of context switches is reduced by two. In our case study we see such improvements occur, because there at least a $tock$ action is part of every process and will be part of synchronisation. We use $\ell(H_i)$ to denote tha maximum length of a path in a component. This represent the longest trace in the process that is represented by the component. For a set of parallel processes this leads to a set of components, where the longest path of the graph is then a summation over the longest path of each component of the graph, $\sum_i \ell(H_i)$.

In general the processes represented by these components may suffer from deadlocks. Two processes, H_1 and H_2, are *consistent* if for every path in H_1 there exists a path in $H_1 \boxtimes H_2$, possibly after skipping some actions in $H_1 \boxtimes H_2$. These skipped actions then belong to H_2 and are asynchronous actions. Analogously for H_2 there exists a path in $H_1 \boxtimes H_2$, possibly after skipping some actions. These skipped actions then belong to H_1 and are asynchronous actions. Define a set of parallel processes to be *pairwise consistent* if every pair from them is *consistent*.

Deadlock freedom implies pairwise consistency, but not vice-versa. Deadlock is caused be a cycle of committed attempts to synchronise and, if that cycle has length greater than 2, the system will be pairwise consistent. If we reduce such a deadlock cycle down to just 2 processes through VRSP, those two processes will not be consistent - exposing the deadlock. So, without deadlock freedom, VRSP does not preserve pairwise consistency - see Figure 5.

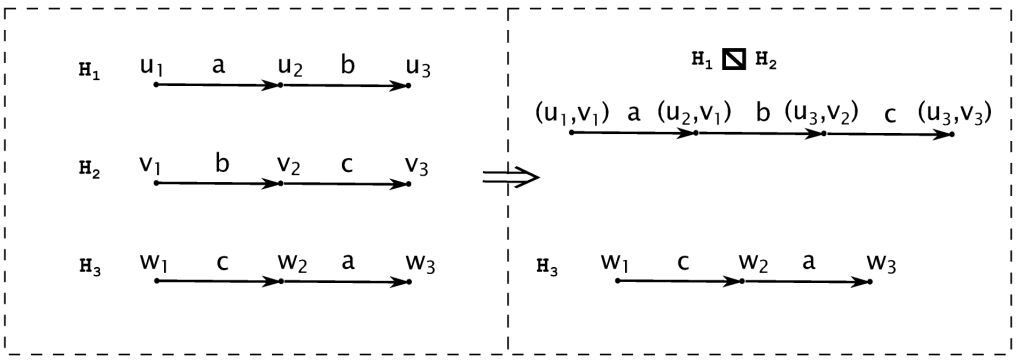

Figure 5. VRSP is not preserving pairwise consistency

However, we also need pairwise consistency for VRSP to be associative (see Figure 6). Without associativity, as we reduce concurrency through VRSP combination, the resulting system would depend on the order in which we chose to combine the processes!

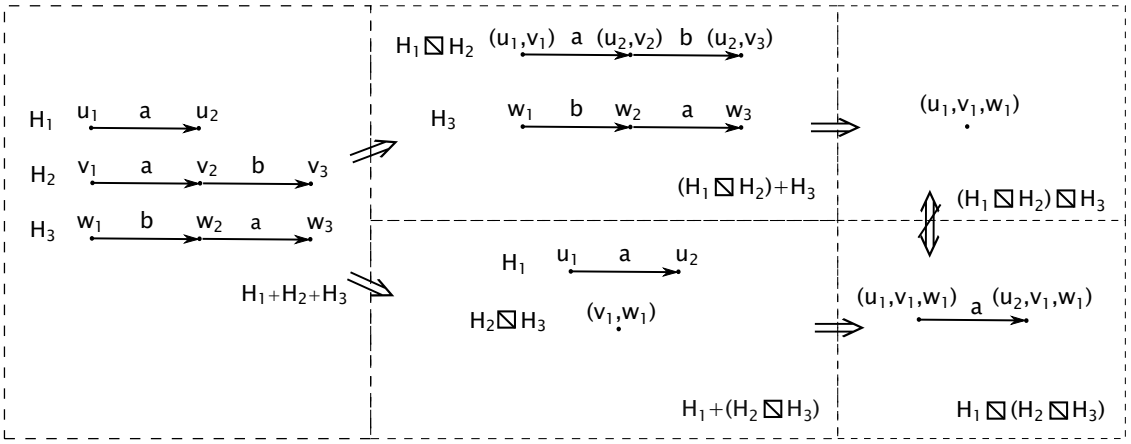

Figure 6. Non-associativity of not pairwise consistent components

If we can verify deadlock freedom in the original set of processes, for example through the use of a model checker such as [7], then we can combine processes using VRSP with no worries. Otherwise, we have to repeat checks for pairwise consistency before each VRSP combination. The first such check is an $O(n^2)$ operation. However, after combining processes A and B to get process AB say, we only need to check that AB is consistent with each of the remaining processes. We do not need to re-check pairwise consistency within those remaining processes, since they have not changed and the previous check still holds. So, subsequent checks for pairwise consistency are only $O(n)$. If we continue until a single process remains without any pairwise consistency check failing, the system must have been deadlock free.

2. Synchronised Product as a Lattice

Using VRSP in an effective manner, we have to calculate all possible combinations of products of subsets of the components representing the original process specification. So a partition of a graph H is a division of H into components in such a manner that these components form a union ($+$) of subsets, where the components in each subset are multiplied using VRSP ($\boxtimes$). The number of partitions of the graph $H = \sum_{i=1}^{n+1} H_i$, has an exponential distribution and is given by the Bell number, $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k, B_0 = 1$, [1][3].

Using the synchronised product we can create a partial order for each combination of $+$ and $\boxtimes$ [3]. Such a lattice has as an infimum the set $\sum_{i=1}^{n} H_i$ and as a supremum the set $\boxtimes_{i=1}^{n} H_i$. In this lattice, represented by a Hasse-diagram, two vertices are connected if (from top to bottom) in the graph represented by the upper vertex, two components are multiplied by our synchronised product, leading to a set of components represented by the lower vertex. Furthermore, as an example, there are only three paths to produce $V_{1011} \simeq H_1 \boxtimes H_3 \boxtimes H_4 + H_2$. Either $V_{1010} \simeq H_1 \boxtimes H_3 + H_2 + H_4$, $V_{0110} \simeq H_1 + H_2 \boxtimes H_3 + H_4$ or $V_{0011} \simeq H_1 + H_2 + H_3 \boxtimes H_4$. This is illustrated by the bold edges in Figure 7. In the same figure, $v_{0000} \simeq \sum_{i=1}^{4} H_i$ is connected to $v_{1100}, v_{1010}, v_{1001}, v_{1001}, v_{0110}, v_{0101}, v_{0011}$. The first position in the index of a

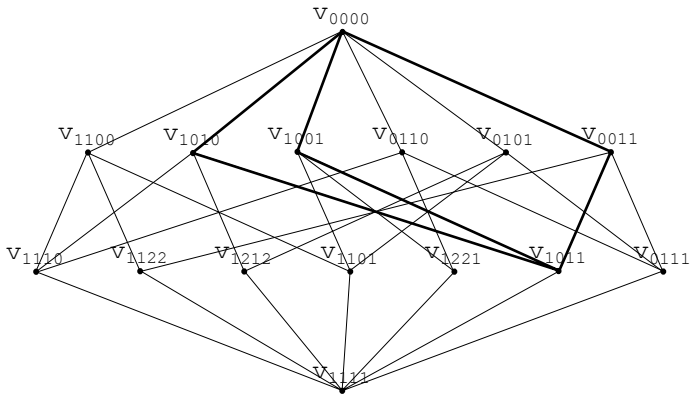

Figure 7. Hasse-diagram for H_1 through H_4. In bold the possible paths from v_{0000} to v_{1011}

vertex is H_1, the second H_2, and so on. Identical numbers in the index describe the relation ($+$ or $\diamond$) between the related components. Zero stands for the summation, numbers not equal to zero stand for the synchronised product, e.g. v_{10122} means $H_1 \diamond H_3 + H_2 + H_4 \diamond H_5$. The vertices in the lattice represent all possible combinations of the $+$ and $\diamond$.[4]

For a set of components $\sum_{i=1}^{n} H_i$, the depth of the Hasse-diagram is $n-1$. Each vertex represents a summation over synchronised products, $H' = \sum_{i=1}^{k} \bigdiamond_{j=1}^{l_i} H_{i,j}$, $\sum_{i=1}^{k} l_i = n$. A vertex is a solution if $\ell(H') \leqslant \mathcal{D}$ and $size(H') \leqslant \mathcal{M}$, where $\mathcal{D}$ is the deadline of the application and $\mathcal{M}$ is the available memory to store the data representing H'.

If a solution exists, it lies on a path from $v_{0\cdots0}$ (the infimum of the lattice) to $v_{1\cdots1}$ (the supremum of the lattice). Because there can be many paths from the source to the vertex representing a solution, the synchronised product of the graph H has to be commutative and associative, so the components in the graph H have to be pairwise consistent. Moreover each product of components has to be pairwise consistent with the other remaining components. Otherwise associativity further down the Hasse-diagram is jeopardised. Without deadlock freedom VRSP does not preserve pairwise consistency. Therefore the heuristic has to check whether the components are still pairwise consistent after every multiplication by VRSP.

3. Algorithms

Periodic real-time processes are defined as components of a finite directed acyclic multi-graph. The longest path in such a graph is the most time consuming with respect to context switches. If two processes are synchronizing over an action and one combines two such processes into one process, it reduces the process context switch overhead.

Unfortunately the number of possible products and therefore the number of choices follows the Bell number. Calculating all possible additions over products is not tractable for sufficiently large n (e.g $n > 20$).

A brute force algorithm that calculates for every vertex of the Hasse-diagram the synchronised products, is possibly not even in NP. Therefore, out of n components, the heuristics will always combine two components into one new component. In this (greedy) manner at most $n-1$ products have to be calculated.

There are several orders to synchronise the processes. All of them form some kind of path through the Hasse-diagram generated by all partitions under the synchronised product of the set of components $\sum H_i$. Out of many we consider three options, where the $calcAlgorithm$

[4] v_{1122} is equal to v_{2211}, which are related to respectively v_{1100} and v_{0011}.

in Algorithm 1 of Appendix B represents a choice out of the three algorithms described in Section 3.1 through 3.3.

Appendix B gives the various algorithms, which are all polynomial with respect to space and time.

3.1. The largest alphabetical intersection

A simple and polynomial time calculation is the Largest Alphabetical Intersection (LAI). For each pair of components the size of the synchronising alphabet is calculated. At each iteration the two components with the largest alphabetical intersection are multiplied. This gives no guarantee that a solution will be found that fits in the available memory. Also the length $\ell(H_i \boxtimes H_j)$ of H_i and H_j may be equal to the length of the sum $\ell(H_i + H_j)$ of H_i and H_j. Because we do not require that every longest path in H_i synchronises over some action with a full path in H_j. If the two components synchronise over arcs originating in the same vertex, it may be that another choice of components gives a better improvement of the performance of the represented processes. As shown in Figure 8, although the common label set is of size n, the length of the components is reduced by only one. It could even be that the product of two

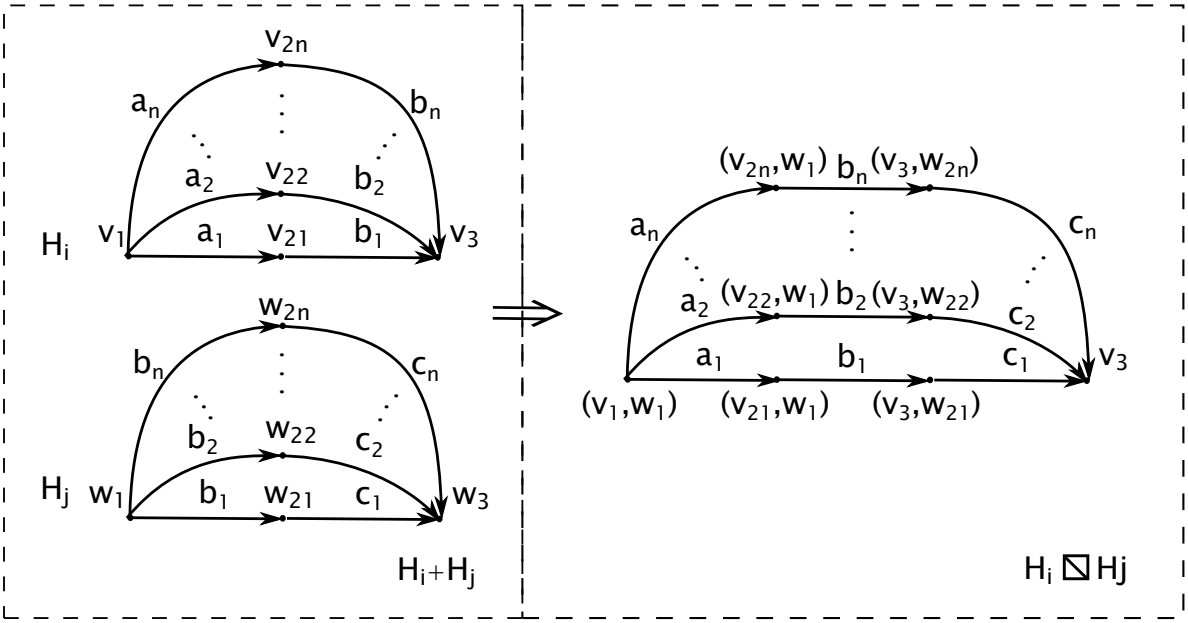

Figure 8. Synchronising over choice.

components, due to state-space explosion, is not calculable. LAI is a polynomial algorithm, given in Appendix B, Algorithm 5.

3.2. Maximising Synchronising Arcs

An adaptation of the algorithm in Section 3.1 is the maximisation of the number of synchronising arcs, Maximal Synchronising Arc set (MSA). The number of synchronising arcs is determined by their label. Without stating the algorithm we select those two components out of the set of components where the number of synchronising arcs is maximal.

Clearly this algorithm will only work for components where the set of component pairs with the largest synchronising set contains more than one element. Otherwise if one component has a synchronising arc set (pairwise with all other components), greater than the synchronising arc set of all other components (pairwise with all other components), then this component will become a greedy one. It will always be selected as one of the components for multiplication.

3.3. Minimising Not Synchronising Arcs

The disadvantage of LAI and MSA is that they do not optimise with respect to the Cartesian part of the synchronised product. The algorithm for minimising the not-synchronising arc set, Minimal Not-Synchronising Arc set (MNSA) tries to give the least vertex space explosion. Unfortunately this is not always the case. As an example, the components H_1 and H_2 that synchronise over arcs that are at the beginning (H_1) and arcs that are at the end (H_2), may have a very large asynchronous arc set, but the $H_1 \boxtimes H_2$ is linear with respect to the size of $H_1 + H_2$. Without stating the algorithm we have that the selected H_i and H_j have the smallest asynchronous arc set. The disadvantage, with respect to MSA, is that for the first iterations the improvement of the length of the components may be minimal.

4. The Production Cell Case Study

As a case study we use a Production Cell given in Figure 9 [5]. This Production Cell has

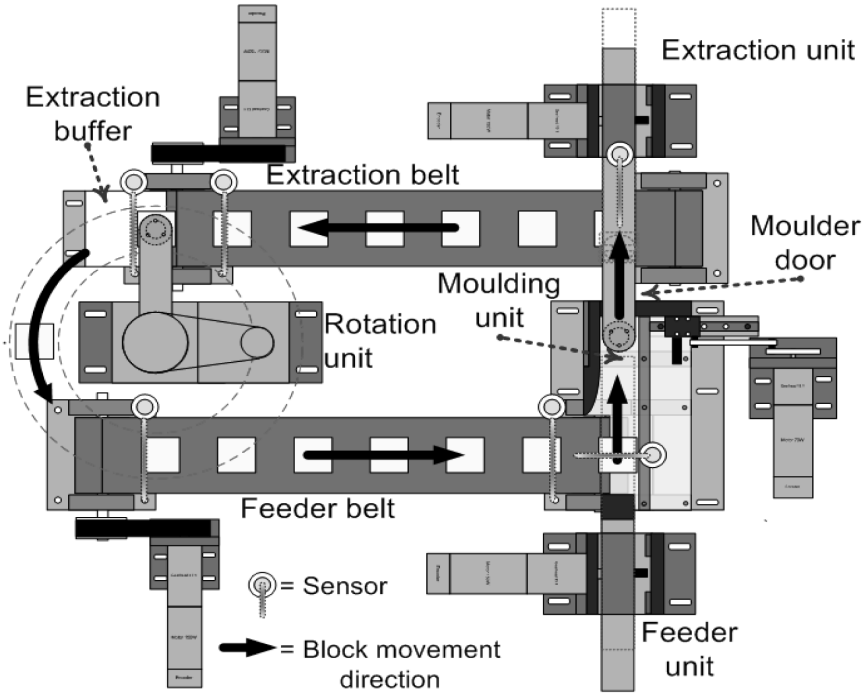

Figure 9. Production Cell.

six optical sensors and six motors. Each motor also contains an angle sensor. For the control loop, the duty cycle is 1 ms.

Veldhuijzen [9] shows that the cost for a context switch is on average $7.7\mu s$ on a 560 MHz pentium IV processor, running under the QNX1 operating system. We use this value to give an estimate of the average action-related overhead.

The memory occupancy is given in hypothetical units, where each unit represents the maximum amount of memory needed for a data-structure to store one vertex and its outgoing arcs. Clearly for our small example the memory occupancy is not really a problem, but in a real application with more than e.g. 100 processes, the exponential growth of memory needs may make the application not feasible.

To analyse the Production Cell, we give a model of the concurrent processes in Section 4.1, followed by a description of the processes in Section 4.2. The impact and an example of the synchronised product is discussed in Section 4.3. In Section 4.4 we analyse the

$$
\begin{array}{l}
||ProductionCell = \\
(feederBelt : Sensor \quad ||feederUnit : Sensor \quad ||mouldingUnit : Sensor \quad || \\
\ extractionUnit : Sensor \ ||extractionBelt : Sensor \quad ||rotationUnit : Sensor \quad || \\
\ feederBelt : Motor \quad ||feederUnit : Motor \quad ||angleRotationUnit : Sensor|| \\
\ extractionUnit : Motor \ ||extractionBelt : Motor \quad ||rotationUnit : Motor \quad || \\
\ extractionUnit : Magnet||angleRotationUnit : Magnet||MoulderDoor \quad || \\
\ Clock) \ /\{tock/\{feederBelt, feederUnit, mouldingUnit, extractionUnit, \\
\qquad\qquad extractionBelt, rotationUnit, angleRotationUnit\}.tock\}.
\end{array}
$$

Listing 1: Concurrent Processes of the `Production Cell`.

performance data and show the time and space related behaviour of the presented algorithms. In Section 4.5 we discuss the results so far.

4.1. Overview of the Concurrent Processes

For simplicity, out of the six angle sensors, we only model the angle sensor of the rotation unit. An overview of concurrent processes of the Production Cell is given in Listing 1. For the sixteen processes this means that in the worst case 60 action related context switches per period will be executed. As the duty cycle is 1 ms, this results in an average overhead of about 46%.

For the Production Cell, the six motors and six optical sensors and one angle sensor are represented by motor and sensor processes. The two magnets are represented by two magnet processes. Because of the real-time constraints we have a clock process containing a timer that expires every 1 ms. These sixteen processes lead to 10,480,142,147 vertices in the Hasse-diagram.

4.2. Process Description

In Listing 2 we give a description of the processes of the Production Cell. Where necessary a tock action transition is included in the model to avoid deadlocks not related to STOP. All processes synchronise at least over the tock action. This ensures that all processes will reach the final state represented by the sink of the related component.

MoulderDoor contains five tock actions, because it synchronises with feederUnit.Sensor, feederUnit.Motor and extractionUnit.Sensor. The components representing the processes MoulderDoor and feederUnit.Motor are given in Figure 10.

4.3. Synchronised Products of the Production Cell

The synchronised product of the processes MoulderDoor and feederUnit.Motor is given in Figure 11. It shows a reduction of the longest path of three. This means that by taking this product, there are three less context switches. The memory occupancy is extended by seven units (Appendix B, Table 2).

Other synchronised products show a reduction of the length of the longest path (by two) as well as a reduction of the memory occupancy (by six), like extractionUnit.Sensor and extractionUnit.Motor. In these cases the first action of one component synchronises with the almost last component of the other component. This leads almost to a linearisation of the two components.

If the tock action is the only event over which is synchronised, the synchronised product will suffer from a state space explosion[5].

[5]The tock action is at the end of each path.

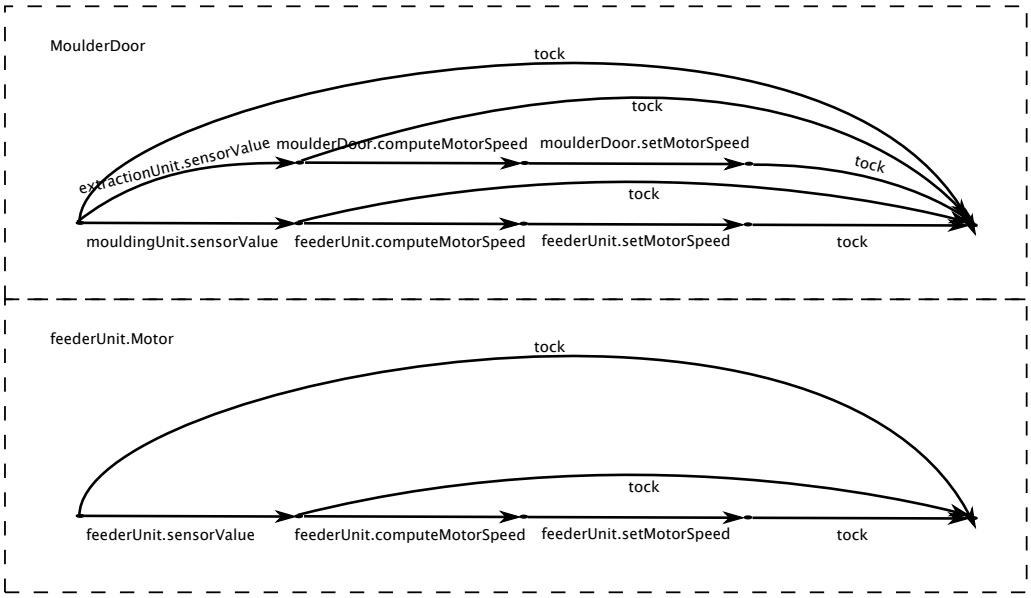

Figure 10. Components representing the parallel processes MoulderDoor and feederUnit.Motor

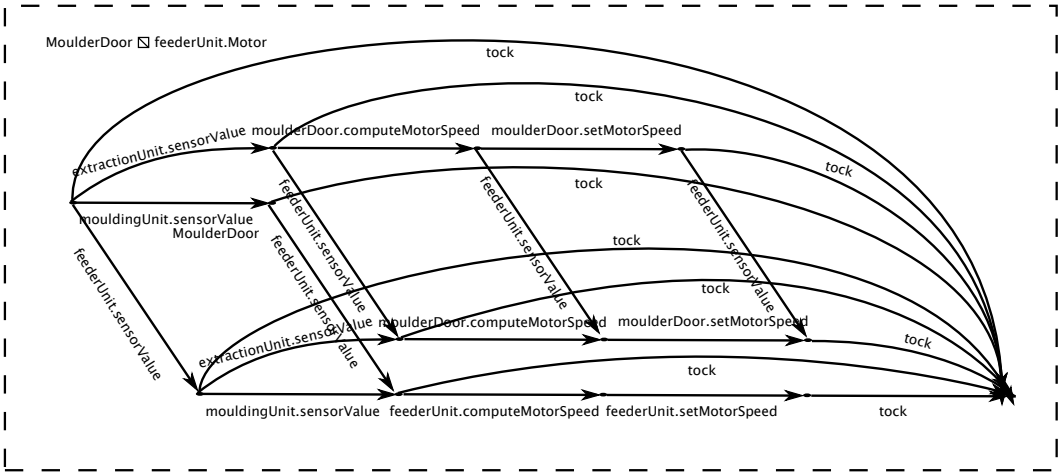

Figure 11. The Synchronised Product of the components MoulderDoor and feederUnit.Motor

4.4. Performance of the Production Cell

In Table 1 the memory occupancy and the longest paths of the components representing the processes in the Production Cell are given. The memory occupancy M is an indication of the amount of memory that will be used for the processes representing the components. It describes the usage of memory in relation to the space complexity. M consists of the number of vertices and the number of arcs used for $\sum_i \bigboxslash_j H_{i,j}$. The memory needed in practice will depend on the kind of data-structures that will be used for the implementation of the specification. The longest path, $\ell(H_i)$, reflects the maximum number of action related context switches for each process.

$$
\begin{aligned}
Motor &= (sensorValue \rightarrow (computeMotorSpeed \rightarrow setMotorSpeed \rightarrow tock \rightarrow MotorStop \\
&\quad |tock \rightarrow MotorStop) \\
&\quad |tock \rightarrow MotorStop), \\
MotorStop &= STOP.
\end{aligned}
$$

$$
\begin{aligned}
Sensor &= (readSensor \rightarrow calculateSensorValue \rightarrow (sensorValue \rightarrow tock \rightarrow SensorStop \\
&\quad |tock \rightarrow SensorStop) \\
&\quad |tock \rightarrow SensorStop), \\
SensorStop &= STOP.
\end{aligned}
$$

$$
\begin{aligned}
Magnet &= (sensorValue \rightarrow (angleZero \rightarrow contraction \rightarrow tock \rightarrow MagnetStop \\
&\quad |anglePI \rightarrow release \rightarrow tock \rightarrow MagnetStop \\
&\quad |tock \rightarrow MagnetStop) \\
&\quad |tock \rightarrow MagnetStop), \\
MagnetStop &= STOP.
\end{aligned}
$$

$$
\begin{aligned}
&MoulderDoor = \\
&(mouldingUnit.sensorValue \rightarrow (feederUnit.computeMotorSpeed \\
&\quad \rightarrow feederUnit.setMotorSpeed \rightarrow tock \rightarrow MoulderDoorStop \\
&\quad |tock \rightarrow MoulderDoorStop) \\
&|extractionUnit.sensorValue \rightarrow (moulderDoor.computeMotorSpeed \\
&\quad \rightarrow moulderDoor.setMotorSpeed \rightarrow tock \rightarrow MoulderDoorStop \\
&\quad |tock \rightarrow MoulderDoorStop) \\
&|tock \rightarrow MoulderDoorStop), \\
&MoulderDoorStop = STOP.
\end{aligned}
$$

$$Clock = (oneMilliSecondTimer \rightarrow tock \rightarrow STOP).$$

Listing 2: Description of the `Production Cell`.

i	$Process_i$	$\ell(H_i)$	M		i	$Process_i$	$\ell(H_i)$	M
1	$feederBelt.Sensor$	4	11		9	$feederBelt.Motor$	4	11
2	$feederUnit.Sensor$	4	11		10	$feederUnit.Motor$	4	11
3	$mouldingUnit.Sensor$	4	11		11	$extractionUnit.Motor$	4	11
4	$extractionUnit.Sensor$	4	11		12	$extractionBelt.Motor$	4	11
5	$extractionBelt.Sensor$	4	11		13	$rotationUnit.Motor$	4	11
6	$rotationUnit.Sensor$	4	11		14	$MoulderDoor$	4	19
7	$angleRotationUnit.Sensor$	4	11		15	$angleRotationUnit.Magnet$	3	12
8	$Clock$	2	5		16	$extractionUnit.Magnet$	3	12

Table 1. Worst case number of action-related context switches per process.

We use for the new concurrent process specification, the three algorithms that will calculate up to fifteen synchronised products. A calculation of the expected gain of the Production Cell specification is given in Appendix A, Table 2.

Based on Table 2, Figure 12 describes the behaviour of the three algorithms with respect to (the hypothetical values) M and D. The abscissa represents the length of the graph H. This stands for the number of action-related context switches. The ordinate represents the ^2log of the amount of memory used to store the graph related data.

For the Production Cell, $\mathcal{M}$ is the amount of memory available in the target system and $\mathcal{D}$ is the deadline for every period. The deadline $\mathcal{D}$ is 1 ms and is based on two parameters. Firstly, the calculation of the application and secondly, the overhead of the synchronised ac-

tions. The second one is represented by D. The dotted ellips shows the component compositions that fulfil the requirements.

Figure 12 shows that for our case study the MNSA algorithm has a slightly better performance with respect to memory utilisation, compared to the LAI algorithm. But the area within the ellipse fulfils the requirements and there LAI is slightly better than MNSA.

The MSA algorithm behaves poorly, because within the process specification the MoulderDoor process contains the most synchronising actions with respect to the other processes. In the component representing the MoulderDoor are five occurrences of the $tock$ action. For this reason the MoulderDoor (and, while traversing through the Hasse-diagram, its synchronised product with repeatedly the other components) component will always be chosen for synchronisation with remaining components. Figure 12 shows that the reduction of the $\ell(H)$ leads to a state space explosion from the fifth synchronised product onwards ($\ell(H) = 47$, ^{2}LOG(M)$\approx$10.7).

Of course it depends on the requirements of the application which vertex in the Hasse-diagram will be chosen as a basis to produce the new process specification. In our case study, this could arguably lead to the choice of $V_{1223345012334253}$ which is reached after 10 iterations using the LAI algorithm. The improvement is in this case approximately 16% of a duty cycle. The reduction of the number of context switches is slightly better than the number of context switches produced by MNSA. The best case gives an overhead reduction of approximately 20% of a duty cycle. Unfortunately this case suffers from a state space explosion and may not be tractable.

In practice a choice will be made, based on the question "How much memory do we have?". Based on that question the best reduction of the length of the components will be taken for the new process specification.

4.5. Discussion

In practice the number of parallel processes, and therefore the number of components of the graph H, is often limited to 15 or 20 processes. For 15 processes, there are $B(15) \approx 10^9$ nodes in the related lattice. But for 20 processes there are $B(20) \approx 5{\cdot}10^{13}$ nodes in the lattice. Depending on the speed of the computing system it may take several days to calculate the optimal solution out of all partitions for 20 processes (assuming the algorithm that calculates

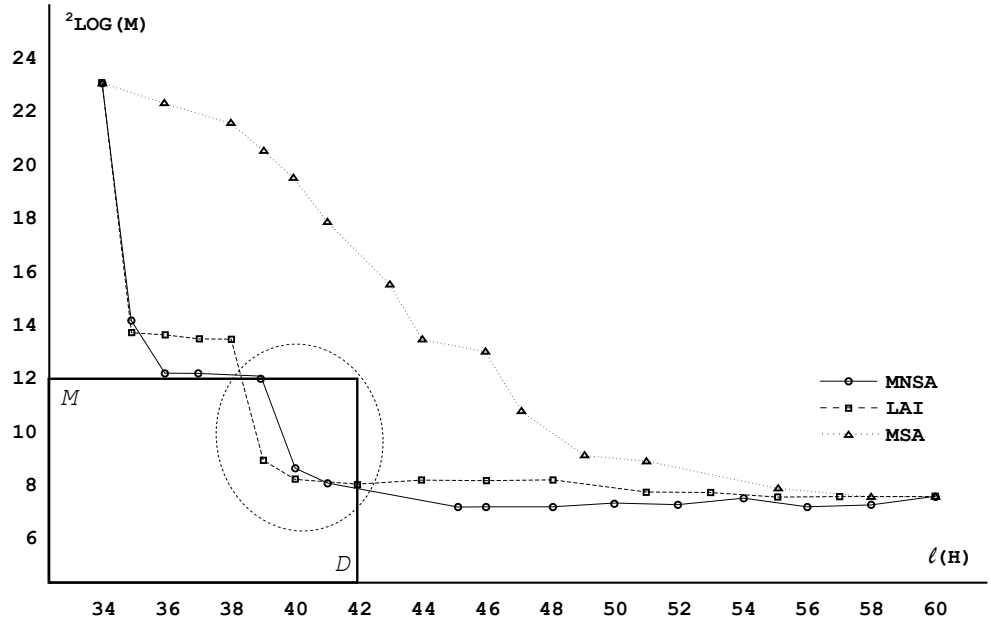

Figure 12. Performance of MNSA, LAI, MSA.

the optimal solution uses not more than the available memory to store the intermediate data). Each extra process will result in almost 10 times as much execution time. For this reason with the technology of today an upper limit of 20 processes is probably still tractable.

In our case the new set of processes is calculated off-line during the design process and forms no burden on an active real-time system.

5. Conclusions

A set of processes that does not meet its deadline or does not fit in the available memory can be transformed into a set of processes that will fulfil both requirements.

We have build a lattice that consists of all possible combinations of additions of products of components. The size of the lattice is exponential with respect to the number of components, representing the original set of processes and is given by the Bell number. In practice the number of parallel processes, and therefore the number of components of the graph H, is often limited to 15 or 20 processes. For 15 processes, there are $B(15) \approx 10^9$ nodes in the related lattice. But for 20 processes there are $B(20) \approx 5 \cdot 10^{13}$ nodes in the lattice. Depending on the speed of the computing system it may take several days to calculate the optimal solution out of all partitions for 20 processes (assuming the algorithm that calculates the optimal solution uses not more than the available memory to store the intermediate data). For this reason with the technology of today an upper limit of 20 processes is probably still tractable. Clearly for applications containing hundreds of processes heuristics have to be developed that will give an educated guess which partitions need to be calculated. In our case the new set of processes is calculated off-line during the design process and forms no burden on an active real-time system. In real-time systems, where on-the-fly processes are added to the system, our transformation will only work for the initial set of processes due to the extensive calculations that are necessary.

Because the components have to be pairwise consistent, to compose the original set of components, the designer is limited in his description of the system. But by using a model checker like e.g. FDR2 this should not be an issue.

We have developed heuristics in pseudo-code, which calculate from a set of components, a set of components that show a theoretical performance improvement, at the cost of an increasing memory occupancy.

6. Future Work

Several issues in our design cycle have not been addressed yet. They include the idempotency, commutativity and associativity of VRSP.

The classification of an algorithm that finds the optimal solution (a vertex in the Hasse-diagram) for the set of components is still open. Whether this is in NP, or even better decide whether it is NP-complete is also for future research. To proof the NP-completenes, some kind of Synchronised Product Problem (SPP) with its constraints has to be constructed. Then one has to show whether this SPP is in NP or is in EXSPACE.

But also the transformation functions T and T^{-1} are not defined yet. Allowing our processes to have different periods will introduce scheduling problems, that are avoided by requiring equal periods. Also the extension of our theory to cyclic components would strengthen the tool-chain. Another improvement would be the development of theory to factor the components in sub-components and use VRSP on these components. This may give a solution that is not available in the original set of components.

The goal is to give the designer of a set of processes full power of expression. The designer should not be bothered by issues related to the compliance of the design with the

available memory or to meeting deadlines. The developed theory forms the basis for future tooling essential to support the designer.

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 Sciences, Alkmaar, The Netherlands.

References

[1] E. T. Bell. Exponential polynomials. *Annals of Mathematics*, 35(2):pp. 258–277, 1934.
[2] J.A. Bergstra and J.W. Klop. Acpτ a universal axiom system for process specification. In Martin Wirsing and JanA. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, volume 394 of *Lecture Notes in Computer Science*, pages 445–463. Springer Berlin Heidelberg, 1989.
[3] G. Birkhoff. *Lattice Theory*. American Mathematical Society colloquium publications. American Mathematical Society, 1984.
[4] A. H. Boode, H. J. Broersma, and J. F. Broenink. Improving the performance of periodic real-time processes: a graph theoretical approach. In *Communicating Process Architectures 2013, Edinburgh, UK*, 35th WoTUG conference on concurrent and parallel programming, pages 57–79, Bicester, August 2013. Open Channel Publishing Ltd.
[5] M. A. Groothuis, R. M. W. Frijns, J. P. M. Voeten, and J. F. Broenink. Concurrent design of embedded control software. In T. Margaria, J. Padberg, G. Taentzer, T. Levendovszky, L. Lengyel, G. Karsai, and C. Hardebolle, editors, *Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM2009), Denver, United States*, volume 21 of *Electronic Communications of the EASST*, page 10, Berlin, November 2009. EASST.
[6] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.
[7] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement*. FDR2 User Manual, version 2.91 2010.
[8] Jeff Magee and Jeff Kramer. *Concurrency: State Models &Amp; Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.
[9] B. Veldhuijzen. Redesign of the CSP execution engine. MSc thesis 036CE2008, Control Engineering, University of Twente, February 2009.
[10] Stefan Wöhrle and Wolfgang Thomas. Model checking synchronized products of infinite transition systems. In *in: Proc. 19th LICS, IEEE Comp. Soc*, pages 2–11. IEEE Computer Society Press, 2004.

Appendix

A. Memory versus Deadline Table

With every iteration two components are multiplied using VRSP. So for $n = 0$ we have the set of graphs representing the original parallel specification. For $n = 15$ all components have been multiplied. For all three algorithms, the length of the graph, $\ell(H)$, which is the number of context switches in the representing processes. The function $m(H)$ calculates the number of vertices that is used by the graph H. It gives a measure what can be expected as far as the memory occupancy is concerned.

Iteration	Algorithms / $\sum_i \diagdown\!\!\diagdown_j H_{i,j}$	MNSA		LAI		MSA	
n	$Vertex$ in Hasse-diagram	$\ell(H)$	$\mathcal{M}$	$\ell(H)$	$\mathcal{M}$	$\ell(H)$	$\mathcal{M}$
0	$V_{0000000000000000}$	60	175	60	175	60	175
1	$V_{1000000010000000}$	58	169	-	-	-	-
	$V_{0000000001000100}$	-	-	57	182	-	-
	$V_{0000000000000101}$	-	-	-	-	58	183
2	$V_{1200000012000000}$	56	163	-	-	-	-
	$V_{1000000012000200}$	-	-	55	176	-	-
	$V_{0000000001000101}$	-	-	-	-	55	218
3	$V_{1230000012000300}$	54	177	-	-	-	-
	$V_{1200000012000200}$	-	-	53	197	-	-
	$V_{0100000001000101}$	-	-	-	-	53	338
4	$V_{1234000012400300}$	52	171	-	-	-	-
	$V_{1203000012300200}$	-	-	51	191	-	-
	$V_{0100000001100101}$	-	-	-	-	51	475
5	$V_{1234500012450300}$	50	165	-	-	-	-
	$V_{1223000012300200}$	-	-	48	300	-	-
	$V_{0101000001100101}$	-	-	-	-	49	546
6	$V_{1234560012456300}$	48	159	-	-	-	-
	$V_{1223400012340200}$	-	-	46	294	-	-
	$V_{0111000001100101}$	-	-	-	-	47	1,618
7	$V_{1234567012456370}$	46	153	-	-	-	-
	$V_{1223450012345200}$	-	-	44	288	-	-
	$V_{1111000001100101}$	-	-	-	-	46	7,855
8	$V_{1234567812456378}$	45	159	-	-	-	-
	$V_{1223456012345260}$	-	-	42	282	-	-
	$V_{1111000011100101}$	-	-	-	-	44	11,925
9	$V_{1223456712345267}$	41	283	-	-	-	-
	$V_{1223456012345263}$	-	-	40	292	-	-
	$V_{1111000011101101}$	-	-	-	-	43	54,133
10	$V_{1223345612334256}$	40	358	-	-	-	-
	$V_{1223345012334253}$	-	-	39	484	-	-
	$V_{1111100011101101}$	-	-	-	-	41	242,771
11	$V_{1222234512223245}$	39	4,381	-	-	-	-
	$V_{1222234012223242}$	-	-	38	11,978	-	-
	$V_{1111110011101101}$	-	-	-	-	40	367,945
12	$V_{1222233412223234}$	37	4,563	-	-	-	-
	$V_{1222234312223242}$	-	-	37	11,990	-	-
	$V_{1111111011101101}$	-	-	-	-	39	1,630,657

$Iteration$	Algorithms / $\sum_i \bigtriangledown_j H_{i,j}$	MNSA		LAI		MSA	
n	$Vertex$ in Hasse-diagram	$\ell(H)$	$\mathcal{M}$	$\ell(H)$	$\mathcal{M}$	$\ell(H)$	$\mathcal{M}$
13	$V_{1222233112223231}$	36	4,689	-	-	-	-
	$V_{1222233312223232}$	-	-	36	12,190	-	-
	$V_{1111111111101101}$	-	-	-	-	38	3,465,960
14	$V_{1222211112221211}$	35	18,318	-	-	-	-
	$V_{1222211112221212}$	-	-	35	13,734	-	-
	$V_{1111111111111101}$	-	-	-	-	36	4,810,387
15	$V_{1111111111111111}$	34	7,960,961	34	7,960,961	34	7,960,961

Table 2. Memory occupancy and worst case execution time.

B. Algorithms

In Algorithm 1, we describe the general structure of how to implement the algorithm, which contains a call to the specific calculation method $calcAlgorithm(H)$. In Algorithm 1 the subroutine $pairwiseConsistent(H)$ checks for a set of components $H = \sum_i H_i$ whether the VRSP over two of its components is still pairwise consistent with the other components. A breadth first search will solve this for each remaining combination. The subroutines $calcSize$ and $calcDeadline$ are a summation over the size of all vertices and their outgoing arcs. The subroutines $calcCartSize$ and $calcSyncProd$ are (worst case) the product of the vertex and arc sizes. Therefore these subroutines are polynomial with respect to space and time.

Algorithm 2 calculates the Cartesian product, Algorithm 3 calculates the intermediate product and Algorithm 4 calculates the synchronised product of two components H_i and H_j.

The pseudo-code of the Largest Alphabetical Intersection is given in Algorithm 5. Because the pseudo-code of the other two $calcAlgorithm(H)$'s is likewise straightforward, they are left out.

Algorithm 1 Calculating a General Synchronised Product Heuristic

Require: $H = \sum_{i=1}^{n} H_i$, $\mathcal{D}$ = $deadline$, $\mathcal{M} = available\ memory\ in\ target\ system$
1: $sizeH = calcSize(H)$
2: $deadlH = calcDeadline(H)$
3: **for** $i = 1$ to $n - 1$ **do**
4: **if** $sizeH \leqslant \mathcal{M}$ and $deadlH \leqslant \mathcal{D}$ **then**
5: **return** H
6: **else**
7: **if** $\neg pairwiseConsistent(H)$ **then**
8: **return** $\varnothing$
9: **else**
10: $(i, j) = calcAlgorithm(H)$
11: $H = (H \bigcup (H_i \bigtriangledown H_j)) \backslash (H_i \bigcup H_j)$
12: $sizeH = sizeH - calcSize(H_i \bigcup H_j) + calcSize(H_i \bigtriangledown H_j)$
13: $deadlH = deadlH - calcDeadline(H_i \bigcup H_j) + calcDeadline(H_i \bigtriangledown H_j)$

Algorithm 2 Calculating the Cartesian Product

Require: H_i, H_j
$V(H_i \square H_j) = V(H_i) \times V(H_j)$
$A(H_i \square H_j) = \varnothing$
for all $g_i, g_i' \in V(H_i)$ **and** $h \in V(H_j)$ **do**
  switch $(\delta(g_i, g_i'))$
  case $\Delta, 0$**:**
    break
  case 1**:**
    $A(H_i \square H_j) = A(H_i \square H_j) \bigcup \{(g_i, h)(g_i', h)\}$
    for all $\lambda(g_i g_i') \in L(H_i)$ **do**
      $L(H_i \square H_j) = L(H_i \square H_j) \bigcup \{\lambda((g_i, h)(g_i', h)) | \lambda((g_i, h)(g_i', h)) = \lambda(g_i g_i')\}$
  end switch
for all $g_j, g_j' \in V(H_j)$ **and** $h \in V(H_i)$ **do**
  switch $(\delta(g_j, g_j'))$
  case $\Delta, 0$**:**
    break
  case 1**:**
    $A(H_i \square H_j) = A(H_i \square H_j) \bigcup \{(h, g_j)(h, g_j')\}$
    for all $\lambda(g_j g_j') \in L(H_j)$ **do**
      $L(H_i \square H_j) = L(H_i \square H_j) \bigcup \{\lambda((h, g_j)(h, g_j')) | \lambda((h, g_j)(h, g_j')) = \lambda(g_j g_j')\}$
    break
  end switch

Algorithm 3 Calculating the Intermediate Product

Require: H_i, H_j
$V(H_i \boxtimes H_j) = V(H_i) \times V(H_j)$
$A(H_i \boxtimes H_j) = \varnothing$
for all $g_i, g_i' \in V(H_i)$ **and** $h_j, h_j' \in V(H_j)$ **or** $g_j, g_j' \in V(H_j)$ **and** $h_i, h_i' \in V(H_i)$ **do**
  switch $(\delta_{int}(g, g'))$
  case $\Delta, 0$**:**
    break
  case 1^a**:**
    $A(H_i \boxtimes H_j) = A(H_i \boxtimes H_j) \bigcup \{(h, g_j)(h, g_j')\}$
    for all $\lambda(g_j g_j') \in L(H_j)$ **do**
      $L(H_i \boxtimes H_j) = L(H_i \boxtimes H_j) \bigcup \{\lambda(h, g_j)(h, g_j') | \lambda(h, g_j)(h, g_j') = \lambda(g_j g_j')\}$
    break
  case 1^s**:**
    $A(H_i \boxtimes H_j) = A(H_i \boxtimes H_j) \bigcup \{(h, g_j)(h, g_j')\}$
    for all $\lambda(g_j g_j') \in L(H_j)$ **do**
      $L(H_i \boxtimes H_j) = L(H_i \boxtimes H_j) \bigcup \{\lambda(h, g_j)(h, g_j') | \lambda(h, g_j)(h, g_j') = \lambda(g_j g_j')\}$
    break
  end switch

Algorithm 4 Calculating the Synchronised Product

Require: $H_i \square H_j, H_i \times H_j$
$H_i \boxslash H_j = H_i \times H_j$
for all $g \in V(H_i \square H_j)$ **do**
  calculate $level(g)_{H_i \square H_j}$
for all $g \in V(H_i \boxslash H_j)$ **do**
  calculate $level(g)_{H_i \boxslash H_j}$
for all $g \in V(H_i \boxslash H_j)$ **do**
  if $level(g)_{H_i \square H_j} \neq 0$ *and* $level(g)_{H_i \boxslash H_j} = 0$ **then**
    for all $(g, g') \in A(H_i \boxslash H_j)$ **do**
      $A(H_i \boxslash H_j) = (A(H_i \boxslash H_j) \backslash gg')$
    $V(H_i \boxslash H_j) = (V(H_i \boxslash H_j) \backslash g)$
    for all $g \in V(H_i \boxslash H_j)$ **do**
      calculate $level(g)_{H_i \boxslash H_j}$
$L(H_i \boxslash H_j) = \varnothing$
for all $(g, g') \in A(H_i \boxslash H_j)$ **do**
  $L(H_i \boxslash H_j) = L(H_i \boxslash H_j) \bigcup \{\lambda(gg')\}$

Algorithm 5 Calculating the Largest Alphabetical Intersection

Require: $H = \sum_{i=1}^{k} H_i$

```
1: first  = 1
2: second = 2
3: num    = 0
4: for i = 1 to k − 1 do
5:     for j = i + 1 to k do
6:         newNum = |L(H_i) ⋂ L(H_j)|
7:         if (newNum > num then
8:             num    ← newNum
9:             first  ← i
10:            second ← j
11: return (first, second)
```